CS 280
Programming Language Concepts

About Assignment 4

Requirements for Assignment 4

- Implement an interpreter for our language
- Calculate the value for every expression
- Implement every statement
- Implement runtime error checks

From the assignment

Below is part of the list from assignment 4. Note that several of the items in the list are shown with [RT] next to them. These are items that must be checked at runtime.

- A Repeat statement evaluates the Expr. The Expr must evaluate to an integer **[RT]**. If the integer is nonzero, then the Stmt is executed, otherwise it is not. The value of the Expr is decremented by 1 each repetition. In other words “repeat 5 begin println “hello” end” will print “hello” 5 times.
- It is an error if a variable is used before a value is assigned to it **[RT]**.
- Performing an operation with incorrect types or type combinations is an error **[RT]**.
- Multiplying a string by a negative integer is an error **[RT]**.
- A Repeat statement whose Expr is not integer typed is an error **[RT]**.

NJIT New Jersey's Science & Technology University COLLEGE OF COMPUTING SCIENCES

Representing values: val.h

```
#ifndef VALUE_H
#define VALUE_H

#include <iostream>
#include <string>
using namespace std;

enum ValType { VINT, VSTR, VERR };

class Value {
   ValType     T;
               int                 I;
   string      S;

public:
   Value() : T(VERR), I(0) {}
   Value(int vi) : T(VINT), I(vi) {}
   Value(string vs) : T(VSTR), I(0), S(vs) {}

   ValType GetType() const { return T; }
   bool IsErr() const { return T == VERR; }
   bool IsInt() const { return T == VINT; }
   bool IsStr() const { return T == VSTR; }

   int GetInt() const { if( IsInt() ) return I; throw "RUNTIME ERROR: Value
not an integer"; }
   string GetStr() const { if( IsStr() ) return S; throw "RUNTIME ERROR:
Value not a string"; }

   // add op to this
   Value operator+(const Value& op) const;

   // subtract op from this
   Value operator-(const Value& op) const;

   // multiply this by op
   Value operator*(const Value& op) const;

   // divide this by op
   Value operator/(const Value& op) const;

   friend ostream& operator<<(ostream& out, const Value& op) {
      if( op.IsInt() ) out << op.I;
      else if( op.IsStr() ) out << op.S;
      else out << "ERROR";
      return out;
   }
};

#endif
```


Value

- A Value is a container for either an integer, a string, or an error (a “ValType”)
- There are getters to determine what the Value is and what its type is
- Exceptions can be thrown if types mismatch (i.e. you cannot get the integer value of something that is not an integer) OR an error value can be returned; up to you. Eventually someone will have to throw an exception…

Value

- There are comments in places where you need to implement code.
- Observe there is type checking and an exception thrown on type mismatch error:

```
Value operator+(const Val& op) const {
    if( isInt() && op.isInt() ) // integer addition
           return GetInt() + op.GetInt();
    if( isStr() && op.isStr() ) // string addition
           return GetStr() + op.GetStr();
    throw "RUNTIME ERROR …"
}
```


Implementation of Interpreter

- Every class in the parse tree needs an evaluation function. Call it Eval.
- A symbol table is needed to map an identifier to a Value. This is best done with a map declared in main:

  ```
  map<string,Value> symbols;
  ```

- The argument to Eval should be a reference to this map.

Implementation

- Pass a reference to the symbol table map to the Eval method so that every class has access to the symbol table. The signature of the method in the base class looks like

```
Value Eval(map<string,Val>& syms)=0;
```

- The "= 0" makes the function a "pure virtual" method and forces every class to provide an implementation.

Implementation

- The Value returned by an Eval() for most of the statements is unimportant, because statements have no Value that is carried along. (exception is assignment, which has a Value)
- For example, Eval() for StmtList can be

```
Value Eval(map<string,Val>& symbols) {
        left->Eval(symbols);
        if( right )
            right->Eval(symbols);
        return Value();
}
```

Implementation

- The Eval for the Primary items are quite simple. For example, IConst is

```
Value Eval(map<string,Val>& symbols) {
    return Value(val);
}
```

- Note it simply makes the integer constant into a Value.
- The Eval for identifier is a symbol table lookup, and the Eval for assignment makes a symbol table entry.

Implementation

- Eval for the operators can use the overloaded operator functions for the Value class, but you must handle errors.
- Might not be a bad idea to write a function to create properly formatted RUNTIME ERROR strings

NJIT New Jersey's Science & Technology University COLLEGE OF COMPUTING SCIENCES

Eval Function for other nodes

- Statement List: Evaluate the left child, and if there is a right child, evaluate the right child
- Identifier: Look up the identifier in the symbol table, return the Value from the symbol table; runtime error if not found
- Print: Eval the expression, and then print it
- Assignment: Eval the expression, then put the Value into the symbol table

Eval Function for operators

- Addition, Subtraction, Multiplication, etc. all must implement the operations as defined in the assignment
- Format of the Eval functions for these operations is: Eval the operands, perform the operation, return the result
- All operations check for various errors and generate runtime errors if checks fail

Runtime Checks

- There are specific checks for particular operations that must be implemented at runtime on a case-by-case basis
- Examples:
  - when multiplying an integer by a string, the integer must be nonnegative

Exceptions

- In this assignment we use exceptions for our runtime errors
- It is sometimes more convenient to be able to "throw" an error out of the middle of a computation
- This eliminates the need to "check for errors" returned from a function call, and to propagate the error up the chain of function calls
- Best use of this is for errors that really cannot be recovered from

Exception implementation

- Usually implemented as a “try/catch” block
  - ”try” to run some code in the try block
  - An error condition may cause the code to “throw” an exception
  - In the catch block, you “catch” a particular exception if it happens while the code runs
- The place that the exception is thrown may be several levels down in function calls
- Throwing an exception might therefore involve returning from multiple functions at once!

Implementing exceptions

- Performing an orderly unwinding of function calls is a critical part of the language’s exception handling
- A clean implementation of “throwing” an exception involves:
  - Cleaning up variables created inside the function
  - Generating a return from the function (but, note, an exception, not a value)
  - If the “catch” is in the function you just returned to, then execute the catch block
  - If not… repeat this list!

Exceptions

- In Java, a new object, derived from class Exception, is created and thrown
- In C++, anything can be thrown and caught
  - It's considered good practice to catch a reference to an object
  - C++ defines some standard exceptions in the std library
  - The standard exceptions have a what() method to access a string message that is thrown with the exception

C++ example

- Example throw

```
throw std::string("Type mismatch for arguments of +");
```

- Example catch

```
try {
    prog->Eval();
}
catch( std::string& e ) {
    cout << "RUNTIME ERROR " << e << endl;
}
```

NJIT New Jersey's Science & Technology University COLLEGE OF COMPUTING SCIENCES


NJIT
THE EDGE IN KNOWLEDGE