Assignment #7: Objects & Garbage Collection

Parts 1 – 3 due: March 9, 11:59pm
Parts 4 – 5 due: March 16, 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

- Objects
- Garbage Collection

This project requires that you extend the parser and interpreter from the previous assignment to support objects and garbage collection of these objects.

You may begin with either your solution or the posted sample solution.

The `jsish` Syntax

The following extends the grammar from the previous assignment.

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.

statement	→	... | `gc ;` | `inUse ;`
expressionStatement	→	... ($*$ cannot begin with `function` or `{` $*$)
callExpression	→	**memberExpression** {**arguments** {**arguments** | `.` `id`}*}$_{opt}$
memberExpression	→	{**primaryExpression** | `new` **memberExpression** **arguments**} {`.` `id`}*
primaryExpression	→	... | `this` | **objectLiteral**
objectLiteral	→	{ {**propertyNameAndValueList**}$_{opt}$ }
propertyNameAndValueList	→	**propertyAssignment** {, **propertyAssignment**}*
propertyAssignment	→	`id` : **assignmentExpression**

As noted in the grammar, to distinguish between an **expressionStatement** and a **blockStatement**, an **expressionStatement** is not allowed to begin with { (even though an **expression** may).

The `.` operator is left-associative.

Part 1: The Lexer and Parser

Update the lexer (tokenizer) to support the two new keywords: `gc` and `inUse`. These will be used for part 5 of the assignment.

Update the parser to recognize syntactically legal programs according to the modified grammar.

Part 2: Abstract Syntax Tree

Extend the abstract syntax tree definition to represent the new language features. Recall that the `.` operator is left-associative.

Part 3: Echo

Update the AST printing functions to support the new language features.

Part 4: Semantic Analysis (without garbage collection)

Update the interpretation functions to support the new language features without garbage collection (i.e., without the evaluation of `gc` and `inUse`).

You will likely want to implement this part in steps. For instance, you might begin with simple manipulation of object literals, then implement the basic `new` functionality, and then move on to proper method invocation supporting `this`. After you have verified that object operations work, then add support for garbage collection.

This part introduces a new type of value (an `object`). Details are outlined below.

Semantic Rules

As evaluation begins, and before any of the user code is evaluated, a new object is created. This object is referred to as the "global object." This global object represents the top-level environment. As such, any change to the top-level environment modifies the properties of this object (this implies that the environment representing the "global" variables is the same as the table of properties for the "global object").

State Model

For this assignment the model of the program's state will be updated to include a representation of the heap for dynamic allocation and for garbage collection. As such, each "object" value will actually be represented as a new constructor (in your `value` datatype) for a "reference value". The heap will map addresses (integer values) to the actual object type (a new datatype). As such, a "reference value" must store the address of the referenced object.

The addresses used must start at 0 and increment by one for each allocation. For this model, your program will never reuse an address for an allocation.

Note well:

- You are allowed to use a single global heap. You might represent this as a mapping from integers (you can use `Word.fromInt` for the hashing function) to objects.
- You are allowed to use a reference (e.g., `int ref`) to store the current address (to allow incrementing).
- You are allowed, if you so choose, to store a reference within the object value to represent the mark.

Evaluation

- The creation of an object (by any means) will provide a reference to a newly allocated object stored in the heap.
- An object literal is evaluated by creating a new object and populating the new object's properties based on the initializers in the literal. Each `assignmentExpression` is evaluated and the result is bound to the associated `id`.

  The property assignments are evaluated in lexical order; this means that the last assignment to a duplicate `id` is the only value visible after the object is fully constructed.
- A `dot` expression must evaluate the expression to the left of the `dot` and, if the result is an object, access the value of the property specified by the `id`. If the object does not have the specified property, then the result is the `undefined` value.

- Take note that assignment to a `dot`ted left-hand expression works in the same way but instead of accessing the value of the property, the property's value is changed. (This also implies that the left-hand side of an assignment can now include a `dot` expression in addition to the previously required `id` expressions.)
- Objects are compared for reference equality (i.e., the addresses are the same).
- When a function is invoked, the `this` reference is made available for use. If the function is invoked on an object (i.e., using the `dot` notation), then `this` refers to the object to the left of the `dot`. A function invoked without an object is passed the "global object" as `this`[1].
- In top-level code (i.e., outside of any function), `this` is a reference to the "global object".
- When a `new` expression is evaluated, the specified `memberExpression` is evaluated to get a closure value. A new object is created and bound to `this` during execution of the closure with the specified arguments (evaluated as they would be for a normal invocation).

  If the invoked function explicitly returns an object, then that value is the result of the `new` expression. Otherwise, the result is the newly constructed object (the one bound to `this` during execution of the function)[2].

Dynamic Type Checking

- Assignments now allow, in addition to an indentifier, a left-hand side of a dotted expression.
- A `dot` expression requires an object as the target.
- The sub-expression in a `new` expression must evaluate to a closure value (to be used as the constructor).

Part 5: Garbage Collection

Helpful Functions: `HashTable.listItemsi, HashTable.remove, ListMergeSort.sort, String.concatWith, Word.fromInt`

With the previous part implemented, your interpreter now supports the creation and use of objects. Unfortunately, as the references to these objects go out of scope, the referenced object values will remain in the heap until the program terminates (in fact, by maintaining this heap, the garbage collector in ML cannot reclaim this memory either).

For this part you will implement a mark-and-sweep garbage collector to remove object values from the heap.

Due to the time available, and because bugs happen, garbage collection in this project will not be automatic (this is to prevent a bug in the collector from breaking evaluation in the previous part). Instead, you will support the `gc` statement to initiate garbage collection and the `inUse` statement to report the current status of the heap. The following details are provided for these two commands.

- Evaluating the `gc` statement must initiate a mark-and-sweep garbage collection pass. Additional details about the marking phase are provided below. After the sweep phase, print to `stdOut` a message of the form `RECLAIMED: <address>*` where `<address>*` is the comma-separated addresses (in ascending order) of those (garbage) objects removed from the heap.
- Evalulating the `inUse` statement must report the addresses of all objects still in use within the heap (i.e., this just reports what is in the heap). These comma-separated addresses are to be printed in ascending order in a message of the form `IN USE: <address>*`.

Marking Tips

As you think through the problem of marking, it is likely apparent that the marking phase must traverse the current environment (chain of tables) to find all heap references (the root set). While true, you must consider that this environment does not reflect the full root set. Consider the following advice.

[1] This is commonly considered a design flaw in JavaScript.

[2] This may seem like an odd special case, but can be useful for "factory methods". That said, it's still odd.

- The current environment represents what is in scope to the currently executing function, but does not necessarily represent all of the scopes accessible to function invocations that are still pending (i.e., those waiting for the current call to complete). Consider growing your state model (i.e., pass a tuple or record as your state) to include a list of all pending environments as represented by the call stack (i.e., when a function is called, add the environment of the caller to the call stack list).
- Marking an object should then continue with marking all objects to which the initial object (transitively) refers.
- Beware cycles in the object graph[3]!
- A closure that was returned from another function may contain an environment not otherwise represented in the call stack list (i.e., there is no outstanding call using that environment). As such, the stored environment should also be marked (i.e., traversed to mark the reference objects and environments within closures).
- Beware cycles formed by closures! A closure might store an environment that contains a reference to the closure. You should mark the individual (tables within the) environments as well; consider inserting a mark into each table as it is first encountered (and then remove it when done).
- There are some references to objects that may exist entirely internally to your implementation at the point when `gc` is evaluated. These objects are not referenced by any existing environment and, in fact, may never be referenced by an environment. Consider adding a list of pending values to your state model (distinct from the call stack).

  In general, such an object (value) should be added to the list of pending values if it might be used after the evaluation of another expression. Some specific examples include the evaluation of arguments as part of a function call, the evaluation of the member expression for a function call, the evaluation of the properties in an object literal, and, perhaps suprisingly, the evaluation of the first operand in a binary expression.

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	10
2	10
3	10
4	55
5	15

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

[3] This feels like an interview question.

Full Grammar to This Point

program	→	**sourceElement***
sourceElement	→	**statement** | **functionDeclaration** | **variableElement**
functionDeclaration	→	`function` `id` (**parameterList**) { **sourceElement*** }
parameterList	→	{`id` {, `id`}*}$_{opt}$
variableElement	→	`var` **variableDeclarationList** ;
variableDeclarationList	→	**variableDeclaration** {, **variableDeclaration**}*
variableDeclaration	→	`id` {= **assignmentExpression**}$_{opt}$
statement	→	**expressionStatement** | **blockStatement** | **ifStatement** | **printStatement** | **iterationStatement** | **returnStatement** | `gc` ; | `inUse` ;
expressionStatement	→	**expression** ; (∗ cannot begin with `function` or { ∗)
blockStatement	→	{ **statement*** }
ifStatement	→	`if` (**expression**) **blockStatement** {`else` **blockStatement**}$_{opt}$
printStatement	→	`print` **expression** ;
iterationStatement	→	`while` (**expression**) **blockStatement**
returnStatement	→	`return` {**expression**}$_{opt}$;
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** {**arguments** {**arguments** | . `id`}*}$_{opt}$
arguments	→	(**argumentList**)
argumentList	→	{**assignmentExpression** {, **assignmentExpression**}*}$_{opt}$
memberExpression	→	{**primaryExpression** | `new` **memberExpression** **arguments**}{. `id`}*
primaryExpression	→	(**expression**) | `number` | `true` | `false` | `string` | `undefined` | `id` | **functionExpression** | `this` | **objectLiteral**
functionExpression	→	`function` (**parameterList**) { {**sourceElement**}* }
objectLiteral	→	{ {**propertyNameAndValueList**}$_{opt}$ }
propertyNameAndValueList	→	**propertyAssignment** {, **propertyAssignment**}*
propertyAssignment	→	`id` : **assignmentExpression**
eqOp	→	== | !=
relOp	→	< | > | <= | >=
addOp	→	+ | −
multOp	→	∗ | / | %
unaryOp	→	! | `typeof` | −