City University London

IN2009

Language Processors

Week 1

Introduction to Language Processors & Recap of Java

From code to action

Christian Cooper

Who am I?

- *(Remember me from IN1007?)*
- Christian Cooper
- Visiting Lecturer
- Situated in A503
- **chi@soi.city.ac.uk**

- Teaching **IN2009 Language Procesors** this semester.

Ground Rules

Please:

- Arrive on time (all lectures start on the hour, so come in and sit down a few minutes early).
- Switch off 'phones, alarms etc.
- Ask questions.

Please **DON'T**:

- Chat in the background.
- Text message/play phone games/etc.

Audio/Video recording of classes

- My policy:
- You are free to use a Dictaphone or similar to record the audio of my lectures/labs…
  - …so long as you give me the courtesy of asking me first; I rarely refuse permission.
- Video recording of lectures and labs is generally ***not*** permitted (without good reason).

Dates for your diary...

- Lectures (10):
  - Weeks 1-5, 7-11.
  - No lecture weeks 6 and 12.
- Labs (10):
  - ***Start this week!***
  - If you didn't know that, you've missed them!
  - Weeks 1-5, 7-11.
  - No labs weeks 6 and week 12.

What's in store this week

Week 1

- Module details, aims, resources
- What is language processing and implementation?
- Syntax definition
- Straight-line programming language
- Abstract syntax trees
- Some Java reminders...

Module Details

- **Module Leader - Prof. David Bolton**
  - But please contact *me* in the first instance for any day-to-day queries…
- **Support**
  - The primary form of support is during the lab sessions.
  - Teaching Assistant: ***Kamal Pal***

Module Details

- **Support** (con't)
  - Online support is through Cityspace: **`http://www.city.ac.uk/cityspace`**
  - Use the discussion boards well!
  - e-mail should be seen as a last resort; unless it is discussing purely personal issues, it will simply be anonymised and cut & paste into the discussion boards on Cityspace!
  - …with up to a week's delay!

Module Details

- **Rationale**
- "Most computer programs process input whose structure is expressible in a language definition. Such input includes programming languages themselves, of course."

Module Details

- "An understanding of how to write such definitions, and how to turn a definition into a recogniser and translator for the language, will give students an understanding of programming language structure and implementation that will compliment programming skills and aid the learning of new programming languages."

Module Details

- "Also, an understanding of the run-time environment for the translated program will give insight into how high-level programs execute at machine level."

Module Aims

- "To introduce students to the **specification** and **implementation** of programming languages. In particular the module aims to provide an introduction to the **structure** of programming languages, the **algorithms and data structures** used in compilers, the tools which may be used to **automate** compiler construction, and to the **run time environments** in which programs execute."

Indicative Content

- Syntax definition using regular and context free grammars
- Abstract and concrete syntax, and abstract syntax tree representations
- Introduction to type-checking
- Translation
- Runtime environments

Core Text

- Appel, A., *"Modern Compiler Implementation in Java"*, 2nd ed., 2002, Cambridge University Press, ISBN-13: 978-0521820608

Appel (2002)

Learning Outcomes

- On successful completion of this module, you will be able to:
  - Use formal languages to define input language syntax.
  - Explain techniques for syntactic and semantic analysis and translation.
  - Explain the compiled code, and run-time environment requirements, for various common programming language structures.
  - Program data structures and algorithms for representation and analysis and translation of programming languages.
  - Use standard compiler generation tools.

Assessment

- In-module coursework tasks
  - 2 practical exercises (from weeks 4 - 11).
  - 30% of module marks (12%, 18%).
  - Pair working may be allowed in the first task, but you **must** declare if you do so! More on this at a later date.
- End-of-module exam
  - Unseen, written 1.5-hour paper.
  - 70% of module marks (*min threshold 30%*).
  - In May 2007.

Coursework

- The first assessment brief will be published on Cityspace within the next fortnight.
- Deadlines:
  - All deadlines are by ***5pm*** on ***Friday*** of the indicated week.
  - Assessment 1: Week 7
  - Assessment 2: Week 12

Coursework

- *(From the student handbook)*
- **4.4.7. Lateness Penalties**
- If you hand in a piece of coursework after 5pm on the deadline date it will be marked as late.
- Lecturers will apply a late penalty of up to 10 marks per day. Therefore assignments received 10 days or more pass the deadline will receive a mark of 0.

A note on Plagiarism

- Plagiarism is the failure to acknowledge any work which is not 100% your own.
- This includes undeclared “group working” on assessments.
  - You are permitted to work as a pair in the first coursework, but you must declare this (and tell me who your partner was for each submitted exercise). Groups of three and more is not allowed.
- Plagiarism leads to a formal hearing!

Coursework

- Coursework marks and feedback will be returned to you via Cityspace.
  - *Usually* within two weeks of the deadline.
- It is my intention to produce a guideline answer for all assessments - published 3-4 weeks after the deadline.
  - Not a “model” solution, but one which is fit for purpose and would have achieved approximately 50-60%

Structure of the lectures

- This week serves as an introduction to the topic, and will also focus on a brief recap of Java.
  - This is the only time I will focus on Java and programming from a “learning a language” perspective rather than “compiling a language” - I expect you to revise Java in your own time this week, and to practice if you do not feel 100% confident with the language!

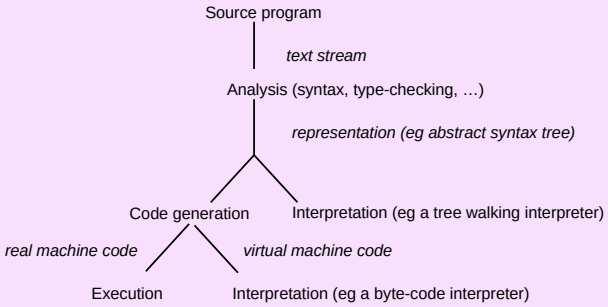
Structure of the lectures

- Including this week, there are 9 lectures focussing on aspects of language processing, organised into 8 logical learning blocks
  - “Sessions”
  - Some sessions may span multiple lectures.
- The final week’s lecture will focus on reviewing the module, considering revision and working on past papers.

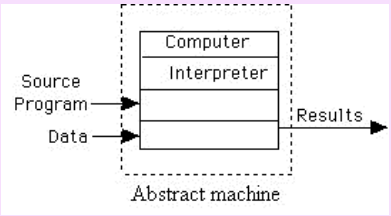
Structure of the lectures

1. ***Introduction to Language Processing*** (this week)
2. ***Language processing & lexical analysis*** (next week!)
3. ***Parsing I (syntax analysis)***
4. ***Parsing II (abstract syntax)***
5. ***MiniJava abstract syntax trees***
6. ***Semantic analysis***
7. ***Activation records (stack frames)***
8. ***Translation to intermediate representation***

Language processing & implementation

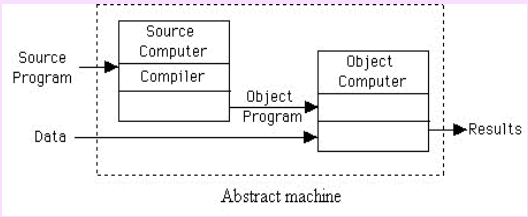


Pure interpreter



- Programming language, plus computer and interpreter, provides an abstract (or virtual) machine for the programmer.

Pure compiler

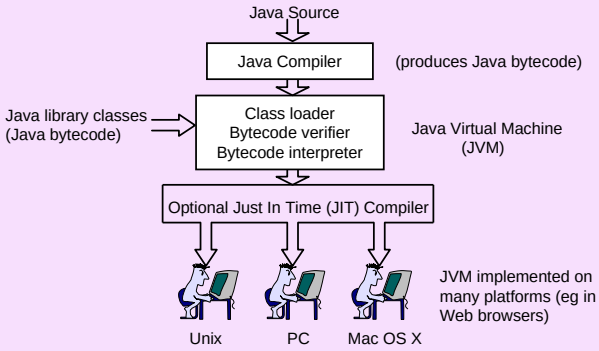


- Programming language plus compiler and computer provides an abstract machine.

Pure compiler

- But only rarely do we have a pure interpreter or compiler.
- Typically code is first compiled to *intermediate* form.
- Then…
  - interpreted, or
  - code generated for a virtual machine interpreter or for a real machine

Example: Java implementation



What we will do in this module…

- Work through Appel (2002)
  - Leaving out much of the theory
- Define a simple programming language
- Implement some some example abstract trees for the simple programming language
- Introduce a more complicated language
- Implement a lexical analyser using JavaCC

What we will do in this module…

- Implement a syntax analyser using JavaCC
- Implement an abstract syntax tree builder
- Look at semantic analysis (eg type-checking)
- Look at runtime environments and translation

Syntax definition

- Use context-free grammars (e.g. Backus–Naur Form or ***BNF***) to define a grammar for the language

```
Stm     →    id := Exp  |  ifStm |   …
ifStm   →    if Exp then Stm  |  if Exp then Stm else Stm
Exp     →    id  |  num  |  Exp Binop Exp  |  …
```

|	"or" - separates alternatives
→	"is defined as"
`IfStm, Stm`	non-terminals
`if, id, num, :=`	terminals (tokens)

Syntax definition

- Each definition is called a production; many productions define a grammar
- Repetition by recursive definition
- You may have seen BNF before…
  - …any MySQL users? The manual and instruction list shows the statement syntax in BNF.

Syntax definition

```
<syntax>      ::=   <rule> | <rule> <syntax>
<rule>        ::=   <opt-whitespace> "<" <rule-name> ">"
                    <opt-whitespace> "::="
                    <opt-whitespace> <expression>
                    <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<expression>  ::=   <list> | <list> "|" <expression>
<line-end>    ::=   <opt-whitespace> <EOL>
                    | <line-end> <line-end>
<list>        ::=   <term>
                    | <term> <opt-whitespace> <list>
<term>        ::=   <literal> | "<" <rule-name> ">"
<literal>     ::=   '"' <text> '"' | "'" <text> "'"
```

- (From Wikipedia - guess what this BNF is for?)

Straight-line programming language

Stm	→	Stm ; Stm	*(CompoundStm)*
Stm	→	`id` := Exp	*(AssignStm)*
Stm	→	`print` (ExpList)	*(PrintStm)*
Exp	→	`id`	*(IdExp)*
Exp	→	`num`	*(NumExp)*
Exp	→	Exp Binop Exp	*(OpExp)*
Exp	→	(Stm , Exp)	*(EseqExp)*
ExpList	→	Exp , ExpList	*(PairExpList)*
ExpList	→	Exp	*(LastexpList)*
Binop	→	+	*(Plus)*
Binop	→	-	*(Minus)*
Binop	→	×	*(Times)*
Binop	→	/	*(Div)*

Straight-line programming language

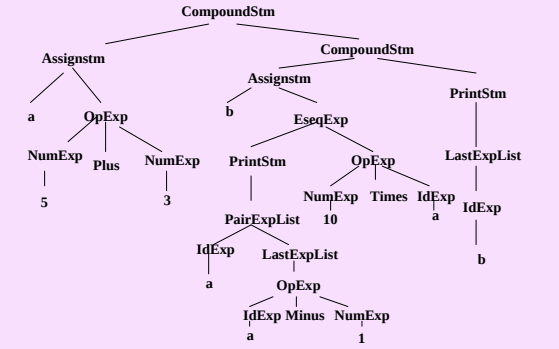
Stm → Stm ; Stm | `id` := Exp | `print` (ExpList)
Exp → `id` | `num` | Exp Binop Exp | (Stm , Exp)
ExpList → Exp , ExpList | Exp
Binop → + | - | × | /

- A program (what does it do?):

```
a := 5+3;
b := (print(a, a-1), 10*a);
print (b)
```



```
a := 5+3;
b := (print(a, a-1), 10*a);
print (b)
```



Java representation of the abstract syntax

```
static Stm prog =
new CompoundStm(new AssignStm("a",new OpExp(new NumExp(5),
                      OpExp.Plus, new NumExp(3))),
 new CompoundStm(new AssignStm("b",
   new EseqExp(new PrintStm(new PairExpList(
                new IdExp("a"),
                new LastExpList(new OpExp(new IdExp("a"),
                        OpExp.Minus, new NumExp(1))))),
             new OpExp(new NumExp(10), OpExp.Times,
                      new IdExp("a")))),
    new PrintStm(new LastExpList(new IdExp("b")))));
```


Java abstract syntax representation: Program 1.5

```
abstract class Stm {}

class CompoundStm extends Stm {
   Stm stm1, stm2;
   CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}
}

class AssignStm extends Stm {
   String id; Exp exp;
   AssignStm(String i, Exp e) {id=i; exp=e;}
}

class PrintStm extends Stm {
   ExpList exps;
   PrintStm(ExpList e) {exps=e;}
}
```


Java abstract syntax representation: Program 1.5

```
abstract class Exp {}

class IdExp extends Exp {
   String id;
   IdExp(String i) {id=i;}
}
class NumExp extends Exp {
   int num;
   NumExp(int n) {num=n;}
}

class OpExp extends Exp {
   Exp left, right; int oper;
   final static int Plus=1,Minus=2,Times=3,Div=4;
   OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}
}
```


Java abstract syntax representation: Program 1.5

```
class EseqExp extends Exp {
   Stm stm; Exp exp;
   EseqExp(Stm s, Exp e) {stm=s; exp=e;}
}

abstract class ExpList {}

class PairExpList extends ExpList {
   Exp head; ExpList tail;
   public PairExpList(Exp h, ExpList t) {head=h; tail=t;}
}

class LastExpList extends ExpList {
   Exp head;
 public LastExpList(Exp h) {head=h;}
}
```


`maxargs` and `interp`

- **`int maxargs(Stm s)`** returns the maximum number of arguments of any print statement within any subexpression of a given statement in a Straightline program.
  - **`maxargs(prog)`** returns 2
  - remember that print statements can contain expressions that contain other print statements
- **`void interp(Stm s)`** "interprets" a program written in the Straightline language

What you should do now…

- Read, digest and understand these slides!
  - in particular work out how you would write **`maxargs(Stm s)`** - I will give you **`interp(Stm s)`**, which is much harder…

```
class interp {
  static void interp(Stm s) { /* you write this part */ }

  static int maxargs(Stm s) { /* you write this part */
                              return 0;
  }

  public static void main(String args[])
                  throws java.io.IOException {
     System.out.println(maxargs(prog.prog));
     interp(prog.prog);
  }
}
```

Java recap - “datablast I”

- The basics:
  - objects, classes, primitive data types, method invocation, attributes, parameters, return values/types, fields, constructors, mutator vs. accessor, variables, assignment, object references, selection (`if` statements), basic iteration (`while`/`for` loops), class diagrams, string manipulation, basic arithmetic and comparison, `public` vs `private`, using a debugger, commenting, class documentation, using API libraries, arrays, collections, hashes, `iterator`, basic testing...

Java recap - “datablast” II

- Designing Classes & O-O concepts:
  - Cohesion, coupling, refactoring, subtyping, overriding, polymorphism, abstract classes
- GUI programming in `JSwing`, Inheritance, Java `interface`, Event-driven programming, layout managers, Errors, Exceptions, File handling, serialisation...

Java recap - doing without BlueJ

- It is also time for us to do away with the “training wheels” and develop without BlueJ holding our hands.
- This is easier than it might seem;
- Our main program needs an ***entry point*** - which is the method `main()`:
- `public static void main(String args[])`
- This method needs to set up and construct your initial objects, and then that is it!

Required reading for Week 2

- Before next week’s labs and lecture you should read the following:
  - Appel (2002), ch. 1-2.

Next Lecture

- *Language processing & lexical analysis*
- Monday 29th January, 2007
  - 12:00 - 13:50
  - CM383