7-2003

Toward a Comprehensive Supplement for Language Courses

Krishnaprasad Thirunarayan
Wright State University - Main Campus, t.k.prasad@wright.edu

Stephen P. Carl
Wright State University - Main Campus

Toward a Comprehensive Supplement for Language Courses

Krishnaprasad Thirunarayan & Stephen P. Carl

Department of Computer Science & Engineering
Wright State University
Dayton, OH-45435

WRIGHT STATE UNIVERSITY

Outline

- Relevance of techniques and tools for Programming Language (PL) Design and Implementation to Information Technology
  - Simplified illustrative examples of related concepts and features
- Symptoms showing student lack of understanding of PL basics
- Concrete ways to improve assimilation of PL concepts and constructs with benefits to IT education via analogies

Information Systems : Representation and Processing

- Syntax
  - "Standard" Language of Expression and Interchange
    - Code reuse (DOM)
    - Ease of parsing
      - E.g., XML-DTD, which defines permissible data and attribute fields, based on context-free grammars (actually, Deterministic Extended Backus Naur Formalism)
        - XML documents can be self-describing (using DTD schema) and hence, can be validated.

(cont'd)

- XML/DTD is just a context free grammar.
- A DOM is just a parse tree.
- An XML parser does the same job as LEX/YACC except that its interpreted.
  - ☹ *P. Windley*: Syntactic sugar makes the syntax of a language pretty. XML is *syntactic arsenic*.
  - ☹ *P. Wadler*: XML is just a notation for trees, a verbose variant of LISP S-expressions.

DTD Example

```
<?xml version="1.0"?>
<!DOCTYPE BOOK [
   <!ELEMENT p                                (#PCDATA)>
   <!ELEMENT BOOK
                    (OPENER,INTRODUCTION?,(SECTION | PART)+)>
   <!ELEMENT OPENER                           (TITLE_TEXT)*>
   <!ELEMENT TITLE_TEXT                       (#PCDATA)>
   <!ELEMENT INTRODUCTION                     (HEADER, p+)+>
   <!ELEMENT PART                             (HEADER, CHAPTER+)>
   <!ELEMENT SECTION                          (HEADER, p+)>
   <!ELEMENT HEADER                           (#PCDATA)>
   <!ELEMENT CHAPTER                          (CHAPTER_NUMBER, CHAPTER_TEXT)>
   <!ELEMENT CHAPTER_NUMBER                   (#PCDATA)>
   <!ELEMENT CHAPTER_TEXT                     (p)+>
]>
```

```xml
<BOOK>
  <OPENER>
    <TITLE_TEXT>All About Me</TITLE_TEXT>
  </OPENER>
  <PART>
    <HEADER>Welcome To My Book</HEADER>
    <CHAPTER>
      <CHAPTER_NUMBER>CHAPTER 1</CHAPTER_NUMBER>
      <CHAPTER_TEXT>
        <p>Glad you want to hear about me.</p>
        <p>There's so much to say!</p>
        <p>Where should we start?</p>
        <p>How about more about me?</p>
      </CHAPTER_TEXT>
    </CHAPTER>
  </PART>
</BOOK>
```

Semantics

- Machine processable tags embedded into human sensible documents (*Semantic Web*)
  - Interoperability issues due to lack of consensus on the semantics of XML-tags
    - Different concepts, same tags -> Context-sensitivity.
    - Same concept, dissimilar tags -> Equivalence issue.
  - Transformations using recursive tree traversals
  - Content extractors/semantic taggers based on compiler-frontend tools

Reasoning

- Declarative specification and Querying
  - Logic and functional languages; Ontologies
  - Source to source transformations in Search Engines, Semantic Taggers, etc
- Defining Interface and Behaviors
  - Separation of concerns
    - E.g., Web services context
- Reliability and Security
  - Type systems

Other Issues

- Portability across Platforms
  - Standardization through specification
    - E.g., Language reference manuals
- Robust Architecture
  - Smooth assimilation of changes, over time
    - E.g., Object-Oriented Paradigm
- Rapid Prototyping
  - Improving programmer productivity
    - E.g., Scripting languages

Educational Gaps w.r.t. PL and IT

In spite of good resources, students

- use technical jargon (and acronyms) without understanding them.
- describe a concept in abstract terms but cannot recognize or apply it concretely.
- use hackneyed examples when requested to provide illustrations.

Retrospective on Causes

- Examples are from correlated sources
- Different languages embody the same concepts using different syntax, and similar looking syntax in different languages can have subtly different semantics.
  - E.g., Java and C++ syntax
- Inadequate mathematical training and maturity

- Phil Windley summarizes the related IT education problem thus:
  - Most of the computing literature on XML, SOAP, and Web Services fails to relate these technologies back to CS theory and PL that any computer scientist should know.
  - The writings on these technologies is full of hype, making them seem more complicated than they are.
  - Most programmers are not familiar with RPC or messaging to any great extent and so their generalizations are even more obtuse.

A Proposal for Effective Teaching of Language Features and Techniques

- Develop an example-rich supplement that gets across language fundamentals and a comparative study of modern languages
  - E.g., Focus on similarities, differences, subtleties, and trade-offs among related features
  - E.g., Illustrate IT issues lucidly in a simpler setting
- Provide progressively difficult but well-integrated exercises, to apply and gauge, the grasp and appreciation of the material

Supplementary Topics and Materials for Comparative Languages

- Programming Styles
  - Imperative style
    - L-values *vs* R-values
      - E.g., assignment *vs* l-value yielding function
  - Imperative *vs* Functional
    - Iteration *vs* Recursion
      - E.g., I < R :: expressive power argument
      - E.g., I /\ R :: tail-recursion and space-time trade-off

TAIL RECURSION:

```
int fibtr(int n, int prev1, int prev2) {
    if (n == 0)                 return prev1;
    else if (n == 1)            return prev2;
    else return fibtr(n-1, prev2, prev1 + prev2);
}
```

Exercises : Specification and Implementation

- E.g., Parsing arithmetic expressions, type checking/inference, and generating bytecodes
- E.g.,
  - *HW*: Calculator for constant arithmetic expressions
  - *PL*: Augmenting expressions with variables, programmable calculator, etc
- E.g.,
  - *Spec*: Algebraic specification of *Polynomials*
  - *HW*: Implementing polynomials
  - *PL*: Augmenting polynomial calculator with memory, programmable polynomial calculator, etc

Procedural *vs* OOP Architecture

- Continuity : evolution under updates
  - In OOP Style, smooth assimilation of new implementation of an abstract behavior (E.g., data format changes)
  - In Procedural Style, smooth assimilation of new functionality
- Interfaces : Client-Server View
  - In Procedural Style, a client is responsible for invoking appropriate server action.
  - In OOP Style, a server is responsible for conforming to the standard interface, assumed by the client.

```
(define (size C)
  (cond ((vector? C)  (vector-length C))
    ((pair? C)     (length C))
    ((string? C)  (string-length C))
    ( else                    ... )) ))

(size '(one ``two'' 3))
```

```java
interface iCollects { int size(); }

class cVector extends Vector implements iCollects {}
class cString extends String implements iCollects {
   public int size() { return length(); }
}
class cArray implements iCollects {
   int[] array;
   public int size() { return array.length; }
}

iCollects c = new cVector(); c.size();
```

- Declarative *vs* Procedural
  - "Interpreter supplies control strategy for using the same declarative specification to answer different queries"

```
append([], L, L).
append([ H | T ], L, [ H | R ]) :-
                        append(T, L, R).
```

- "." and ":-" are logical connectives that stand for "*and*" and "*if*" respectively.
- "[]" and "|" stand for *empty list* and *cons* operation.

- Concatenation
  - sig: list x list -> list
    - append([1], [2,3], R).

- Verification
  - sig: list x list x list
    - append([1], [2,3], [1,2,3]).
- Constraint solving
  - sig: list x list -> list
    - append(R, [2,3], [1,2,3]).
  - sig: list -> list x list
    - append(A, B, [1,2,3]).
- Generation
  - sig: -> list x list x list
    - append(X, Y, Z).

Exercises

- Paradigm Comparison
  - Develop attribute grammar for static semantics of expressions
  - Modify to obtain an executable specification in a logic language (e.g., Prolog Definite Clause Grammars)
  - Convert into an object-oriented language (e.g., Java), a functional (e.g., Scheme) and an imperative language (e.g., C).
  - Explore relationship between magic sets in databases and attribute grammars

Programming Language Design

- Portability ("Importance of Language Definition")

```
#include <stdio.h>
main() {
 int i = 5;
 printf("\t i = %d, i/++i = %d, i = %d\n\n",
           i,     i/++i,     i);
}
/* Compilers: cc, gcc
    SUN3 :            i = 6,  i/++i  = 1, i = 5
  SPARC20:            i = 6,  i/++i  = 1, i = 6
   ALPHA :            i = 5,  i/++i  = 1, i = 6
    MIPS :            i = 5,  i/++i  = 1, i = 6
INTUITION:            i = 5,  i/++i  = 0, i = 6  */
```

Object-Oriented Languages

- Class
  - Static description; unit of modularity; type
- Object
  - Runtime structure
- Inheritance and Polymorphism
  - Code reuse
- Polymorphism and Dynamic Binding
  - Representation independence; Information hiding
  - Interaction with type system
- Inheritance *vs* Delegation

```
class P {
   public void f(P p)
   { System.out.println("f(P) in P. "); }
}

class C extends P {
   public void f(P p)
   { System.out.println("f(P) in C. "); }
   public void f(C cp)
   { System.out.println("f(C) in C. "); }
}
```

```
class DynamicBinding {
   public static void main(String[] args) {
      P px  = new P();   C cx  = new C();
      P py  = cx;
      px.f(px);             //f(P) in P.
      px.f(cx);             //f(P) in P.
      py.f(px);             //f(P) in C.
      py.f(cx);             //f(P) in C.
      cx.f(px);             //f(P) in C.
      cx.f(cx);             //f(C) in C.
   }
}
```

Conclusion

- Discussed an example-rich approach to comparative languages
- Based on our experience, this approach is effective
- We believe that IT educators can benefit both from content and pedagogy proposed here.