CMPT 225 Assignment 5 (5%)

Submit your solutions by Friday, August 5, 2022 11:00pm.

0. [0 marks] *Orientation*

   In this assignment you will build two things: the first is a simple open addressed hash table (Question 1, `hash.h`); the second is a simple interpreter for Python-ish programs (Question 2, `pish.cpp`).

   The basecode in the care-package contains those two files, but also a supporting cast of other files. Here's a brief description of what's there.

   - `hash.h` — Question 1 directs you to implement a dynamic set using a hash table with open addressing and quadratic probing. Your code will replace the basecode in `hash.h`, which initially contains a functional, but not terribly efficient, implementation of a dynamic set. When you run `make`, it will build the executable `hashdemo`, which includes your `hash.h`.

   - `pish.cpp` — Question 2 directs you to implement an interpreter for a Python-ish programming language. Input to the executable `pish` will be a Python-ish program; you may find sample programs in the directory `samples/pish`. The output of `pish` should be the program's output—the expected outputs of the sample programs are also in `samples/pish`. We strongly recommend you use command-line redirection when you test your code, e.g., to run program 0, try, "`./pish < samples/pish/program0.pish`". You should remove and replace the basecode, which initially contains a pretty printer for Python-ish programs.

   - `Scanner.h` and `Parse.h` — You'll need to understand these header files to complete Question 2. Why? The first thing that `pish` will do is call `Parse(cin);`, which, if successful, will generate a *parse tree* for the Python-ish program. `Parse.h` will give you details about the tree nodes {`StmtsNode`, `StmtNode`, `ExpnNode`}; `Scanner.h` will give you details about each token, typically a number, variable, operator or keyword. For examples on how to access tree nodes and tokens, check out `testParse.cpp` and `testScan.cpp`.

   - You can safely ignore the rest of the provided files.

   One last note: Though there is a lot to read and digest here, there's nothing fundamentally difficult about hash tables nor tree traversals. In fact, the full solution takes fewer than 150 lines of code.

1. [40 marks] *Hash Tables* (`hash.h`)

   Implement an ADT for a `Set` of strings using a hash table with quadratic probing. The strings are composed of a nonempty sequence of letters, digits and underscores, but the first character is never a digit. This implementation is meant to be compatible for use with the identifiers in question 2, but if you don't complete this question, use the code in `set.h` instead.

   Hash Function

   You will use a modular arithmetic hash function where the modulus m is chosen from a list of primes.

   - Take the last 3 digits of your student number, e.g., 301156789 → 789.
   - Add 3000 to it, e.g., 789 → 3789.
   - Use the largest prime which is no bigger than this number, e.g., 3789 → 3779.

   3742 => 3739

To convert the string contained within `data->key` to an integer, use the character mapping shown in the following table, positionally convert to base-64, and then multiply by `scale` which is defined in the basecode as 225.

[*Note:* You multiply by `scale` to spread out common variable names like `i` and `j`.]

0	0	a	10	k	20	u	30	A	36	K	46	U	56	_	62
1	1	b	11	l	21	v	31	B	37	L	47	V	57		
2	2	c	12	m	22	w	32	C	38	M	48	W	58		
3	3	d	13	n	23	x	33	D	39	N	49	X	59		
4	4	e	14	o	24	y	34	E	40	O	50	Y	60		
5	5	f	15	p	25	z	35	F	41	P	51	Z	61		
6	6	g	16	q	26			G	42	Q	52				
7	7	h	17	r	27			H	43	R	53				
8	8	i	18	s	28			I	44	S	54				
9	9	j	19	t	29			J	45	T	55				

For example, say you wanted to hash the string "`Index`". Using the table above, this generates the sequence `44 23 13 14 33`, which, when interpreted in base-64, is the number

$$44 \cdot 64^4 + 23 \cdot 64^3 + 13 \cdot 64^2 + 14 \cdot 64^1 + 33.$$

Therefore, the string "`Index`" hashes to

$$[225 \times (44 \cdot 64^4 + 23 \cdot 64^3 + 13 \cdot 64^2 + 14 \cdot 64^1 + 33)] \texttt{ \% } m.$$

In the case where $m = 3779$, the hash value is 1448.

Your Job

Complete `insert(...)` and `search(...)` using this hash function and quadratic probing. To facilitate marking your code, `insert(...)` has a return value: the table index of the location of the newly inserted key. It should return -1 on the rare occurrence that the quadratic probe fails to find an empty table slot.

Note: In the interest of expedience, you will *not* implement `delete(...)`, dynamic table expansion, copy constructor or `operator=`. They are important things to implement, but you have already completed similar implementations in your other Assignment work.

2. [60 marks] *Python-ish Interpreter* (`pish.cpp`)

For this problem, you will write the various parts of an interpreter for a Python-ish language.

Usage: The command `./pish < program.pish` should produce the output of the Python-ish program, i.e., the outcome of any `print` statements or run-time errors **and nothing else**.

The main differences between this pared-down language and real Python are:

- There is only one data type: 32-bit signed integers.
- There are only four types of statements: print statements, assignment statements, if statements and while statements.
- Real Python uses newlines and indentation to denote the end of a statement and/or membership to a block of statements, but your Python-ish Scanner/Parser are not that sophisticated. Therefore, we use the familiar `;` and `{ }` from C++/Java instead.

Statements and Expressions

The grammar for Python-ish describes both *statements* and *expressions*. A statement is one of:

- `print` *expression* `;`
  Displays the value of *expression* terminated by a newline.
- *identifier* `=` *expression* `;`
  Stores the value of *expression* into the variable named by the string *identifier*. If *identifier* already has a value, then it is overwritten by the new one.
- `if` *expression* `{` *statements* `}`
  If the value of *expression* is nonzero, then executes the sequence of *statements* exactly once.
- `while` *expression* `{` *statements* `}`
  If the value of *expression* is nonzero, then executes the sequence of *statements*. After the execution of *statements*, tests *expression* again, and if it is nonzero, executes *statements* again. Repeats this test-execution sequence until *expression* is zero.

A Python-ish expression allows a much richer set of operators than those from Assignment 2. The operators, listed in order of highest to lowest precedence are:

- `( )` — brackets
- `+ -` — unary plus or minus
- `* /` — multiply or divide
- `+ -` — add or subtract
- `< > <= >= == !=` — relative operators
- `not` — unary logical negation
- `and` — logical and
- `or` — logical or

A call to `Parse(cin);` will return a parse tree for a Python-ish program from standard input. (There are 6 sample Python-ish programs in the `samples` directory.) If the input is not valid Python-ish, `Parse()` will generate an exception.

The root node of the parse tree is a tree node of type `StmtsNode *`, which represents a sequence of Python-ish statements. The first statement in the sequence is the left child of the root node; the rest of the sequence is contained in the subtree of the right child. When a `StmtsNode *` is `NULL`, it represents an empty sequence of statements.

A `StmtNode *` represents one of several types of statements. To tell what kind of statement it is, examine its `->tok` attribute:

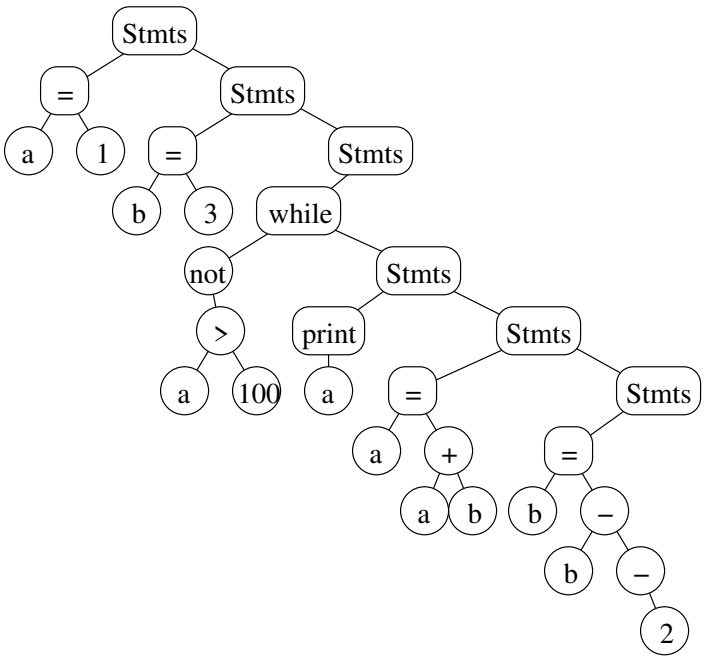
- `printtok` means it's a print statement;
- `asgntok` means it's an assignment statement;
- `iftok` means it's an if statement; and
- `whiletok` means it's a while statement.

In all cases, the `->expn` attribute refers to a tree node of type `ExpnNode *`, which is an expression tree to be evaluated and used by the statement. It is a standard expression tree, which means that a postorder evaluation will yield its value. All leaves are either integers or identifiers, and all interior nodes have two children except for the unary operators `-` and `not` (`mitok` and `nottok`) which have `NULL` for their left child. (The unary `+` is eliminated by the parser.)

The `->ident` attribute is used only in an assignment statement, in order to hold its left-hand-side; and the `->stmts` attribute is used only in an if statement or a while statement, in order to hold the block of statements which are to be conditionally executed.

As an example, here is a coding listing and graphical representation of `program4.pish`.

```
a = 1;
b = 3;
while not a > 100 {
    print a;
    a = a + b;
    b = b - -2;
}
```



A run of the sample `testParse` will generate a pretty-printed version of the program using the recursive subroutines `printStmts(...)` and `printE(...)`.

Some Notes About Python-ish

- All of the operators `*/+-` behave the same in both Python-ish and in C++ except for `/` (integer division). In Python-ish, the expression a/b behaves as follows:
  - When b is positive, a/b results in q, where $a = b \cdot q + r$ and r is an integer such that $0 \leq r < b$ (i.e., r is nonnegative).
  - When b is negative, a/b results in q, where $a = b \cdot q + r$ and r is an integer such that $b < r \leq 0$ (i.e., r is nonpositive).

  For examples, `7 / 3` results in 2 (which is the same as C++); `-7 / 3` results in -3 (which is different than C++); `7 / -3` results in -3 (which is different than C++); and `-7 / -3` results in 2 (which is the same as C++).
- All the relative operators `< > <= >= == !=` are not associative, so Python-ish prevents you from chaining two or more together. This means the statement `2 < x < 10` is a parse error. (Use `2 < x and x < 10` instead.)
- All relative operators evaluate to either 0 (if false) or 1 (if true).
- `not x` evaluates to 1 if the value of `x` is 0; otherwise it evaluates to 0.
- `x and y` evaluates to 0 if either operand is 0; otherwise it evaluates to 1.
- `x or y` evaluates to 0 if both operands are 0; otherwise it evaluates to 1.
- Python-ish has no scoping rules, i.e., there are no local variables for sub-blocks of statements.

(a) [40 marks] Write a program that interprets and executes programs written in Python-ish. You should use the hash table you wrote from Question 1, but if it is not finished, you can use the `Set` provided within `set.h` instead.

Complete your code in `pish.cpp`.

(b) [20 marks] For 10 marks apiece, implement any *two* of the following:

i. [10 marks] *Short-Circuit Evaluation*
To speed up the calculation of expression trees, use the following math trick. If the left subtree evaluates to 0 *and* the node is of type `asttok`, then your program doesn't need to evaluate the right subtree: the value is going to be 0 no matter what. There are three other operators in Python-ish which behave this way. Implement short-circuit evaluation for all 4 within `pish.cpp`.

ii. [10 marks] *Exception Handling*
In the provided version of Python-ish, two run-time errors are possible. Figure out what they are, and enhance your `pish.cpp` code so that it terminates gracefully in these cases.

iii. [10 marks] `break ;`
Add the statement `break ;` to Python-ish so that it unconditionally jumps out of its innermost loop. If `break ;` occurs outside of a loop, display a run-time error.

iv. [10 marks] `if/elif/else`
In this enhancement, Python-ish's `if` statement may be followed by 0 or more `elif` ("else if") clauses, and finally an optional `else` clause. Enhance your `pish.cpp` code so it follows the correct branch of the `if/elif/elif/.../else` structure.

Getting Started

We strongly recommend that you build your functionality in stages. It is likely we will test your code in the same stages.

- Start by getting simple print statments working. E.g., `print 42;`.
- Next, write code that evaluates expression trees. Start with `+-`, then add in `*/`, then add in the relative operators, and the logical operators. You will have to ignore evaluating variables/identifiers for the time being.
- Variables should come next. You will use your hash table for this.
- Next implement `if`, and finally implement `while`. They have a similar structure.