CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Thanks!
- There will be five assignments - total 40 marks.
- Midterm = 20 marks, Final = 40 marks.
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a *good* question, Answer a *chosen* question, Make a good point! Take 0.5 marks each. Max one marks per day per person.
  - A bonus assignment - you can get 10 marks.
  - Maximum additional marks on the table = 15. So even if you don't do well in a couple of assignments, you can make it up!
- *Plagiarism* - A good word to know. A bad act to own.

Contact (Anytime) :
Email: nvk@cse.iitm.ac.in, Skype (nvkrishna77), Office: BSB 352.

What, When and Why of POPL

- **What**:
  - A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer.
  - Fundamental principles in the design, definition, analysis, and implementation of programming languages and programming systems.
- **When**
  - Early 19^{th} century - programmable looms.
  - 1930s - Mathematical abstractions (such as lambda calculus and Turing machines).
- **Why Study**
  - It is good to know the food you eat.
    - Helps appreciate the languages.
    - Helps learn techniques to analyze programs.
  - Experts in POPL are in great demand (both in academia and in industry).

Mutual expectations

For the class to be a mutually learning experience:

- What will be required from the students?
  - An open mind to learn.
  - Curiosity to know the basics.
  - Explore their own thought process.
  - Help each other to learn and appreciate the concepts.
  - Honesty and hard work.
  - Leave the fear of marks/grades.
- What are the students expectations?
  - Apply studied techniques.
  - Help appreciate new languages.
  - May help in future.

Outline

Course outline

- Introduction: tools.
- Program semantics: semantics and equivalence.
- Type systems: type soundness, lambda calculus
- Type inference: inference algorithms
- Continuation passing style and closure conversion.
- Exceptions
- Partial evaluation.

Outline

The Java Compiler Compiler (JavaCC)

- Can be thought of as “Lex and Yacc for Java.”
- It is based on LL(k) rather than LALR(1).
- Grammars are written in EBNF.
- The Java Compiler Compiler transforms an EBNF grammar into an LL(k) parser.
- TheJavaCC grammar can have embedded action code writtenin Java, just like a Yacc grammar can have embedded action code written in C.
- The lookahead can be changed by writing LOOKAHEAD(. . .).
- The whole input is given in just one file (not two).

JavaCC input

One file

- header
- token specification for lexical analysis
- grammar

Example of a token specification:

```
TOKEN : {
  < INTEGER_LITERAL: ( ["1"-"9"] (["0"-"9"])* | "0" ) >
}
```

Example of a production:

```
void StatementListReturn() :
{}
{
  ( Statement() )* "return" Expression() ";"
}
```

Generating a parser with JavaCC

```
javacc fortran.jj   // generates a parser with a specified name
javac Main.java     // Main.java contains a call of the parser
java Main < prog.f  // parses the program prog.f
```

Outline

Why have lexer at all? Why not do everything using the parser?

The Visitor Pattern

- The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.
- Implication: the ability to add new *operations* to existing object structures *without* modifying those structures.
- Interesting in object oriented programming and software engineering.

Requirements

- The set of classes must be fixed in advance, and
- each class must have an `accept` method.

Motivate Visitor by summing an integer list

```
interface List {}

class Nil implements List {}

class Cons implements List {
  int head;
  List tail;
}
```

1/3 approach: instanceof and type casts

```
List l;       // The List-object
int sum = 0;
boolean proceed = true;
while (proceed) {
  if (l instanceof Nil)
     proceed = false;
  else if (l instanceof Cons) {
     sum = sum + ((Cons) l).head;
     l = ((Cons) l).tail;
     // Notice the two type casts!
  }
}
```

Adv: The code is written without touching the classes `Nil` and `Cons`.
Drawback: The code constantly uses explicit type cast and `instanceof` operations.

2/3 approach: dedicated methods

- The first approach is NOT object-oriented!
- Classical method to access parts of an object: dedicated methods which both access and act on the subobjects.

```
interface List {
  int sum();
}
```

- We can now compute the sum of all components of a given List-object `ll` by writing `ll.sum()`.

2/3 approach: dedicated methods (contd)

```
class Nil implements List {
  public int sum() {
    return 0;
  }
}
class Cons implements List {
  int head;
  List tail;
  public int sum() {
    return head + tail.sum();
  }
}
```

- **Adv:** The type casts and instanceof operations have disappeared, and the code can be written in a systematic way.
- **Drawback**: For each new operation, new dedicated methods have to be written, and all classes must be recompiled.

3/3 approach: Visitor pattern

The Idea:

- Divide the code into an object structure and a Visitor.
- Insert an accept method in each class. Each accept method takes a Visitor as argument.
- A Visitor contains a `visit` method for each class (overloading!) A `visit` method for a class C takes an argument of type C.

```
interface List {
  void accept(Visitor v);
}
interface Visitor {
  void visit(Nil x);
  void visit(Cons x);
}
```

3/3 approach: Visitor pattern

- The purpose of the accept methods is to invoke the visit method in the Visitor which can handle the current object.

```
class Nil implements List {
  public void accept(Visitor v) {
    v.visit(this);
  }
}
class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

3/3 approach: Visitor pattern

- The control flow goes back and forth between the visit methods in the Visitor and the accept methods in the object structure.

```
class SumVisitor implements Visitor {
  int sum = 0;
  public void visit(Nil x) {}
  public void visit(Cons x) {
    sum = sum + x.head;
    x.tail.accept(this);
  }
}
.....
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

The visit methods describe both
1) actions, and 2) access of subobjects.

3/3 approach: Visitor pattern control flow

```
interface List {
  void accept(Visitor v); }
interface Visitor {
  void visit(Nil x);
  void visit(Cons x); }
```

```
class Nil implements List {
  public void accept(Visitor v) {
    v.visit(this); } }
class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) {
    v.visit(this); } }
```

```
class SumVisitor implements Visitor {
  int sum = 0;
  public void visit(Nil x) {}
  public void visit(Cons x) {
    sum = sum + x.head;
    x.tail.accept(this); } }
.....
SumVisitor sv = new SumVisitor();
l.accept(sv);
```

Comparison

#	detail	Frequent type casts	Frequent recompilation
1.	Instanceof + type-cast	Yes	No
2.	Dedicated methods	No	Yes
3.	Visitor pattern	No	No

- The Visitor pattern combines the advantages of the two other approaches.
- **Advantage** of Visitors: New methods without recompilation!
- **Requirement** for using Visitors: All classes must have an accept method.

Tools that use the Visitor pattern:

- JJTree (from Sun Microsystems), the Java Tree Builder (from Purdue University), both frontends for The JavaCC from Sun Microsystems.
- ANTLR generates default visitors for its parse trees.

Visitors: Summary

- Visitor makes adding new operations easy. Simply write a new visitor.
- A visitor gathers related operations. It also separates unrelated ones.
- Adding new classes to the object structure is hard. Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most like to change the classes of objects that make up the structure.
- Visitors can accumulate state.
- Visitor can break encapsulation. Visitor's approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.

(Useless?) Assignment 1

- Write the three versions of code corresponding to each of the above discussed approaches.
- Populate the lists with 'N' number of elements.
- Print the Sum of elements.
- Convince yourself about the programmability with Visitor pattern.
- See which of the three approaches is more efficient?
- Vary 'N' - 10; 100; 1000; 100,0000; 10,00,000.
- Make a table and report the numbers.
- Write a paragraph or two reasoning about the performance.
- Mention any thoughts on performance improvement.

The best "useless" answer(s) will be recognized.

Outline

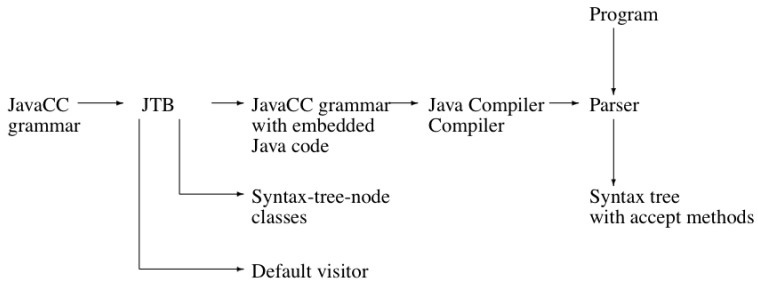
Java Tree builder

- The Java Tree Builder (JTB) has been developed here at Purdue (my ex group).
- JTB is a frontend for The Java Compiler Compiler.
- JTB supports the building of syntax trees which can be traversed using visitors. Q: Why is it interesting?
- JTB transforms a bare JavaCC grammar into three components:
  - a JavaCC grammar with embedded Java code for building a syntax tree;
  - one class for every form of syntax tree node; and
  - a default visitor which can do a depth-first traversal of a syntax tree.

The Java Tree Builder

The produced JavaCC grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees.

The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.



Invoking JTB

```
jtb fortran.jj     // generates jtb.out.jj
javacc jtb.out.jj  // generates a parser with a specified name
javac Main.java    // Main.java contains a call of the parser
                      and calls to visitors
java Main < prog.f // builds a syntax tree for prog.f, and
                      executes the visitors
```

(simplified) Example

For example, consider the Java production

```
void Assignment() : {}
   {PrimaryExpression() AssignmentOperator() Expression()}
```

JTB produces:

```
Assignment Assignment () :
{ PrimaryExpression n0;
  AssignmentOperator n1;
  Expression n2; {} }
{ n0=PrimaryExpression()
  n1=AssignmentOperator()
  n2=Expression()
  { return new Assignment(n0,n1,n2); }
}
```

Notice that the production returns a syntax tree represented as an Assignment object.

(simplified) Example

JTB produces a syntax-tree-node class for Assignment:

```
public class Assignment implements Node {
  PrimaryExpression f0; AssignmentOperator f1;
  Expression f2;
  public Assignment(PrimaryExpression n0,
                    AssignmentOperator n1,
                    Expression n2)
  { f0 = n0; f1 = n1; f2 = n2; }
  public void accept(visitor.Visitor v) {
      v.visit(this);
} }
```

Notice the `accept` method; it invokes the method `visit` for `Assignment` in the default visitor.

(simplified) Example

The default visitor looks like this:

```
public class DepthFirstVisitor implements Visitor {
  ...
   //
   // f0 -> PrimaryExpression()
   // f1 -> AssignmentOperator()
   // f2 -> Expression()
   //
   public void visit(Assignment n) {
      n.f0.accept(this);
      n.f1.accept(this);
      n.f2.accept(this);
} }
```

Notice the body of the method which visits each of the three subtrees of the `Assignment` node.

(simplified) Example (multiple visitors in action)

Here is an example of a program which operates on syntax trees for Java programs. The program prints the right-hand side of every assignment. The entire program is six lines:

```
public class VprintAssignRHS extends DepthFirstVisitor {
   void visit(Assignment n) {
      VPrettyPrinter v = new VPrettyPrinter();
      n.f2.accept(v); v.out.println();
      n.f2.accept(this);
} }
```

When this visitor is passed to the root of the syntax tree, the depth-first traversal will begin, and when `Assignment` nodes are reached, the method `visit` in `VprintAssignRHS` is executed.

`VPrettyPrinter` is a visitor that *pretty prints* Java programs.

JTB is bootstrapped.

Outline

Scheme Language

An interpreted language.
A sample session: (the shell evaluates expressions)

```
$ mzscheme
Welcome to Racket v5.2.
> 3
3
> (+ 1 3)
4
> '(a b c)
(a b c)
> (define x 3)
> x 3
> (+ x 1)
4
```

```
> (define l '(a b c))
> l
(a b c)
> (define u '(+ x 1))
> u
(+ x 1)
> (define u (+ x  1))
> x
3
> u
4
>
```

Procedures

Creating procedures with lambda: `(lambda (x) body)`

```
> (lambda (x) (+ x 1))
#<procedure>
> ((lambda (x) (+ x 1)) 4)
5
> (define mysucc (lambda (x) (+ x 1)))
> (mysucc 4)
5
> (define myplus (lambda (x y) (+ x y)))
> (myplus 3 4)
7
> ((lambda (x y) (+ x y)) 3 4)
7
```

Procedures can take other procedures as arguments:

```
> ((lambda (f x) (f x 3)) myplus 5)
8
```

Q: How are C pointers different than a lambda?

Procedures (contd)

Procedures can return other procedures; this is called Currying:

```
> (define twice
  (lambda (f)
      (lambda (x)
            (f (f x)))))

> (define add2 (twice (lambda (x) (+ x 1))))

> (add2 5)

> 7
```

Kinds of data

- **Basic values** = Symbols ∪ Numbers ∪ Strings ∪ Lists
- **Symbols**: sequence of letters and digits starting with a letter. The sequence can also include other symbols, such as -,$,=,*,/,?, . Numbers: integers, etc.
- **Strings**: ”this is a string”
- **Lists**:
  1. the empty list is a list ()
  2. a sequence $(s_1, \cdots s_n)$ where each s_i is a value (either a symbol, number, string, or list)
  3. nothing is a list unless it can be shown to be a list by rules (1) and (2).

  This is an inductive definition, which will play an important part in our rea- soning. We will often solve problems (e.g., write procedures on lists) by following this inductive definition.

List Processing

Basic operations on lists:

- `car`: if `l` is $(s_1 \cdots s_n)$, then `(car l)` is s_1.
  - The car of the empty list is undefined.

```
> (define l '(a b c))
> (car l)
> a
```

- `cdr`: if `l` is $(s_1\ s_2 \cdots\ s_n)$, then `(cdr l)` is $(s_2\ \cdots\ s_n)$.
  - The cdr of the empty list is undefined.

```
> (cdr l)
> (b c)
```

Combining car and cdr:

```
> (car (cdr l))
> b
> (cdr (cdr l))
> (c)
```

Building lists

- `cons`: if `v` is the value `s`, and `l` is the list $(s_1 \cdots s_n)$, then `(cons s l)` is the list $(v\ s_1\ \cdots\ s_n)$.
- `cons` builds a list whose `car` is `s` and whose `cdr` is `l`.

```
(car (cons s l)) = v
(cdr (cons s l)) = l
```

```
cons : value * list -> list
car  : list -> value
cdr  : list -> list
```

Genesis of the names

- Lisp was originally implemented on the IBM 704 computer, in the late 1950s.
- The 704 hardware had special support for splitting a 36-bit machine word into four parts:
  1. an ”address part” of fifteen bits,
  2. a ”decrement part” of fifteen bits,
  3. a ”prefix part” of three bits,
  4. a ”tag part” of three bits.
- Precursors to Lisp included functions:
  1. car = ”Contents of the Address part of Register number”,
  2. cdr = ”Contents of the Decrement part of Register number”,
  3. cpr = ”Contents of the Prefix part of Register number”,
  4. ctr = ”Contents of the Tag part of Register number”.
- The alternate `first` and `last` are sometimes more preferred. But `car` and `cdr` have some advantages: short and compositions.
  - cadr = car cdr, caadr = car car cdr, cddr = cdr cdr
- `cons` = constructs memory objects.

Boolean related

Literals:

```
#t, #f
```

Predicates:

```
(number? s)                          (number?  3)
(symbol? s)                          (symbol? 'a)
(string? s)                          (string? "Hello")
(null? s)                            (null? '())
(pair? s)                            (pair? '(a . b))
(eq? s1 s2)     -- works on symbols  (eq? '(a b) '(a b))
                                     (eq? 'a 'a)
                                     (eq? "a" "a")
(equal? s1 s2)  -- recursive         (equal? "a" "a")
(= n1 n2)       -- works on numbers  (= 2 2)
(zero? n)                            (zero? x)
(> n1 n2)                            (> 3 2)
```

Conditional:

```
(if bool e1 e2)
```

Recursive Procedures

- Say we want to write the power function: $e(n, x) = x^n$.
- $e(n, x) = x \times e(n-1, x)$
- At each stage, we used the fact that we have the problem solved for smaller n — Induction.

Recursive procedures

```
(define e
  (lambda (n x)
    (if (zero? n)
        1
        (* x
           (e (- n 1) x)))))
```

Why does this work? Let's prove it works for any n, by induction on n:

1. It surely works for n=0.
2. Now assume (for the moment) that it works when `n = k`. Then it works when `n=k+1`. Why? Because `(e n x) = (* x (e k x))`, and we know `e` works when its first argument is `k`. So it gives the right answer when its first argument is `k + 1`.

Structural Induction

- **The Moral**: If we can reduce the problem to a smaller subproblem, then we can call the procedure itself ("recursively") to solve the smaller subproblem.
- Then, as we call the procedure, we ask it to work on smaller and smaller subproblems, so eventually we will ask it about something that it can solve directly (eg `n=0`, the basis step), and then it will terminate successfully.
- Principle of structural induction: If you always recur on smaller problems, then your procedure is sure to work.

Recursive procedures

```
(define fact
  (lambda (n)
      (if (zero? n) 1 (* n (fact (- n 1))))))

(fact 4) = (* 4 (fact 3))
         = (* 4 (* 3 (fact 2)))
         = (* 4 (* 3 (* 2 (fact 1))))
         = (* 4 (* 3 (* 2 (* 1 (fact 0)))))
         = (* 4 (* 3 (* 2 (* 1 1))))
         = (* 4 (* 3 (* 2 1)))
         = (* 4 (* 3 2))
         = (* 4 6)
         = 24
```

- Each call of `fact` is made with a promise that the value returned will be multiplied by the value of `n` at the time of the call; and
- thus `fact` is invoked in larger and larger control contexts as the calculation proceeds.

Loops

Java

```
int fact(int n) {
  int a=1;
  while (n!=0) {
    a=n*a;
    n=n-1;
  }
  return a;
}
```

Scheme

```
(define fact-iter
   (lambda (n)
      (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
   (if (zero? n)
     a
     (fact-iter-acc (- n 1)
                    (* n a)))))
```

Q: Is it not a recursive function?

Trace - loop

```
(fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

- `fact-iter-acc` is always invoked in the same context (in this case, no context at all).
- When `fact-iter-acc calls` itself, it does so at the ”tail end” of a call to fact- iter-acc. That is, no promise is made to do anything with the returned value other than return it as the result of the call to `fact-iter-acc`.
- Thus each step in the derivation above has the form `(fact-iter-acc n a)`.

Many way switch - cond

```
(cond
  (test1 exp1)
  (test2 exp2)
  ...
  (else exp_n))
```

Let

When we need local names, we use the special form let:

```
(let ((var1 val1)
      (var2 val2)
      ...)
     exp)
```

```
(let ((x 3)
      (y (+ x 4)))
(* x y))
```

```
(let ((x 5))
     (let ((f (+ x 3))
           (x 4))
          (+ x f)))
```

Limitations of let

Scheme

```
(define fact-iter
   (lambda (n)
      (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a
      (fact-iter-acc (- n 1) (* n a)))))
```

Can we write a local recursive procedure?

```
(define fact-iter
  (lambda (n)
    (let ((fact-iter-acc
            (lambda (n a)
              (if (zero? n)
                  a
                  (fact-iter-acc (- n 1) (* n a))))))
      (fact-iter-acc n 1))))
```

Local recursive procedures

The scope of fact-iter-acc doesn't include it's definition. Instead, we can use `letrec`:

```
(letrec
 ((name1 proc1)
  (name2 proc2)
  ...)
  body)
```

`letrec` creates a set of mutually recursive procedures and makes their names available in the body. So we can write:

```
(define fact-iter
  (lambda (n)
    (letrec ((fact-iter-acc
                (lambda (n a)
                  (if (zero? n) a
                    (fact-iter-acc (- n 1)
                                   (* n a))
              ))))
              (fact-iter-acc n 1))))
```

Practise problems (with and without using letrec)

- Find the 'n' the element in a given list. (Input: a list and n. Output: error or the n'th element)
- symbol-only? – checks if a given list contains only symbols. List → boolean
- member?: (List, element) → boolean
- remove-first: List → List
- replace-first: (List, elem) → List
- remove-first-occurrence: (List, elem) → List
- remove-all-occurrences: (List, elem) → List

Outline

- Update on useless assignment 1?
- Update on practise problems (last class)?
- Assignment 1?
- Health card update for Lecture 1: JavaCC, Visitor Pattern, JTB

Revise the scope of let and letrec

```
(let (var1 exp1) (var2 exp2) S)

var1 is visible inside S.
var2 is visible inside S.

(letrec (var1 exp1) (var2 exp2) S)

var1 is visible in exp1, exp2, and S.
var2 is visible in exp1, exp2, and S.
```

One requirement: no reference be made to `var1` and `var2` during the evaluation of `exp1`, and `exp2`.
This requirement is easily met if `exp1` and/or `exp2` are `lambda` expressions - reference to the variables `var1` and `var2` are evaluated only only when the resulting procedure is invoked.

Examples with let and letrec

```
(letrec ((x (+ x 1))) x) -- undefined.
(letrec ((x y) (y 1)) x) -- undefined
(letrec ((x (lambda () (+ y 1))) (y 3)) (x)) -- 4
(let ((x 3)) (let ((y (+ x 4))) (* x y)))
```

$\neq$

```
(let ((x 3) (y (+ x 4))) (* x y))
```

Argument sequencing

Arguments are evaluated before procedure bodies.

- In `((lambda (x y z) body) exp1 exp2 exp3)`

  exp1, exp2, and exp3 are guaranteed to be evaluated before body, but we don't know in what order exp1, exp2, and exp3 are going to be evaluated, but they will all be evaluated before body.
- This is precisely the same as
  `(let ((x exp1) (y exp2) (z exp3)) body)`

  In both cases, we evaluate exp1, exp2, and exp3, and then we evaluate body in an environment in which x, y, and z are bound to the values of exp1, exp2, and exp3.

Things to Do

- Make sure you have a working account.
- Start brushing on Java and Scheme.
- Review Java development tools.
- Check out the course webpage: `http://www.cse.iitm.ac.in/~krishna/cs6848/`, (for the assignment 1 - due on?).

Health card

A Basic Scheme syntax and lists
B Recursive procedure and induction
C Let and Letrec

4: Can teach myself, 3: Can teach with help, 2: Need a bit of help, 1: No clue.