Project 1: Scheme Pretty-Printer

CSC 4101, Fall 2011

Due: 29 September 2011

For this programming assignment, you will implement a pretty-printer for Scheme in either C++ or Java. The code should be developed in a team of two using pair programming. Your pretty-printer will read a Scheme program from standard input, parse it, and print it back out onto standard output. The output program will be indented in a uniform style. (It won't really be all that pretty.) You should use an object-oriented programming style using inheritance and virtual functions where appropriate.

Structure of the Pretty-Printer

Like any tool that processes text, the pretty-printer needs to parse the input first and store it in an internal data structure, before it can print it in some indented style. The pretty-printer consists of the following parts:

1. a lexical analyzer that splits the input text into tokens,
2. a recursive-descent parser that analyses the structure of the input program and builds a parse tree,
3. a parse-tree traversal that pretty-prints the input program.

Lexical Analysis

In lexical analysis, the input file is broken into individual words or symbols. These words or symbols are then represented as *tokens* and passed to the parser. Your lexical analyzer needs to read ASCII characters from standard input and do the following:

1. discard white space (space, tab, newline, carriage-return, and form-feed characters),
2. discard comments (everything from a semicolon to the end of the line),
3. recognize quotes, dots, and open and closing parentheses,
4. recognize the boolean constants `#t` and `#f`,
5. recognize integer constants (for simplicity only decimal digits without a sign),
6. recognize string constants (anything between double quotes),
7. recognize identifiers.

For the precise definition of identifiers, see the Revised^5 Scheme Report and follow that specification:

- `http://www-swiss.ai.mit.edu/~jaffer/r5rs_4.html`

- http://www-swiss.ai.mit.edu/~jaffer/r5rs_9.html

Scheme does not distinguish between uppercase and lowercase characters in identifiers. It is, therefore, easiest for later parts of the pretty printer or the interpreter to convert all letters to lowercase.

The typical structure of a lexical analyzer is to write a function `getNextToken()`, that reads a character at a time from the input and returns the next token that it finds. Use a program structure like this (in C++ syntax):

```
enum TokenType {
  QUOTE, DOT, LPAREN, RPAREN, TRUET, FALSET, INT, STRING, IDENT
};

class Token {
  TokenType tt;
  // ...
};

class IntToken : public Token {
  int intVal;
  // ...
};

class StrToken : public Token {
  char * strVal;
  // ...
};

class IdentToken : public Token {
  char * name;
  // ...
};

class Scanner {
public:
  Token * getNextToken ();
  // ...
};
```

If a special character or a boolean constant is recognized in the input, the method `getNextToken()` returns an object of class `Token` with the appropriate token type set. If an integer constant, string constant, or identifier is recognized, the method `getNextToken()` returns an object of class `IntToken`, `StrToken`, and `IdentToken`, respectively. These tokens contain the integer value, string value, and the name of an identifier, respectively, so that the values are available to the printing routine.

Parser

The parser gets tokens from the scanner and analyzes the syntactic structure of the token stream. Since Scheme has a very simple grammar, parsing using a recursive-descent parser is not difficult.

The subset of Scheme that we will be working with for testing and for the interpreter in project 2 is defined by the following grammar:

```
exp  -> ()
     |  #f
     |  #t
     |  ' exp
     |  integer_constant
     |  string_constant
     |  identifier
     |  ( list )

list -> quote exp
     | lambda ( [ parm ] ) exp+
     | lambda identifier exp+
     | begin exp+
     | if exp exp [ exp ]
     | let ( bind* ) exp+
     | cond case+
     | define def
     | set! identifier exp
     | exp+ [ . exp ]

case -> ( test exp* )

test -> else
     |  exp

bind -> ( identifier exp )

def  -> identifier exp
     |  ( parm ) exp+

parm -> identifier+ [ . identifier ]
```

For details of the syntax of Scheme and the meaning of these constructs, you can refer to the Revised[5] Scheme Report,

- `http://www-swiss.ai.mit.edu/~jaffer/r5rs_9.html`

but just for parsing, you don't need to worry about the meaning of these constructs yet.

Since Scheme allows constructing code as data, we need to make the parser more general so that it doesn't complain about incorrect code inside a list. We therefore need two parsing steps. The recursive descent parser for project 1 will only need to recognize the much simpler language

```
exp  -> ( rest
     |  #f
     |  #t
     |  ' exp
     |  integer_constant
     |  string_constant
     |  identifier
rest -> )
     |  exp+ [ . exp ] )
```

and then build a parse tree. The second parse step, were we check the detailed syntax of the special forms will happen in the form of error checks in the interpreter we'll write for project 2.

When the main program requests an expression to be parsed, the parser needs to make parsing decisions without lookahead. This grammar is designed so you can easily build a recursive descent parser without lookahead. Inside the parser, for parsing `rest`, you will need a token of lookahead to make parsing decisions.

For the recursive descent parser, define a class `Parser` as follows:

```
class Parser {
public:
  Parser (Scanner *);
  Node * parseExp ();
  // ...
};
```

where `Node` is the root of the parse tree node class hierarchy.

The pretty printer is simple enough that you could print the input program to the output during parsing, i.e., without constructing a parse tree. However, we will extend the pretty-printer into an interpreter in project 2. For the interpreter we will need the parse tree. These parse trees will be our internal list representation. We will later use the pretty-printer in the interpreter as our printing routine for printing the result of interpreting an expression (which will be a parse tree again).

Parse Tree

For Scheme, the parse trees are really just regular Scheme lists. However, since we are not programming in Scheme, we need to implement the list data structure. The typical way to implement such a data structure in an object-oriented language is as a class hierarchy:

```
class Node { ... };
class Ident : public Node { ... };
class BoolLit : public Node { ... };
class IntLit : public Node { ... };
class StrLit : public Node { ... };
class Nil : public Node { ... };
class Cons : public Node { ... };
```

where the class `Node` is an abstract class.

Build your parse tree such that only a single object of class `Nil` and only two objects of class `BoolLit` get created. E.g., multiple occurrences of `Nil` should be pointers pointing to the same object. This will simplify the implementation of some of the built-in functions in project 2.

To make the code for cons-cells more modular and more object-oriented, we will further specialize the data structure by factoring out the printing code (and later the evaluation code) that is specific to a special form. This results in the following class hierarchy:

```
class Special { ... };
class Quote : public Special { ... };
class Lambda : public Special { ... };
class Begin : public Special { ... };
class If : public Special { ... };
class Let : public Special { ... };
```

```
class Cond : public Special { ... };
class Define : public Special { ... };
class Set : public Special { ... };
class Regular : public Special { ... };
```

A cons cell (an object of class `Cons`) will then contain an object of class `Special`. When constructing a cons cell, the constructor of class `Cons` will parse the list, build an object of the appropriate subclass of `Special`, and keep a pointer to that object in the cons cell.

Any code that is in common between all special forms can be kept in class `Special`. Any code for regular function applications or lists as data structures will be in class `Regular`.

These data structures are designed using the following Design Patterns. The `Node` class hierarchy is an instance of the Interpreter Pattern. The classes `Nil` and `BoolLit` are instances of the Singleton Pattern. And the `Special` hierarchy is an instance of the Strategy Pattern. In a production-quality interpreter, we would further implement class `Ident` using the Flyweight Pattern and the `print()` methods using the Visitor Pattern, but for our purposes that would add too much complexity. The printing code for the `Token` data structure is not implemented in an object-oriented style to get you started faster.

Pretty Printing

Print the output according to the following rules:

- constants and identifiers are printed directly,
- `Cons` expressions that are not special forms (i.e., regular lists) as well as `set!` special forms are printed in the style:

  ```
  (+ 2 3)
  (set! x (+ 2 3))
  ```

  The elements of a list are separated by a single space.
- a quoted expression is printed with a quote character followed by the printed representation of its argument, e.g., `'x` or

  ```
  '(1 2 3)
  ```

  Quoted special forms are printed as regular lists.
- the special forms `begin`, `let`, and `cond` are printed with the keyword immediately following the left parenthesis and with subsequent lines indented by four spaces each, e.g.,

  ```
  (begin
      (set! x 6)
      (set! y 7)
      (* x y)
  )
  ```

  Subexpressions of special forms, are printed as regular lists.
- the special forms `if`, `define`, and `lambda` are printed with the first two list elements on the same line and subsequent lines indented by four spaces each:

```
(define (fac n)
    (if (= n 0)
        1
        (* n (fac (- n 1)))
    )
)
```

Ok, this printing style isn't exactly pretty, but it's reasonably easy to generate. A better pretty-printer would get a lot more complicated.

The object-oriented style of structuring the pretty-printing code is to include a virtual method `print()` in each class of the parse tree node hierarchy as well as in each class of the `Special` hierarchy. I suggest you pass the current position on the output line as argument to `print()`. Also, for printing lists, you will need a boolean parameter indicating whether the opening parenthesis has been printed already or not.

For any expressions for which more than one of the above printing rules applies, the printing style is undefined. This means you can decide how to format it.

Administrative Stuff

Program this project in groups of two. I strong suggest that you work in a 'Pair Programming' style, i.e., that you always sit in front of the screen together and that one of you types.

Turn in a directory containing all the files needed to build your program as well as a Makefile (if you are programming in C++) and a README file. If you are programming in Java, make sure the main class is called `Main`.

Put your files into the directory `~/prog1` in your `cs4101xx` account and submit using

```
/classes/cs4101/cs4101_bau/bin/p_copy 1
```

(or using the command '`make submit`'). Do not submit executables, object files, or byte code files. Write the Makefile such that it produces an executable called `spp`. In the README file, briefly explain how you designed your program and what works and what doesn't. If there are problems with your submission, include any information that would make it easier for the grader to give you partial credit. Make sure that you mention who the group members are.

You can find a code skeleton in the directory `~cs4101_bau/pub/`, as well as on the course web page.

The hardest parts of this project are understanding the data structures and understanding object-oriented programming. To make this process easier, look at the scanner and token data structure first and implement the scanner. Later add the parse tree data structure and write the parser and the pretty printer in parallel for one language construct at a time.