Assignment #5: Statements and Identifiers

Due: May 10, 11:59pm

Note: Your modifications will be made to the files from the previous assignment. As such, the files must be named as before.

Overview

Educational Goals

- Control Flow
- State Maintenance

This project requires that you extend the parser and interpreter from the previous assignment to support statements, identifier expressions, and assignment expressions. You may begin with either your solution or the posted sample solution.

The `jsish` Syntax

The following extends the grammar from the previous assignment to add statements[1], identifier expressions, and assignment expressions to the `jsish` language.

Grammar

Ellipses are used to indicate the productions from the previous assignment. In some cases rules from the previous assignment have been updated; these changes are indicated by underlining.

program	→	...
statement	→	**expressionStatement** | **blockStatement** | **ifStatement** | **printStatement** | **iterationStatement**
expressionStatement	→	...
blockStatement	→	{ **statement*** }
ifStatement	→	`if` (**expression**) **blockStatement** {`else` **blockStatement**}$_{opt}$
printStatement	→	`print` **expression** ;
iterationStatement	→	`while` (**expression**) **blockStatement**
expression	→	...
assignmentExpression	→	**conditionalExpression** {= **assignmentExpression**}$_{opt}$
primaryExpression	→	... | `id`

Note Well: The grammar for `jsish` is based on JavaScript but differs in the rule for `assignmentExpression` due to issues with limited lookahead (the JavaScript grammar is not technically suitable for recursive descent parsing). The grammar for `jsish` is, however, suitable for recursive descent parsing, but the `assignmentExpression` rule allows for more syntactically "valid" expressions than the language really supports. Specifically, the grammar allows the left-hand side of an assignment to include structures that do not make sense in this context.

To handle this, your parser should first attempt to parse a `conditionalExpression` and then, assuming the parse was successful, if an assignment operator is found, verify that the left-hand side (the just parsed `conditionalExpression`) is valid. For this assignment, the only valid left-hand side will be an identifier. This check can be done for Part 2 by checking the returned abstract syntax tree.

[1] A subset of those supported by JavaScript.

Part 1: The Parser

Update the parser to recognize syntactically legal programs containing statements, identifier expressions, and assignment expressions and to report syntactically illegal programs.

Part 2: Abstract Syntax Tree

Extend the abstract syntax tree definition to represent statements, identifier expressions, and assignment expressions. Once done, modify the AST construction in the parser to construct (sub-)trees for these new features.

Unlike the majority of binary operators, the assignment operator is right-associative. Your parser should construct the subtree for nested assignment expressions accordingly.

Part 3: Echo

Update the AST printing functions to support statements, identifier expressions, and assignment expressions.

Part 4: Semantic Analysis

Update the interpretation functions to support statements, identifier expressions, and assignment expressions.

You will need to represent the "state" of a program by tracking the values bound to each identifier. This "state" can be represented as a hashtable that is passed to each evaluation function.

As part of this, you will need to remove the "echoing" of expression statements used to demonstrate functionality in the previous assignment.

Semantic Rules

Evaluation

- A variable (identifier) used before being assigned a value results in an error. Otherwise it will evaluate to the value bound to it.
- Assignment to a variable will bind a value to a variable. If the variable does not exist (i.e., it has not previously been assigned a value), then it is created.
- An assignment expression evaluates to the value assigned to the variable on the left-hand side.
- Assignment is right-associative.
- A print statement will evaluate the contained expression and then output the resulting value to standard output without any additional whitespace. Strings are printed without quotes and with escape sequences processed (by the ML output functions). Numbers are printed as one would expected, booleans print as `true` and `false`, and the undefined value prints as `undefined`.
- The if statement, after evaluating the guard expression, will only evaluate either the "then" or the "else" statement, not both.
- The while statement will evaluate the guard and, if true, evaluate the block statement. This will continue, as normal, until the guard expression evaluates to `false`.
- A block statement evaluates, in order, each of the constituent statements.

Dynamic Type Checking

- A variable (identifier) has type determined by its value. It is an error to use a variable before it is assigned a value.
- A print statement will accept expressions that result in any valid type, but the expression must be checked for validity.
- The expression guarding an if statement must be of boolean type.
- The expression guarding a while statement must be of boolean type.
- For this project, the left-hand side of an assignment must be an identifier.

Notes

- Test data with "correct" output will be given. Your output can differ in minor ways (e.g., syntax error message format) from this "correct" output yet still be correct; it is up to you to verify your code's correctness.
- Your program will be exercised by some shell scripts, which will be provided. These scripts depend on the exit status of your program. If your program detects a "serious" error, your program should use "OS.Process.exit" to terminate.
- Grading will be divided as follows.

Part	Percentage
1	15
2	10
3	10
4	65

- Get started **now** to avoid the last minute rush.

Full Grammar to This Point

program	→	**sourceElement***
sourceElement	→	**statement**
statement	→	**expressionStatement** | **blockStatement** | **ifStatement** | **printStatement** | **iterationStatement**
expressionStatement	→	**expression** ;
blockStatement	→	{ **statement*** }
ifStatement	→	`if` (**expression**) **blockStatement** {`else` **blockStatement**}$_{opt}$
printStatement	→	`print` **expression** ;
iterationStatement	→	`while` (**expression**) **blockStatement**
expression	→	**assignmentExpression** {, **assignmentExpression**}*
assignmentExpression	→	**conditionalExpression** {= **assignmentExpression**}$_{opt}$
conditionalExpression	→	**logicalORExpression** {? **assignmentExpression** : **assignmentExpression**}$_{opt}$
logicalORExpression	→	**logicalANDExpression** {`||` **logicalANDExpression**}*
logicalANDExpression	→	**equalityExpression** {&& **equalityExpression**}*
equalityExpression	→	**relationalExpression** {**eqOp** **relationalExpression**}*
relationalExpression	→	**additiveExpression** {**relOp** **additiveExpression**}*
additiveExpression	→	**multiplicativeExpression** {**addOp** **multiplicativeExpression**}*
multiplicativeExpression	→	**unaryExpression** {**multOp** **unaryExpression**}*
unaryExpression	→	{**unaryOp**}$_{opt}$ **leftHandSideExpression**
leftHandSideExpression	→	**callExpression**
callExpression	→	**memberExpression**
memberExpression	→	**primaryExpression**
primaryExpression	→	(**expression**) | `number` | `true` | `false` | `string` | `undefined` | `id`
eqOp	→	== | !=
relOp	→	< | > | <= | >=
addOp	→	+ | −
multOp	→	$*$ | / | %
unaryOp	→	! | `typeof` | −