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.

- An IfStmt evaluates the Expr. The Expr must evaluate to an integer. **[RT]** If the integer is nonzero, then the Slist is executed.
- A LoopStmt evaluates the Expr. The Expr must evaluate to an integer **[RT]**. If the integer is nonzero, then the Slist is executed. This process repeats until the Expr evaluation does not result in a nonzero integer.
- Addition is defined between two integers (the result being the sum) or two strings (the result being the concatenation) **[RT]**.
- Subtraction is defined between two integers (the result being the difference) **[RT]**.
- Multiplication is defined between two integers (the result being the product) or for an integer and a string (the result being the string repeated integer times) **[RT]**.
- Division is defined between two integers **[RT]**. Division by zero is an error **[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]**.
- An IF or LOOP statement whose Expr is not integer typed is an error **[RT]**.

Representing values: val.h

```
#ifndef VAL_H
#define VAL_H

#include <string>
using namespace std;

class Val {
    int         i;
    string      s;
    enum ValType { ISINT, ISSTR, ISERR } vt;

public:
    Val() : i(0), vt(ISERR) {}
    Val(int i) : i(i), vt(ISINT) {}
    Val(string s) : i(0), s(s), vt(ISSTR) {}
    Val(ValType vt, string errmsg) : i(0), s(errmsg), vt(ISERR) {}

    ValType getVt() const { return vt; }

    bool isErr() const { return vt == ISERR; }
    bool isInt() const { return vt == ISINT; }
    bool isStr() const { return vt == ISSTR; }

    int ValInt() const {
        if( isInt() ) return i;
        throw "This Val is not an Int";
    }
    string ValString() const {
        if( isStr() ) return s;
        throw "This Val is not a Str";
    }

    friend ostream& operator<<(ostream& out, const Val& v) {
        // IMPLEMENT
    }

    string GetErrMsg() const {
        if( isErr() ) return s;
        throw "This Val is not an Error";
    }

    Val operator+(const Val& op) const {
        if( isInt() && op.isInt() )
            return ValInt() + op.ValInt();
        if( isStr() && op.isStr() )
            return ValString() + op.ValString();
        return Val(ISERR, "Type mismatch on operands of +");
    }

    Val operator-(const Val& op) const {
        // IMPLEMENT
    }

    Val operator*(const Val& op) const {
        // IMPLEMENT
    }

    Val operator/(const Val& op) const {
        // IMPLEMENT
    }

    Val operator!() const {
         // IMPLEMENT
    }

};

#endif
```

Val

- A Val is a container for either an integer, a string, or an error (a “ValType”)
- There are methods to determine what the Val is
- One item of note is that if you construct a Val by passing a ValType and a string, the Val is marked as an Error type, and the string is the error message
  - This is a mechanism for passing errors up to the caller

Val

- There are comments in places where you need to implement code. Note operator+ is provided. Observe there is type checking and an error return on a type mismatch error in the implementation

```
Val operator+(const Val& op) const {
    if( isInt() && op.isInt() ) // integer addition
            return ValInt() + op.ValInt();
    if( isStr() && op.isStr() ) // string addition
            return ValString() + op.ValString();
    return Val(ISERR, "Type mismatch on operands of +");
}
```

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 Val. This is best done with a map declared in main:

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

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

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

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

Implementation

- The Val returned by an Eval() for one of the statements is unimportant, because statements have no Val that is carried along.
- For example, Eval() for StmtList can be

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

Implementation

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

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

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

Implementation

- Eval for the operators can use the overloaded operator functions in Val, but you must handle errors. For example, PlusExpr could be:

```
Val Eval(map<string,Val>& symbols) {
        auto L = left->Eval(symbols);
        if( L.isErr() )
            runtime_err(linenum, L.GetErrMsg());
        auto R = right->Eval(symbols);
        if( R.isErr() )
            runtime_err(linenum, R.GetErrMsg());

        auto answer = L + R;
        if( answer.isErr() )
            runtime_err(linenum, answer.GetErrMsg());

        return answer;
    }
```

- NOTE that runtime_err is a function I wrote that assembles a properly formatted string and throws it as an exception.

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
- Let: Eval the expression, then put the Value into the symbol table
- Etc.

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:
  - in an IF statement, the expression must be integer typed
  - when multiplying an integer by a string, the integer must be nonnegative

Adding Checking

- Everything can have a type and a value

- Some checks are type checks
- Some checks are value checks
- Failing either kind of check is a RUNTIME ERROR

- The type and the value might be known statically, or it might need to be determined dynamically

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

THE EDGE IN KNOWLEDGE