City University London

IN2009

Language Processors

Session 8

The SPL Interpreter

Igor Siveroni

Session Plan

Session 7:

- The Source Programming Language (SPL)
- Interpreter implementation
- Test2: Sample questions

Test2: Monday, March 29. 1-2pm
Topic: Grammars, SPL and TPL.

Source Programming Language (SPL)

SPL's syntax:

Program$_S$ → *MainDecl*
MainDecl → `void main(){` *VarDecl*$^+$ *Statement*$^+$ `}`
VarDecl → `int` *id* `;`
Statement → *AssignStm* | *PrintStm* | *IfStm* | *WhileStm*
AssignStm → *id* `:=` *exp* `;`
PrintStm → `print(` *exp* `) ;`

SPL: Syntax

IfStm → `if (` *exp* `)` `then` *Block* [`else` *Block*]
WhileStmr → `while(` *exp* `)` *Block*
Block → `{` *Statement*$^+$ `}`
exp → *exp AOp exp* | *exp BOp exp* | *exp COp exp* |
id | *integer* | `!(`*exp*`)` | `true` | `false`
Aop → + | - | * | / *Bop* → `and` | `or`
Cop → > | >= | ==

Interpreter: The files

- sourcepl.jj: JavaCC file.
- Main.java: Driver class.
- **ast** directory: A Java class per abstract syntax tree element (node).
  - Program.java, MainFunction.java, Stm.java, AssignStm.java, etc.
- **visitor** directory:
  - Visitor.java
  - Interpreter.java and PrinterVisitor.java

Abstract Syntax Implementation

A class per element of abstract:

Program(MainDecl m)
MainDecl(List<id> vars, Statement+ statements)
AssignStm(Id v,Exp e)
PrintStm(Exp e)
IfStm(Exp e, Statement+ ls1, Statement+ ls2)
While(Exp e, Statement+ body)
OpExp(Exp e1, Aop op, Exp e2)
BoolExp(Exp e1, Bop op, Exp e2)
CmpExp(Exp e1, Cop op, Exp e2)
IdExp(Id x) IntLiteralExp(int n) BoolLiteralExp(bool b)

AST: Program and MainDecl

Abstract syntax: Program(MainDecl m)

```
public class Program { // File: Program.java
  public MainFunction mainf;
  public Program(MainFunction m) { mainf=m;}
  public int accept(Visitor v) throws VisitorExcep
    { return v.visit(this); }   }
```

Abstract syntax: MainDecl(List<id> vars, Statement+ ls)

```
public class MainFunction {//File: MainFunction.java
  public List<Stm> statements;
  public List<String> vars;
  public MainFunction(List<String> v,List<Stm> s)
      { statements=s; vars=v; }
  public int accept(Visitor v)  throws VisitorExcep
      { return v.visit(this); }      }
```


JavaCC: Program and MainDecl

```
Program program() :
{ MainFunction m; }
{
  m = mainFunction() <EOF>
  { return new Program(m); }
}


MainFunction mainFunction() :
{ List<Stm> sl; List<String> varDecs; }
{
  "void" "main" "(" ")" "{"
   varDecs=varDeclList()  sl=stmList()  "}"
   { return new MainFunction(varDecs,sl); }
}
```


AST: Assign/If Statement

Abstract Syntax: AssignStm(Id v,Exp e)

```
public class AssignStm extends Stm { //Assign.java
  public String id;
  public Exp exp;
  public AssignStm(String i, Exp e) {id=i; exp=e;}
  public int accept(Visitor v) throws VisitorExcep
    { return v.visit(this); }    }
```

Abstract Syntax: IfStm(Exp e, Stm+ ls1, Stm+ ls2)

```
public class IfStm extends Stm {
  public Exp exp;
  public List<Stm> ls1,ls2;
  public IfStm(Exp e, List<Stm> l1, List<Stm> l2)
      { exp=e; ls1=l1; ls2=l2; }
  public int accept(Visitor v) throws VisitorException
      { return v.visit(this); }    }
```


JavaCC: Statements

```
Stm stm() :
{ Token t; Exp e=null; Stm s; }
{
   t=<ID> ":=" e=exp() ";"
              { return new AssignStm(t.image,e); }
 | <PRINT> "(" e=exp() ")" ";"
              { return new PrintStm(e); }
 | s = ifstm()    { return s; }
 | s = whilestm() { return s; }
}

Stm ifstm() :
{Exp e=null; List<Stm> ls1=null,ls2=null; Stm s; }
{
 <IF> "(" e=exp() ")" <THEN> ls1 = block()
 [ <ELSE> ls2 = block() ]
 { return new IfStm(e,ls1,ls2);  }
}
```


AST: Binary Arithmetical Exp

Abstract syntax: OpExp(Exp e1, Aop op, Exp e2)

File: OpExp.java

```
public class OpExp extends Exp {
   public enum AOp { PLUS,MINUS,TIMES,DIV };
   public Exp left, right;
   public AOp oper;
   public OpExp(Exp l, AOp o, Exp r)
      { left=l; oper=o; right=r;}
   public int accept(Visitor v) throws
                                    VisitorException
      { return v.visit(this); }
}
```


JavaCC: Binary Arithmetical Exp

```
Exp exp() :
{ Exp e; }
{ e=sum() { return e; }  }

Exp sum() :
{ Exp e,e2=null; }
{
e=term() (
   (<PLUS> e2=term()
       { e = new OpExp(e,OpExp.AOp.PLUS,e2); })
 | (<MINUS> e2=term()
       { e = new OpExp(e,OpExp.AOp.MINUS,e2); })
 | (<AND> e2=term()
       { e = new BoolExp(e,BoolExp.BOp.AND,e2); })
 | (<OR> e2=term()
       { e = new BoolExp(e,BoolExp.BOp.OR,e2); })
)*
  { return e; }
}
```

The Interpreter: A visitor

File: visitor/Visitor.java

```
public interface Visitor {
 public int visit(Program n) throws VisitorException;
 public int visit(MainFunction n) throws VisitorException;
 public int visit(PrintStm n) throws VisitorException;
 public int visit(AssignStm n) throws VisitorException;
 public int visit(IfStm n) throws VisitorException;
 public int visit(WhileStm n) throws VisitorException;
 public int visit(IdExp n) throws VisitorException;
 public int visit(OpExp n) throws VisitorException;
 public int visit(CmpExp n) throws VisitorException;
 public int visit(BoolExp n) throws VisitorException;
 public int visit(IntLiteralExp n) throwsVisitorException;
 public int visit(BoolLiteralExp n) throws VisitorException;
 public int visit(NotExp n) throws VisitorException;
}
```


The Interpreter: A visitor

```
public class Interpreter implements Visitor {
    enum Type { INT, BOOL, NOTYPE };
    Type valType;
    Table table = new Table();
    PrinterVisitor printer = new PrinterVisitor();

    public int visit(Program p) throws VisitorException {
       return p.mainf.accept(this);
    }

// A visit method per AST class
```

- **valType**: Type of last evaluated (traversed/visited) expression. Used to verify types during the execution of the program (dynamic typecheck).
- **table**: Object of class Table used to store the values associated to each declared varible of the program (lookup table).
- The **return** value is only used when expressions are evaluated: statements do not return values, they chanage the state of the program.

The Interpreter: MainFunction

```
import java.util.*;
public int visit(MainFunction m)  throws VisitorException {
       Stm s;

       /* initialise declared vars with 0 */
       Iterator<String> itvars = m.vars.iterator();
       while (itvars.hasNext()){
           table.update(itvars.next(),0);
       }

       /* Execute statements */
       Iterator<Stm> iterator = m.statements.iterator();
       while (iterator.hasNext()){
           s = iterator.next();
           s.accept(this);
       }
       valType=Type.NOTYPE;
        return 0;
}
```


Assign Statement

```
public int visit(AssignStm s) throws VisitorException {
      int v = s.exp.accept(this);
      checkAssignType(s,valType,Type.INT);
      table.update(s.id,v);
      valType=Type.NOTYPE;
      return 0;
 }
```

where

```
class Table {
  /* implemented as a list of TableNodes */
  Table() { … }
  void update(String i, int v) { … }
  int lookup(String id) throws TableLookupException { … }
}
```


If Statement

```
public int visit(IfStm s) throws VisitorException {
      int v = s.exp.accept(this);
      checkExpType(s.exp,s.exp,valType,Type.BOOL);
      Iterator<Stm> iterator;
      if (v!=0) {
          iterator = s.ls1.iterator();
          while (iterator.hasNext()){
             iterator.next().accept(this);
          }
      } else if (s.ls2 != null) {
          iterator = s.ls2.iterator();
          while (iterator.hasNext()){
             iterator.next().accept(this);
          }
      }
      valType=Type.NOTYPE;
      return 0;
 }
```


OpExp Expression

```
public int visit(OpExp e) throws VisitorException {
      int r=0;
      int v1 = e.left.accept(this);
      checkExpType(e,e.left,valType,Type.INT);
      int v2 = e.right.accept(this);
      checkExpType(e,e.right,valType,Type.INT);
      switch(e.oper) {
      case PLUS: r = v1+v2; break;
      case MINUS: r = v1-v2; break;
      case TIMES: r = v1*v2; break;
      case DIV: if (v2!=0) r = v1/v2; break;
      default:
          System.out.print("operator unknown");
      }
      valType = Type.INT;
      return r;
   }
```

More expressions…

```
public int visit(IdExp e) throws VisitorException {
   try {
     valType=Type.INT;
     return table.lookup(e.id);
  } catch (TableLookupException ex) {
  throw new VisitorException("Variable "+e.id+" undefined");
  }
}

public int visit(IntLiteralExp e) throws VisitorException {
  valType = Type.INT;
  return e.num;
}

public int visit(BoolLiteralExp e) throws VisitorException {
  valType = Type.BOOL;
  if (e.value) return 1; else return 0;
}  // implement booleans as integers 0/1
```


Test 2: Questions

- Given the grammar:
  (1) E → E "+" E (2) E → E "-" E (3) E → E "*" E
  (4) E → E "/" E (5) E → "(" E ")" (6) E → num

And string "6 / 3 + (2)"

- Show a leftmost derivation that generates the string. Label each arrow with the appropriate rule number.
- Show a rightmost derivation.
- When is a grammar ambiguous?
- Is the grammar above ambiguous? Show using your previous answer.

Test 2: Questions

- Given the grammar:
  L → L ";" L
  L → S

- Write down 3 strings defined by the grammar above.
- Is the grammar left-recursive? If it is, write down an equivalent grammar that is not left-recursive. Compare with the rule given in the lecture notes.
- Write down an equivalent grammar using EBNF.

Test 2: Questions

- TPL: Suppose memory locations 2 and 4 are used to store variables x and y, with values 50 and 100. What does the following code do? What's the output?

```
STORE $2, R1
STORE $4, $2
STORE R1, $4
WRITEI $2
WRITEI $4
```

- Write code that prints the contents of x n times, where n is the value of y. Do not modify y.

Test 2: Questions

- Suppose we want to add a short version of a For statement to TPL using the following syntax:
  ForStm → **(** *id* **=** *Exp* **;** *Exp***;** *id* **=** *Exp* **)** *Body*

- Write down the **abstract syntax** for *ForStm*.
- Write the JavaCC code that parses and creates the AST for the For statement.
- Translate ForStm to TPL.

- More questions in CitySpace

Dates

- Test 2: Monday, March 29, 1-2pm.
- Programming Assignment due: Thursday, April 1, 8pm.

- Test 3 & Revision: Monday, April 26.
  Room and time TBA this week.
  No Programming assignment.