### Administrivia

- **Lecturer:**
  - Kostis Sagonas (Hus 1, 352)
- **Course home page:**
  - [http://user.it.uu.se/~kostis/Teaching/KT1-11](http://user.it.uu.se/~kostis/Teaching/KT1-11)
- **If you want to be enrolled in the course, send mail with your name and UU account to:**
  - kostis@it.uu.se
- **Assistant:**
  - Stavros Aronis (stavros.aronis@it.uu.se)
  - responsible for the lessons and the assignments

### Course Structure

- Course has theoretical and practical aspects
- Need both in programming languages!
- Written examination = theory (4 points)
- Assignments = practice (1 point)
  - Electronic hand-in to the assistant before the corresponding deadline

### Course Literature

- **Compilers**
  - Principles, Techniques, & Tools
  - Second Edition
  - Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

- **Engineering a Compiler**
  - Joel D. Oviatt

- **Crafting a Compiler**
  - Richard P. Jones

- **Modern Compiler Implementation in ML**
  - Philip LeFevre
**Academic Honesty**

- For assignments you are allowed to work in pairs (but no threesomes/foursomes/...)
- Don’t use work from uncited sources
  - Including old assignments

---

**The Compiler Project**

- A follow-up course
- that will be taught by Sven-Olof Nyström
- in period 3

---

**How are Languages Implemented?**

- Two major strategies:
  - Interpreters (older, less studied)
  - Compilers (newer, much more studied)
- Interpreters run programs “as is”
  - Little or no preprocessing
- Compilers do extensive preprocessing

---

**Language Implementations**

- Batch compilation systems dominate
  - gcc
- Some languages are primarily interpreted
  - Java bytecode
  - Postscript
- Some environments (e.g. Lisp) provide both
  - Interpreter for development
  - Compiler for production
(Short) History of High-Level Languages

• 1953 IBM develops the 701

• Till then, all programming done in assembly

• Problem: Software costs exceeded hardware costs!

• John Backus: “Speedcoding”
  - An interpreter
  - Ran 10-20 times slower than hand-written assembly

FORTRAN I

• 1954 IBM develops the 704

• John Backus
  - Idea: translate high-level code to assembly
  - Many thought this impossible
    - Had already failed in other projects

• 1954-7 FORTRAN I project

• By 1958, >50% of all software is in FORTRAN

• Cut development time dramatically
  - (2 weeks → 2 hours)

FORTRAN I

• The first compiler
  - Produced code almost as good as hand-written
  - Huge impact on computer science

• Led to an enormous body of theoretical work

• Modern compilers preserve the outlines of the FORTRAN I compiler

The Structure of a Compiler

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. IR Optimization
5. Code Generation
6. Low-level Optimization

The first 3, at least, can be understood by analogy to how humans comprehend English.
Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

- Note the
  - Capital “T” (start of sentence symbol)
  - Blank “ “ (word separator)
  - Period “." (end of sentence symbol)

More Lexical Analysis

- Lexical analysis is not trivial. Consider:
  
ist his ase nte nce

- Plus, programming languages are typically more cryptic than English:
  
*p->f ++ = -.12345e-5

And More Lexical Analysis

- Lexical analyzer divides program text into “words” or “tokens”
  
  if (x == y) then z = 1; else z = 2;

- Units:
  
  if, (, x, ==, y, ), then, z, =, 1, ;, else, z, =, 2, ;

Parsing

- Once words are understood, the next step is to understand the sentence structure

- Parsing = Diagramming Sentences
  
  - The diagram is a tree
Diagramming a Sentence (1)

This line is a longer sentence

- article
- noun
- verb
- article
- adjective
- noun

- noun phrase
  - noun phrase
  - verb phrase
- sentence

Diagramming a Sentence (2)

This line is a longer sentence

- article
- noun
- verb
- article
- adjective
- noun

- subject
- object
- sentence

Parsing Programs

- Parsing program expressions is the same
- Consider:
  - If \((x == y)\) then \(z = 1\); else \(z = 2\);
- Diagrammed:
  - \(x == y\)
  - \(z = 1\)
  - \(z = 2\)
  - relation
  - assignment
  - assignment
  - predicate
  - then-stmt
  - else-stmt
  - if-then-else

Semantic Analysis

- Once sentence structure is understood, we can try to understand its “meaning”
  - But meaning is too hard for compilers
- Most compilers perform limited analysis to catch inconsistencies
- Some optimizing compilers do more analysis to improve the performance of the program
Semantic Analysis in English

• Example:
  Jack said Jerry left his assignment at home. What does “his” refer to? Jack or Jerry?

• Even worse:
  Jack said Jack left his assignment at home? How many Jacks are there? Which one left the assignment?

Semantic Analysis in Programming Languages

• Programming languages define strict rules to avoid such ambiguities

  ```
  int Jack = 3;
  {
    int Jack = 4;
    cout << Jack;
  }
  ````

  This C++ code prints “4”; the inner definition is used

More Semantic Analysis

• Compilers perform many semantic checks besides variable bindings

• Example:
  Arnold left her homework at home.

• A “type mismatch” between her and Arnold; we know they are different people
  - Presumably Arnold is male

Optimization

• No strong counterpart in English, but akin to editing

• Automatically modify programs so that they
  - Run faster
  - Use less memory/power
  - In general, conserve some resource more economically

• The compilers project has no optimization component
  - for those interested there is KT2!
Optimization Example

\[ X = Y \times 0 \text{ is the same as } X = 0 \]

\textbf{NO!}

Valid for integers, but not for floating point numbers

Code Generation

- Produces assembly code (usually)
- A translation into another language
  - Analogous to human translation

Intermediate Languages

- Many compilers perform translations between successive intermediate forms
  - All but first and last are \textit{intermediate languages} internal to the compiler
  - Typically there is one IL

- IL's generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly

 Intermediate Languages (Cont.)

- IL's are useful because lower levels expose features hidden by higher levels
  - registers
  - memory/frame layout
  - etc.

- But lower levels obscure high-level meaning
Issues

• Compiling is almost this simple, but there are many pitfalls

• Example: How are erroneous programs handled?

• Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design

Compilers Today

• The overall structure of almost every compiler adheres to our outline

• The proportions have changed since FORTRAN
  - Early:
    • lexical analysis, parsing most complex, expensive
  - Today:
    • semantic analysis and optimization dominate all other phases; lexing and parsing are well-understood and cheap

Current Trends in Compilation

• Compilation for speed is less interesting. But:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures, GPUs)

• Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting data races
  - ...

Programming Language Economics

• Programming languages are designed to fill a void
  - enable a previously difficult/impossible application
  - orthogonal to language design quality (almost)

• Programming training is the dominant cost
  - Languages with a big user base are replaced rarely
  - Popular languages become ossified
  - but it is easy to start in a new niche...
Why so many Programming Languages?

- Application domains have distinctive (and sometimes conflicting) needs

Examples:
- **Scientific computing**: High performance
- **Business**: Report generation
- **Artificial intelligence**: Symbolic computation
- **Systems programming**: Efficient low-level access
- Other special purpose languages...

---

Topic: Language Design

- No universally accepted metrics for design
- “A good language is one people use”
- NO!
  - Is COBOL the best language?
- Good language design is hard

---

Language Evaluation Criteria

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplicity</td>
<td>YES</td>
</tr>
<tr>
<td>Data types</td>
<td>YES</td>
</tr>
<tr>
<td>Syntax design</td>
<td>YES</td>
</tr>
<tr>
<td>Abstraction</td>
<td>YES</td>
</tr>
<tr>
<td>Expressivity</td>
<td>YES</td>
</tr>
<tr>
<td>Type checking</td>
<td>YES</td>
</tr>
<tr>
<td>Exceptions</td>
<td>YES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Readability</th>
<th>Writeability</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

---

History of Ideas: Abstraction

- Abstraction = detached from concrete details
- Necessary for building software systems

Modes of abstraction:
- Via languages/compilers
  - Higher-level code; few machine dependencies
- Via subroutines
  - Abstract interface to behavior
- Via modules
  - Export interfaces which hide implementation
- Via abstract data types
  - Bundle data with its operations
History of Ideas: Types

- Originally, languages had only few types
  - FORTRAN: scalars, arrays
  - LISP: no static type distinctions
- Realization: types help
  - provide code documentation
  - allow the programmer to express abstraction
  - allow the compiler to check among many frequent errors and sometimes guarantee various forms of safety
- More recently:
  - experiments with various forms of parameterization
  - best developed in functional languages

History of Ideas: Reuse

- Exploits common patterns in software development
- Goal: mass produced software components
- Reuse is difficult
- Two popular approaches (combined in C++)
  - Type parameterization (List(Int) & List(Double))
  - Class and inheritance: C++ derived classes
- Inheritance allows:
  - specialization of existing abstractions
  - extension, modification and information hiding

Current Trends

- Language design
  - Many new special-purpose languages
  - Popular languages to stay
- Compilers
  - More needed and more complex
  - Driven by increasing gap between
    - new languages
    - new architectures
  - Venerable and healthy area

Why study Compiler Design?

- Increase knowledge of common programming constructs and their properties
- Improve understanding of program execution
- Increase ability to learn new languages
- Learn how to build a large and reliable system
- Learn new (programming) techniques
- See many basic CS concepts at work