Assignment 8 (Bonus)

CSCI 3136: Principles of Programming Languages

Due Apr 19, 2019

Assignments are due on the due date before 23:59. All assignments must be submitted electronically via the course SVN server. Plagiarism in assignment answers will not be tolerated. By submitting your answers to this assignment, you declare that your answers are your original work and that you did not use any sources for its preparation other than the class notes, the textbook, and ones explicitly acknowledged in the answers. Any suspected act of plagiarism will be reported to the Faculty's Academic Integrity Officer and possibly to the Senate Discipline Committee. The penalty for academic dishonesty may range from failing the course to expulsion from the university, in accordance with Dalhousie University's regulations regarding academic integrity.

The original plan for this assignment was to develop a Scheme-- interpreter. Given the very compressed time frame, I do not think this is reasonable anymore. This assignment asks you to construct merely an abstract syntax tree representing the structure of a Scheme-- program. Your goal is to represent the structure of your program and resolve references to names in your program.

Important note: The semantics that are implicit in the following definition of the abstract syntax tree deviate from those of normal Scheme. The goal is to discuss how to represent program structure and how to organize compile-time binding in most statically scoped languages. Scheme's strategy necessarily deviates from that of many languages because it allows new names to be introduced in a scope during run time. For example, it is possible to conditionally choose between having a function defined in a local scope or not, simply by enclosing this definition in an `if`-statement. A subsequent call of this function may then call the locally defined function (if the condition of the `if`-statement was true) or some function with the same name in some outer scope (if the condition was false). The definition of the abstract syntax tree below means that the function call will always refer to the local function name and will result in an error if the body of the `if`-statement was not executed and, as a result, no function was bound to the function name.

The abstract syntax tree (AST) you are to construct is composed of the following node types. Each AST node may have a set of *children* and a set of *attributes*.

SCOPE: A scope represents the whole program (there is exactly one such scope and it's the root of the tree) or a `let`-block.

Attribute: The list of named entities defined in this scope. There are two ways to introduce such named entities into a scope:

- Every variable defined in the variable definition part of a `let`-block (obviously) becomes a named entity in this `let`-block.
- Every name (including functions) defined in a scope using `define` becomes a named entity in the scope.

Children: The list of statements in the scope, ordered in the order they occur in the program text. Every "normal" expression should be represented by its corresponding abstract syntax tree node, as specified below. Every definition should be translated into an assignment of an expression to the named entity introduced by this definition. Specifically, remember that the syntax `(define (fun args) statements)` is just a more convenient version of `(define fun (lambda (args) statements)`. Thus, this definition should be translated into an assignment of the expression `(lambda (args) statements)` to the named entity `fun`.

FUNCTION: This represents a function (defined using `(lambda (...) ...)` or `(define (...) ...)`). Its structure is identical to that of a SCOPE node with one exception: There are *two* lists of named entities. The first list is the list of function arguments (which inside the function act no differently from any other variable). The second list is the list of all local variables introduced via `define` statements. The reason why these entities are stored in separate lists is that they need to be treated differently in the function's call sequence.

Note that, while a `(let (...) ...)` block is represented using a SCOPE node in the AST, a `(let name (...) ...)` block becomes an ASSIGNMENT of a FUNCTION to an IDENTIFIER with name `name` followed by a FUNCALL of the function with name `name`. See the Scheme slides if you do not recall this translation. The point is that named `let`-blocks are nothing but syntactic sugar

for this idiom of defining an inner function followed immediately by calling it, most often in order to obtain a tail-recursive function that simulates a loop.

SEQUENCE: This type of node is used to represent `begin` blocks. Its representation is identical to that of a SCOPE, except that it does not have any attributes. Every definition in a `begin` block introduces a name in the enclosing scope.

ASSIGNMENT: This represents the assignment of an expression to a named entity. It has no attributes. Its first child is an IDENTIFIER node representing the named entity that is defined by this assignment. THe second child is the expression to be assigned to this named entity.

FUNCALL: This represents a function call. It has no attributes. Its first chlid is an IDENTIFIER node that names the called function. The remaining children are, in order, the expressions, used as function arguments.

IF: An IF-node has two or three children: the condition, the then-expression and, possibly, an else-expression. There are no attributes.

COND: A COND-node represents a `cond` expression. Each child represents a branch of this `cond` expression. The order of the children needs to match the order of the branches in the `cond` expression. There are no attributes.

CONDBRANCH: A CONDBRANCH is a child of a COND node. It has two children: an expression that represents the condition of the branch and a SEQUENCE composed of the statements in this branch. There are no attributes.

QUOTE: This represents both types of quotation in Scheme--. It has no attributes and one child, the quoted expression.

NUMBER: This node has no children. Its attribute is the number it represents (stored as a string for simplicity).

CHAR: This node has no children. Its attribute is the character it represents (stored as a string for simplicity).

STRING: This node has no children. Its attribute is the string it represents.

BOOL: This node has no children. Its attribute is the string it boolean value it represents (stored as a string for simplicity).

IDENTIFIER: This node has no children. It has two attributes. The first attribute is the name of the identifier. The second attribute is the number of hops along the static chain necessary to reach the scope that contains the object identified by this identifier. For example, if you follow the list of SCOPE and FUNCTION nodes in the AST that are ancestors of this IDENTIFIER node, sorted from bottom to top and the first two nodes in this list don't have a named entity with the same name as this identifier but the third node does, then the number of hops is 2 (the first scope is the current scope that contains this expression and thus should not be counted as a hop). If the name is not found in any ancestor scope or function, the number of hops should be recorded as `UNDEFINED`. This means that this refers to some identifier that should be defined in the standard library (e.g., this may happen for references to standard functions such as "+", "`eq?`", etc.). The interpreter would have to do extra work to determine the correct binding for these identifiers.

As an example, the AST for our standard `fibonacci` example

```
(define (fibonacci n)
  (let fib ([prev 0]
            [cur 1]
            [i 0])
    (if (= i n)
        cur
        (fib cur (+ prev cur) (+ i 1)))))
```

should look like this:

```
Scope [fibonacci]
└─ Assignment
   ├─ Identifier [fibonacci; 0]
   └─ Function [n; fib]
      ├─ Assignment
      │  ├─ Identifier [fib; 0]
      │  └─ Function [prev cur i;]
      │     └─ If
      │        ├─ FunCall
      │        │  ├─ Identifier [=; UNDEFINED]
      │        │  ├─ Identifier [i; 0]
      │        │  └─ Identifier [n; 1]
      │        ├─ Identifier [cur; 0]
      │        └─ FunCall
      │           ├─ Identifier [fib; 1]
      │           ├─ Identifier [cur; 0]
      │           ├─ FunCall
      │           │  ├─ Identifier [+; UNDEFINED]
      │           │  ├─ Identifier [prev; 0]
      │           │  └─ Identifier [cur; 0]
      │           └─ FunCall
      │              ├─ Identifier [+; UNDEFINED]
      │              ├─ Identifier [i; 0]
      │              └─ number [1]
      └─ FunCall
         ├─ Identifier [fib; 0]
         ├─ Number [0]
         ├─ Number [1]
         └─ Number [0]
```

Note the translation of the `let`-block into a function and a function call instead of a SCOPE. This is because this is a named `let`-block. Also note the convention[1] that, in the printed output, attributes

[1]This is really just convention because, in a proper interpreter or compiler, you would normally never print the AST. It is an internal representation of the program structure that is used as the basis for generating the code to be executed.

should be enclosed in square brackets, list entries in an attribute that is a list of multiple values (e.g., the list of names declared in a given scope) are separated by spaces, and multiple attributes (e.g., the parameter list and the local variables of a function) are separated by semicolons. Please follow this convention to make your output easier to interpret for the marker.

You should modify your parser code from Assignment 5 (or the sample code I provide) to construct the AST for a given program. Normally, the compiler/interpreter has to check whether there are two or more definitions for the same name in a given scope and report and error if this is the case. You do not need to perform any such checks here.

Submission Instructions

Place your code in a folder `a8` of your SVN repository and submit using the appropriate combination of `svn add` and `svn commit`.