Course Aims & Objectives

- To introduce students to the concepts underlying the design and implementation of language processors. More specifically, by the end of the course, students will be able to answer these questions:
  - What language processors are, and what functionality do they provide to their users?
  - What core mechanisms are used for providing such functionality?
  - How are these mechanisms implemented?
Course Aims & Objectives (2)

- Apart from providing a theoretical background, the course places a special emphasis in practical issues in designing language processors. Through labs and the assessed project, students will learn to:
  - Design and implement language processors in C/C++
  - Use tools to automate parts of the implementation process.
Information about the course

- Lecturer: Dr Yiannis Demiris, room EEE-1003
- Email: y.demiris@ic.ac.uk
- Course URL: http://www.iis.ee.ic.ac.uk/yiannis/lp/
- Contact hours:
  - 20 lectures, Mondays/Fridays
  - 9 tutorials, Mondays
  - 10 laboratory sessions, Fridays
Course overview (1)

What is included in the course

- Introduction to language processors: their underlying concepts and design
- Representative algorithms for the implementation of some of the mechanisms, in particular lexical and syntax analysis, context handling, and code generation and optimisation.
Course overview (2)
What is NOT included

- Installing (or removing) particular compilers in your system.
Course syllabus (1)

- **Introduction**
  - Basic concepts in language processors
    - The need for language processors
    - Different types; compilers and interpreters
    - Stages, phases and passes
  - Examples and evolution of language processors

- **Lexical Analysis**
  - Constructing tokens from characters; regular expressions, transition diagrams, finite automata (Non-deterministic (NFA) and Deterministic (DFA))

- **Syntax Analysis**
  - Constructing a syntax tree from the tokens
Course syllabus (2)

- Context Analysis
  - Checking and annotating the abstract syntax tree
- Code Generation:
  - Constructing target code from the annotated syntax tree
  - Code Optimization
  - Run Time Issues
- Advanced Topics [if time permits]
  - Natural Language Processing
- Case Study:
  - GCC
Recommended Reading List

- “Modern Compiler Design”, Grune, Bal, Jacobs and Langendoen, 2001
  [http://www.scifac.ru.ac.za/compilers](http://www.scifac.ru.ac.za/compilers)
Further Internet resources

- Freely-available compiler books – a list at www.compilers.net/Dir/Free/Books/index.htm

  Relevant ones to this course:
  - Compilers and Compiler Generators: an introduction using C++
  - Parsing Techniques: A practical guide
  - A Compact Guide to Lex and Yacc
  - Compiler Construction using Flex and Bison

- Compiler Construction Tools
  - Catalogue of freeware and Commercial Resources for compiler writers: http://www.compilerconstruction.org

- Further resources through the course support page: http://www.iis.ee.ic.ac.uk/yiannis/lp/
Why a Language Processors course

[Apart from the obvious intellectual curiosity for the internal mechanisms of software that you use everyday as ISEs]

- Crucial link between high-level languages and low-level software and hardware, linking several of the courses you have encountered so far.
- LP construction has a wide applicability: compiler construction techniques can be applied outside the immediate field. Language Processors contain generally useful theory and algorithms.
- Note that we will place special emphasis on the acquisition of practical skills, not simply provide encyclopaedic coverage of the subject.
- Tutorials and labs will reinforce this.
What is language processor? (1)

Introduction

- It's an example of a “system program” – class of tools designed to help software developers.
- Allows the development of applications in the language most appropriate to the task, removing the need for developing at the machine level. Programmers can ignore the machine-dependent details of programming.
- Receives a textual representation of an algorithm in a source language, and produces as output a representation of the same algorithm in the object or target language.

Source language  \[\rightarrow\]  Language processor  \[\rightarrow\]  Object or Target language

E2.15 - Language Processors (Lect 1)
What is language processor? (2)

Types of language processing

[What kinds have you already met so far?]

- **Assemblers**: Language processors that map low-level language instructions into machine code, *e.g.* the ARM assembler.
- **Compilers**: Language processors that map high-level language instructions into machine code, *e.g.* Delphi, GCC, Visual C++ etc.
- **Pre-processors**: Language processors that map a *superset* of a high-level language into the original high-level language, or perform simple text substitutions before translation takes place.
- **Interpreters**: Language processors that include an execution component, *i.e.* they perform the operations specified in the source text, rather than re-expressing them in another language; *e.g.* Matlab
- **Disassemblers**: Language processors that attempt to take object code at a low level and regenerate source code at a higher level.
The language processor itself is written in what is called the implementation language.

These relations are often represented in “T-diagrams”
What is language processor? (4)

T-Diagram Example

This diagram represents a language processor which translates Pascal to machine code; The processor is called Delphi, and is written in C++
Language Processor Examples

- GCC, Delphi, Visual Studio
- TeX/LaTeX
- Postscript
- HTML & Web browsers
- XML
>> a=1;
>> b=2;
>> c=a+b

>> Fs=1000;
>> t=0:1/Fs:0.1;
>> x=sin(2*pi*20*t);
>> plot(t,x)
>> n=randn(1,length(x));
>> y=x+n;
Language Processor basics (1)
Compiling and running a compiler

- Compiler source text in implementation language
- Compiler source text in source language
- Input for program

- Compiler for implementation language
- Compiler for source language
- Program

- Executable compiler code
- Executable program code for target machine
- Program Output
When source language = implementation language and the source text to be compiled is actually a new version of the language processor itself, the process is called *bootstrapping*.

The compilation of the compiler itself does not need to be done on the target machine, but instead it can take place on another machine; this is called *cross-compilation*.
Language processing would not be much different than file conversion, if it wasn’t for the “semantics” of the language, *its meaning*, which needs to be preserved by the process.

In other words, there is a structure in the input text, and the semantics of the input is described in terms of and is attached to that structure. The output language also has its own semantics – the compiler essentially reformulates the collected semantics in terms of the target language.
Front-end: the part of the language processor that performs the analysis of the source language.

Back-end: the part of the language processor that does the target language synthesis.

In a fully-modular design the front-end is totally unaware of the back-end – in practice, this is inefficient, and even the best-structured compilers compromise.
Language Processor Basics (5)
Conceptual structure in detail (1)
The program text input module: finds and reads the program text file, and turns it into a stream of characters, allowing for new lines, escape codes, tabs etc. It may also switch to other files when these are to be included.

Lexical analysis module: Isolates tokens in the input stream, and determines their class and representation. Also preprocesses the input, for example by removing comments etc.

```
while A>3 do
    A = A - 1;
```

<table>
<thead>
<tr>
<th>While</th>
<th>keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>identifier - name A</td>
</tr>
<tr>
<td>&gt;</td>
<td>Operator - comparison</td>
</tr>
<tr>
<td>3</td>
<td>constant – value 3</td>
</tr>
<tr>
<td>Do</td>
<td>keyword</td>
</tr>
<tr>
<td>A</td>
<td>identifier – name A</td>
</tr>
<tr>
<td>=</td>
<td>operator - assignment</td>
</tr>
<tr>
<td>A</td>
<td>identifier - name A</td>
</tr>
<tr>
<td>-</td>
<td>operator - subtraction</td>
</tr>
<tr>
<td>1</td>
<td>constant – value 1</td>
</tr>
</tbody>
</table>
Syntax analysis module: involves grouping the tokens into grammatical phrases, usually represented as syntax or parse trees; it does so on the basis of the language’s grammar.

**Syntax Tree (AST)**

- `While` keyword
- `A` identifier - name A
- `>` Operator - comparison
- `3` constant – value 3
- `Do` keyword
- `A` identifier – name A
- `=` operator - assignment
- `1` constant – value 1
Context handling module: collects context information from various places in the program, and annotates nodes with the results; examples:
- Relating type information from declarations to expressions
- Connecting goto statements to their labels

Intermediate Code Generation module: translates language-specific constructs into the AST into more general constructs; example:
- Replacing while statements by tests, labels and jumps

Intermediate code optimisation module: preprocess the intermediate code improving its effectiveness; example:
- Constant folding: all operations in expressions with known simple operants are performed.
Language Processor Basics (5)
Conceptual Structure in detail (5)

- **Code generation module**: generates code for the target machine
  - Selects instructions for segments of the AST
  - Allocates registers to hold the data
  - Arranges instructions in the right order

- **Target code optimisation module**: Tries to optimise the list of symbolic machine instructions by replacing sequences by other faster or shorter target-machine-specific sequences.

- **Machine code generation module**: Converts the symbolic machine instructions into the corresponding machine code. Determines machine addresses of program code and data and produces tables of constants and relocation tables.
Many language processors are capable of generating their output with just one read, a single-pass, through the source code.

In some cases this is not possible, and a multi-pass process is required, for example:
- If the language allows variables to be declared after their first reference to the identifier.
- If the language allows GOTO statements

Another classification results from considering the amount of information a language processor reads in
- Narrow compiler: if it reads a small part of the program (e.g. a few tokens), process the information, outputs the object code, discards the read information, and reads the next bunch.
- Broad Compiler: if it reads the entire program and applies a series of transformations which eventually will result in the object code.
Criteria for LP design

How do we judge the quality of a language processor?

- (Most important property): Correctness of the generated code
- Conformity to the language specification; avoids temptations to implement a subset/superset of the language, which might result in a reduction of portability.
- Quality of the generated code with respect to size and speed.
- Speed of the language processor itself.
- User-friendliness, as evident in its quality of error reporting.
History of Language Processors

- Term “Compiler” coined by Grace Murray Hopper in the early 1950s --- translation was viewed as the “compilation” of sequences of canned subprograms from a library.
- First compilers appeared for FORTRAN in the late 1950s; the first one (Backus et al, 1957) reportedly took 18 staff-years to implement, and placed special emphasis on code optimisation.
- The 1960s-1970s saw a proliferation of new languages; emphasis shifted to “front-ends”, and to automatic generation of parsing code.
- More recently: new paradigms in programming languages (functional/object-oriented/logic/distributed languages) led to new demands in run-time requirements.
Summary

Language Processors:

- Translate their input from the source language to a target language, in one or more passes.
- Analyse the input, converting the stream of characters into tokens (lexical analysis), the tokens into an abstract syntax tree (syntax analysis), the AST into an annotated AST (context analysis), the AAST into intermediate code (IC generation), and after several stages optimisations, generate the target code
- Should generate correct, efficient, and compact code, and provide helpful error messages

Next lecture:
Grammars
Recommended Reading

- Read sections 1.1 -> 1.8 (inclusive) of Grune et al.
- Chapter 1 of Aho et al
- Chapters 1,2,3 of Terry