CS 352 Spring 2015

Compiler Project Part 2

Posted Feb. 13, 2015, Due March 1, 2015, 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

Part 2 of our compiler project is (i) to add new production rules to the Yacc program developed in Part 1 such that new kinds of variables are supported in our *miniscript* language; and (ii) to add semantic actions to the Lex/Yacc program such that the generated parser can interpret any miniscript program after verifying its syntactic correctness. (NOTE: the term "interpret" means that the statement is executed after the parser analyzes its syntax form. The execution is done by issuing appropriate C/C++ statements in the semantic actions, as illustrated in lectures.)

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.

1.1 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.

1.2 The Symbol Table

In order to evaluate the parameters that have numerical or string values to be printed, you will need to use semantic actions to evaluate expressions. The values of variables should be retrieved from a **symbol table** and your previous semantic actions should have stored the values in that symbol table when the corresponding assignment statements were interpreted.

1.3 Type checking and run time execution errors

Your interpreter may encounter type rule violations and run time execution errors. For programs that are compiled into machine code, these two kinds of errors are treated at different time. Most type rule violations are handled at "compile time" w/o execution the program (normally the program does not even know what the input will be). Run time execution errors, or simply run time errors refer to those

errors that cannot be predicted at compile time in general, because they may depend on the input. Examples are divided-by-zero, buffer-overflow, accessing uninitialized variables, and so on. The compiler normally is expected to catch as many type violations as it can during a single compilation, so that the programmer does not need to recompile the program so many times until all type rule violations are reported and fixed. Therefore, the compiler does not abort at the first finding of a type violation.

A fatal run time error usually causes the program to abort immediately, unless the program has prepared exception handling routines such that the program has a way to resume execution somewhere after the error is handled.

For scripts that are interpreted in a dynamic environment, e.g. web browsing, the interpreter is expected to continue execution whenever it can, even when type violations and run time errors are encountered. *We will follow this spirit for miniscript to an extent.*

Section 3 will specify the type rules in miniscript and how to handle type violations. It will also specify how to handle runtime execution errors.

2. New Language Features in Miniscript

We still have three kinds of statements as in Part 1: i) declaration statements; ii) assignment statements; and iii) document.write(….). However new kinds of variables are introduced below.

Like JavaScript, miniscript does not have explicit type declarations in the syntax. Instead, types are inferred by value assignment, which you may already notice from the example used for Part 1:

```
<script type="text/JavaScript">
var two = 2 ; var ten = 10
var linebreak = "<br />"

document.write("two plus ten = ")
var result = two + ten
document.write(result)
document.write(linebreak)

result = ten * ten
document.write("ten * ten = ", result)
document.write(linebreak)

document.write("ten / two = ")
result = ten / two
document.write(result)
var ID
ID = result
document.write(linebreak)
document.write(ID)

</script>
```

In this new part of the project, we introduce ***miniscript objects***. To declare an object without initializing its values, we can write in the following form:

```
var person = { }
```

In the above, white spaces are allowed between the braces, but not line breaks. Just like any other statements, the statement shown above may be followed by a semicolon or a line break. This statement declares person to be an miniscript object. However, we do not know how many fields are in this object. As a matter of the fact, we can introduce new fields to the object in an arbitrary order, e.g.

```
person.firstName = "Abe";
….. Some other statements …
person.lastName = "Lincoln"                    // notice that a semicolon is not required at the end, like before
…. More statements …
person.age = 150; person.eyeColor = "blue"+”grey”;   // For the meaning of “+”  operator, see below.
```

A miniscript object's field always have a ***field name*** (*which must be an ID*, whose regular expression is <letter> (<letter> | <digit>)*, as in Part 1), e.g. firstName, and *a field attribute*, which is either an integer or a string. A field value can never be another object.

Once a field is assigned a value, it can be used as an operand in a later statement just like a scalar variable. Your code will need to look up the symbol table to find the object first and the field next. **In the rest of the specification, when we refer to a variable, it may be a scalar or a field name.**

For convenience, fields can be initialized completely or partially, such as shown below

```
var person = {
   firstName:"Abe",
   lastName:"Lincoln",
   age:150,
   eyeColor:   "blue"+”grey”
};
```

Note that the spaces and new lines between different field definitions are allowed. The opening “{“ may or may not be followed by a field. The last field may or may not be followed by line breaks before the closing “}”. **These are the only exceptions to the rule that a statement does not spread across multiple lines. For simplicity, however, we do not allow a field definition to be split between different lines.**

This is to make it easier to treat newlines. Spaces and tabs are allowed before after the colon and the comma. Multiple fields are allowed to be in the same line:

```
var person = { firstName:"Abe",   lastName:"Lincoln", age:150,  eyeColor:"blue" };
```

or

```
var person = { firstName:"Abe",   lastName:"Lincoln",
        age:150,  eyeColor:"blue" };
```

Not all fields need to be initialized. For example, a later statement can be "person.height = 72;" even though the *height* field was not in the declaration for *person*.

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.

3.1 Operators and Operands

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. A variable may have an integer value or a string value, depending on its most recently assigned value. As your interpreter performs interpretation (i.e. execution of the statements), it needs to determine the type of each variable, e.g. by storing the type information in the symbol table for each variable.

An object name never has a value. Therefore **using it as a variable anywhere is considered a type rule violation.**

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.** However, as mentioned previously, the fields of an object may or may not be initialized when the object is declared. Therefore, after an object is declared, one can write to its field whether or not the particular field is claimed during the declaration, but a reference to an unclaimed field will still be reported as **a *violation* of the type rule.** That is to say, unclaimed object field can be modified (which is regarded as adding field) but cannot be accessed.

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. 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.**

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.

We do not allow two operands to have different types (i.e. one having an integer value and the other string value) for +, -, *, / operations. **Violation of this rule must also be reported.** The issue of recognizing mismatched types in operations is made complicated due to the existence of undefined types of sub-expressions. For example, after execution x = 3+"abc", x will have an undefined type. Suppose later a statement y = x+3+"xyz" is executed. By left associativity, x+3 is evaluated first, also getting an undefined type, which is then added to "xyz", still an undefined type. The seemingly explicit type violation by 3+"xyz" will go unreported.

An undefined type can be propagated far. Suppose x has an undefined type. The sequence of assignment statements a=x; b=a; c=b; d=c; ….. will access many variables that have undefined types. Reporting every encounter of an undefined type is not very helpful to the programmer, because these assignment statements may have nothing wrong by themselves. To avoid such excessive reporting, we establish the following type assignment rules:

- When a variable is declared without initialization, it has an undefined type until it is modified by a statement and obtains a defined type (see next bullet).
- The execution of assignment statement *x = expr* causes x to have the same type as expr, regardless if x is a declared variable.
- If x is not declared, x = expr still causes x to have the same type as expr, but a type violation is reported for x being undeclared. (Such type violation will be reported every time x is referenced, for simplicity of implementation.)
- If a variable is used as an operand before it is unsigned any value, it has an undefined type.
- The expression expr1 <op> expr2 will have an undefined type if
    - At least one of expr1 and expr2 has an undefined type (in which case no type rule violation is reported), or
    - If one operand is of the integer type but the other is of the string type, the expression has an undefined type. **This is a type violation.**
    - If any operand is of the string type for "-, *, /" operations, the expression has an undefined type. **This is a type violation.**

For example, suppose x has an undefined type. For expression “a”+1+x, your interpreter will report the type violation due to “a”+1. However, for expression x+1+”a”, no type violation is reported. This is an artifact due to left associativity of expression evaluation.

3.2. Reporting Type Rule Violations

If your interpreter finds any violation of the type rules listed in this section, it prints 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 interpretation must continue to interpret the rest of the miniscript program. If multiple violations of type rules are found in the same statement, the type violation messages are printed as many times, in separate lines.**

3.3 Handling Runtime Execution Error

If an expression uses a declared variable as an operand before that variable has been written by an assignment statement, we have a run time error. To help the programmer debug the program, a run time error message is printed in the following format:

Line <so-and-so>, <variable name> has no value

As mentioned before, the variable can be a scalar or an object field name. The interpretation must continue after reporting the error.

It is important to make a distinction between a variable that has never been written before being used as an operand and a variable that has been written but, due to the undefined type of the right-hand side expression in the assignment statement, the variable does not obtain a valid value. In the latter case, the variable may be a victim of type violation elsewhere. As we explained earlier, we do not want the propagation of the undefined type to generate “red herrings”, i.e. excessive type violation messages that are not helpful. In the former case, however, we have a genuine runtime execution error and must be reported so the programmer can fix it.

There are several ways to keep appropriate information in the symbol table such that the distinction can be made. For example, each variable, besides having a type and a value, can also be associated with a “written bit” in the symbol table. When accessing a variable that has an undefined type, we check to see whether its written bit is set. If not, we detect a “read before write” run time error. The students, however, are free to use other methods to make the required distinction.

Since the execution of miniscript statements is implemented by executing corresponding C statements in the Yacc/Bison program, other run time errors, such as divide by zero, may also happen. We do not

have special requirement for handling such errors. The program behavior depends on how C language is implemented on our hardware platform.

3.4 Printing Undefined Values

If a parameter in document.write(…) is an expression that has an undefined type, a **string "undefined" must be printed** in the position of the parameter.

3.5 Re-declaration of Variables

Re-declaration is allowed for any variable. However, unless the variable is initialized again, the variable is now considered to be never written. For example, after "var a = 100; …..; var a; " the variable a is considered to have never been written.

Re-declaration of an object will clear all of its former fields. (Your implementation is free to remove such fields if you wish.) Of course, the "var" statement that re-declares an object can have a list of new fields with initialized values.

4. 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. This re-grading part counts **20 out of 100 points** of Part 2.

For each new test case, points are assigned as following.

- 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.

5. Other Requirements

As in Part 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