City University London

IN2009

Language Processors

Week 1

Introduction to Language Processors

Igor Siveroni

IN2009

- Instructor: Igor Siveroni
  - Research Associate/Visiting Lecturer
  - Room A306, College Building
  - `siveroni@soi.city.ac.uk`
- Pre-Requisite: Java
- Lectures: 11.00-12.50 at C350
- Tutorial/Labs
  - 13.00-13.50 at A217
  - 15.00-15.50 at A218

Dates for your diary...

- Lectures (10):
  - Weeks 1-5, 7-11.
  - No lecture weeks 6 and 12.
- Labs (10):
  - ***Start this week!***
  - Weeks 1-5, 7-11.
  - No labs weeks 6 and week 12.

This Week

Week 1

- Module details, aims, resources
- What is language processing?
- Introduction to syntax definition
  - Example: A straight-line programming language
- Abstract syntax trees
- …and some Java.

Module Details

- **Support**
  - Online support is through Cityspace: http://www.city.ac.uk/cityspace
    - E.g. Digital copies of all lectures and labs.
  - Use discussion boards to post questions.
  - Questions (except personal issues) sent by email will be cut & pasted into the discussion boards on Cityspace.
  - …with up to a week's delay!

Core Text

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

Appel (2002)

Assessment

- In-module coursework tasks
  - 3 practical assessments (weeks 4 - 11).
  - 30% of module marks.
  - Pair working is allowed, 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/June 2009.

Coursework

- The first assessment brief will be published on Cityspace next week.
- Deadlines:
  - All deadlines are by ***5pm*** on ***Friday*** of the indicated week.
- Lateness Penalties
  - Assignments handed in after the deadline will receive a 0.
  - Special circumstances will be reviewed by a committee.

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 around 3 weeks after each deadline.
  - Not a "model" solution, but one which is fit for purpose and would have achieved approximately 50-60%

Module Details
Language Processors - Why?

- **Programming Languages is one of the cornerstones of Computer Science**
- Programming Languages design and implementation covers a wide range of theoretical and practical aspects of Computer Science.

Module Details - Why?

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

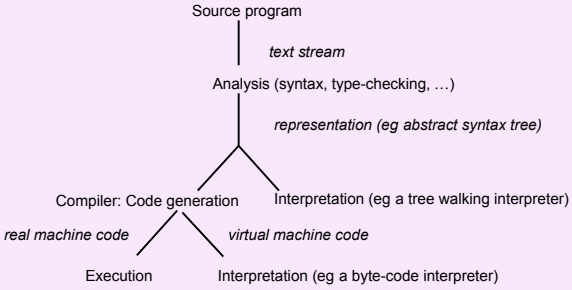
Module Aims

- "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."

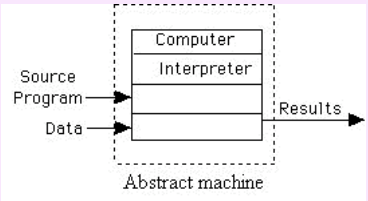
Indicative Content

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

PL Implementation: Interpreters & Compilers

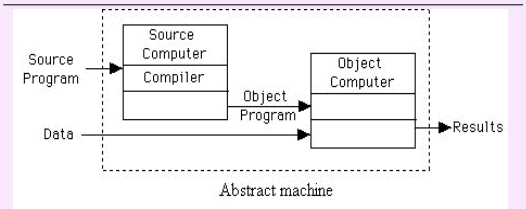


Interpreter



- A computer program that executes instructions written in a Programming Language.

Compiler

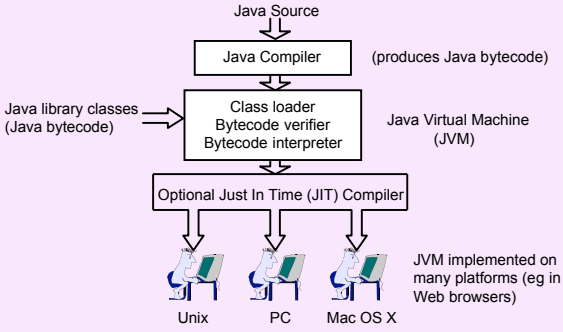


- A computer program that transforms:
  - code written in a programming language (source code) into another computer language (target language e.g. binary, bytecode, intermediate representation)

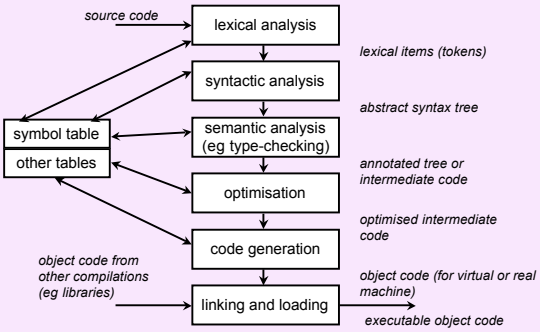
Compilers & Interpreters

- But only rarely do we have a pure interpreter or compiler.
- Typically code is first compiled to *intermediate* form. Then …
  - It is interpreted, or
  - code is generated for a virtual machine interpreter, a real machine or even another programming language.
- Interpreters may also perform internal transformations.

Example: Java implementation



Language processing: Compiler



Concepts and Techniques

Concepts and Techniques used:

- Language Specification
  - Syntax: Regular Expressions and Grammars
  - Semantics: Formal Methods
- Lexical and Syntactic Analysis: Finite Automata and Parsing. Parser generators.
- Semantics Analysis and Optimisation:
  - Type Checking, Intermediate representation.
  - Control and Data Flow Analysis (Program Analysis)
- Code Generation: Runtime environments, machine language, operating systems, garbage collection, etc.
- Programming!!!!

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.

Lectures

1. ***Introduction to Language Processing*** (this week)
2. ***Language processing & lexical analysis***
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***

What we will do in this module…

- Work through Appel (2002)
  - Leaving out much of the theory
- Define a simple programming language
- Implement 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 (e.g. type-checking)
- Look at runtime environments and translation

Intermission

- Since this is a 2 hour lecture, it is time for a short break…
  - Please be seated and ready to continue in **10** minutes time (that doesn't mean arrive back at the room in 10 minutes time!)

Lecture recommences at:

Syntax definition

- Suppose we want to implement a compiler for a language that looks as follows:

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

- How do we:
  - Specify the language's syntax?
  - Represent a program internally?

Syntax definition

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

```
Stm  →  id := Exp  |  Stm ; Stm | print(ExpList)
Exp  →  id  |  num  |  Exp Binop Exp  |  …
```

|	"or" - separates alternatives
→	"is defined as"
`Exp, Stm`	non-terminals
`print, 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.
  - Java specification.

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)*

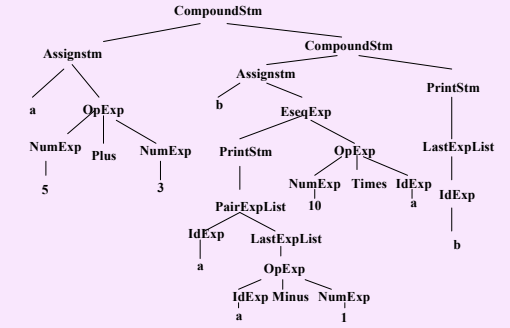
Straight-line programming language

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

- A program in this language:
  - What does it do?
  - How do we represent it inside the computer?
    Answer: We use trees.

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

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



Abstract Syntax Tree (AST) Java 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 representation AST

```
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 representation AST

```
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;}
}
```


Java representation AST Instantiation

```
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")))));
```


Programming the AST `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)`** - next slides
  `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 {
     Stm prog = Parser.parse(args[1]); // next lectures
     System.out.println(maxargs(prog));
     interp(prog);
  }
}
```


Method `maxargs(Stm s)`

```
static int maxargs(Stm s) {
  if (s instanceof PrintStm) {
    return Math.max (maxargs (((PrintStm) s).exps),
                     length (((PrintStm) s).exps));
  // Assuming length is defined for ExpList
  } else if (s instanceof AssignStm)
     return maxargs (((AssignStm) s).exp);
  } else if (s instanceof CompoundStm)
     return Math.max (maxargs (((CompoundStm) s).stm1),
                      maxargs (((CompoundStm) s).stm2));
  } else {
    System.out.println("maxargs(Stm): unrecognised Stm");
    return 0;
  }
}
```


method `maxargs(ExpList e)`

```
static int maxargs (ExpList e) {
   if (e instanceof PairExpList)
     return Math.max
       (maxargs (((PairExpList) e).head),
        maxargs (((PairExpList) e).tail));
   else if (e instanceof LastExpList)
     return maxargs (((LastExpList) e).head);
   else {
     System.out.println
       ("maxargs(ExpList): unrecognised ExpList");
     return 0;
   }
 }
```


method `maxargs(Exp e)`

```
static int maxargs (Exp e) {
   if (e instanceof OpExp) {
     return Math.max
       (maxargs (((OpExp) e).left),
        maxargs (((OpExp) e).right));
   } else if (e instanceof EseqExp) }
     return Math.max
       (maxargs (((EseqExp) e).stm),
        maxargs (((EseqExp) e).exp));
   } else {
     /* it's an IdExp or a NumExp */
     return 0;
 }
```


method `maxargs`

- But this solution is not very “object oriented”.
- What about adding the method **`maxargs()`** to the abstract classes **`Stm`** and **`Exp`**?
- Exercise: Try this approach.
- Exercise: Implement **`interp`**

Java - doing without BlueJ

- Lab and Coursework: We’ll develop Java programs without BlueJ
- 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 Appel (2002), ch. 1-2.
  - Review Java.

Next Lecture

- *Language processing & lexical analysis*
- Monday 2nd February, 2009
- 11:00 - 12:50
  - C.350