City University London

IN2009

Language Processors

Week 1

Introduction to Language Processors

Igor Siveroni

IN2009

- Instructor: Igor Siveroni
  - Research Associate/Visiting Lecturer
  - Room A309, 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 A217

Dates for your diary...

- Lectures (10):
  - Weeks 1-5, 7-11.
  - No lecture weeks 6 and 12.
- Labs (7) / Short Tests (3):
  - ***Start this week!***
  - Weeks 1-5, 7-11.
  - Short tests will take place during Lab hours. Dates TBA.
  - No labs/tests 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!

Readings

- Extra reading material and exercises supplied by instructor.
- Recommended: Appel, A., *"Modern Compiler Implementation in Java"*, 2nd ed., 2002, Cambridge University Press, ISBN-13: 978-0521820608

Assessment

- In-module coursework tasks
  - 3 practical assessments (weeks 4 - 11) made of in-class tests and take-home programming exercises.
  - 30% of module marks.
  - Programming: 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 2010.

Coursework

- The first assessment brief will be published on Cityspace by the end of next week.
- Deadlines:
  - All deadlines are by ***8pm*** 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 complement 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."

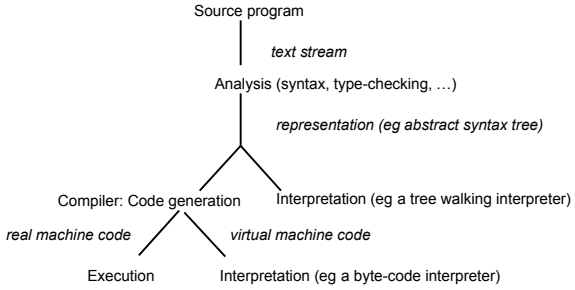
Module Aims

- "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."

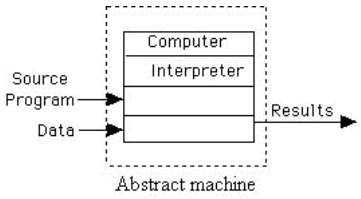
Indicative Content

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

PL Implementation: Interpreters & Compilers

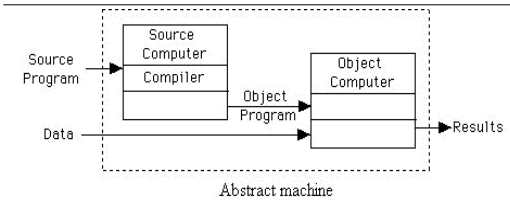


Interpreter



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

Compiler

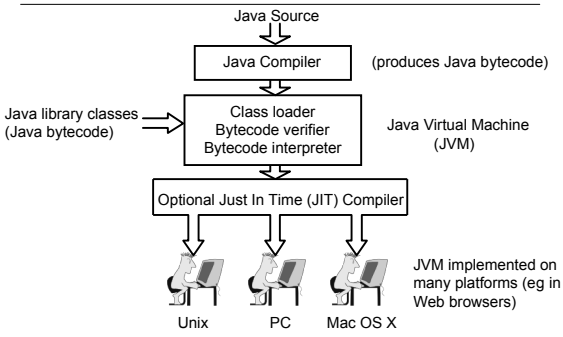


- A computer program that transforms:
  - code written in a programming language (source code) into another computer language (target language e.g. binary, bytecode, intermediate representation)

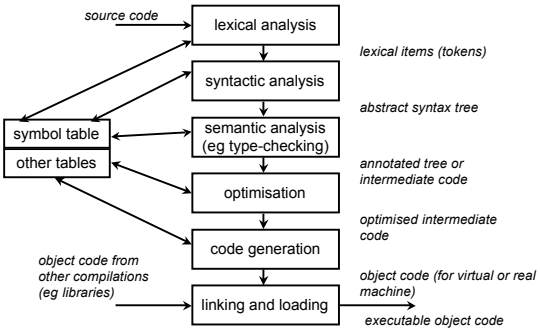
Compilers & Interpreters

- 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. Code Generation I (expressions and statements)
6. Semantic analysis (type checking)
7. Activation records (stack frames)
8. Code Generation II (procedures)

What we will do in this module…

- Learn regular expressions and grammars.
- Learn how to use JavaCC to generate lexical analysers and lexical analysers.
- Introduce an imperative language
- Generata a lexical and syntactic analyser for the imperative language using JavaCC.

What we will do in this module…

- Implement an abstract syntax tree builder.
- Introduce the target language and implement a code generator.
- Look at semantic analysis (e.g. type-checking)
- Extend the language with procedures and introduce stack frames.
- 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/C/C++ 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)*

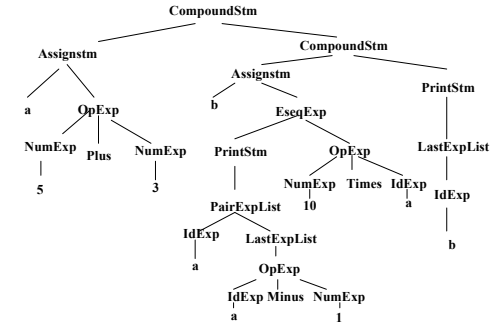
Straight-line programming language

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

```
a := 5+3;
print (b)
```

```
Stm prog =
  new CompoundStm(
    new AssignStm("a",
                  new OpExp(new NumExp(5),
                            OpExp.Plus,
                            new NumExp(3))),
    new PrintStm(new LastExpList(new IdExp("b")))));
```


Programming the AST `interp`

- **`void interp(Stm s)`** "interprets" a program written in the Straightline language.
- Add **`void/int interp(Table t)`** to each class of the AST.

```
class NumExp { …
  int interp(Table t) { return num; }  //returns value
}

class IdExp { …
  int interp(Table t) { return t.lookup(id); }
}

class AssignStm { …
  int interp(Table t) { t.update(id,exp.interp(t)); }
  // side-effect, updates value of id
}
```

What you should do now…

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

```
public static void main(String args[])
                throws java.io.IOException {
    Stm prog = Parser.parse(args[1]); // next lectures
    interp(prog);  // check LAB1 code
  }
}

class interp {
  static void interp(Stm s) {
    s.interp(new Table()); return;
  }
}
```


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!

Recommended reading for Week 2

- Before next week's labs and lecture you should:
  - Review Java
  - Read Appel (2002), ch. 1-2.

Next Lecture

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