1DL321: Kompilatorteknik I  
(Compiler Design 1)

Introduction to Programming 
Language Design and to Compilation

Administrivia

- Lecturer: 
  - Kostis Sagonas (kostis@it.uu.se)
- Course home page: 
  http://www.it.uu.se/edu/course/homepage/komp/h18
- Assistant(s): 
  - Magnus Lång (magnus.lang@it.uu.se)
  - possibly 1/2 more ...
  responsible for the lessons and assignments

Course Structure

- Course has theoretical and practical aspects
- Need both in programming languages!
- Written examination = theory (4 credits)
  - first exam scheduled: 11 January 2019
- Four assignments = practice (1 credit)
  - Electronic hand-in to the assistant before the corresponding deadline
  - You can submit one late assignment, if you need to, but it cannot be later than the deadline of the next assignment (for 1-3) or Dec. 29th (for 4th)

Course Literature

- Compilers
- Engineering a Compiler
- Crafting a Compiler
- Modern Compiler Implementation in ML
Academic Honesty

- For assignments you are allowed to work in pairs (but no threesomes/foursomes/...)
- Don’t share your work with others
  - e.g. post on public repositories
- Don’t use work from non-cited sources
  - including old assignments

PLAGIARISM

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

- Today, batch compilation systems dominate
  - gcc, clang, ...

- Some languages are primarily interpreted
  - Java bytecode compiler (javac)
  - Scripting languages (perl, python, javascript, ruby)

- Some languages (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 is 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 $\rightarrow$ 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 phases can be understood by analogy to how humans comprehend natural languages (e.g. Swedish or English).

First Step: Lexical Analysis

• 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”
  
  \[ \text{if} (x == y) \text{ then } z = 1; \text{ else } z = 2; \]

- Units:
  
  \[ \text{if, } (, x, ==, y, ), \text{ then, } z, =, 1, ;, \text{ else, } z, =, 2, ; \]

Second Step: Syntax Analysis (Parsing)

- Once words are identified, 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

- verb
- article
- noun

subject

object

sentence

Parsing Programs

- Parsing program expressions is the same
- Consider:
  
  \[
  \text{if (} x = y \text{)} \text{ then } z = 1; \text{ else } z = 2; 
  \]
- Diagrammed:

  \[
  \begin{array}{c}
  x \quad y \quad z = 1 \quad z = 2 \\
  \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
  \text{relation} \quad \text{assignment} \quad \text{assignment} \\
  \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
  \text{predicate} \quad \text{then-stmt} \quad \text{else-stmt} \\
  \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
  \text{if-then-else}
  \end{array}
  \]

Third Step: Semantic Analysis

- Once the 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
- This C++ code prints 42; the inner definition is used

```cpp
int Jack = 17;
{
    int Jack = 42;
    cout << Jack;
}
```

**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
    - avoid some source code redundancy
    - exploit the underlying hardware more effectively
  - Use less memory/cache/power
  - In general, conserve some resource more economically

**Optimization Example**

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

\textbf{NO!}

Valid for integers, but not for floating point numbers
<table>
<thead>
<tr>
<th><strong>Code Generation</strong></th>
<th><strong>Intermediate Languages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Produces assembly code (usually)</td>
<td>• Many compilers perform translations between successive intermediate forms</td>
</tr>
<tr>
<td>• A translation into another language</td>
<td>- All but first and last are <em>intermediate languages</em> internal to the compiler</td>
</tr>
<tr>
<td>- Analogous to human translation</td>
<td>- Typically there is one IL</td>
</tr>
<tr>
<td></td>
<td>• Intermediate languages generally ordered in descending level of abstraction</td>
</tr>
<tr>
<td></td>
<td>- Highest is source</td>
</tr>
<tr>
<td></td>
<td>- Lowest is assembly</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Intermediate Languages (Cont.)</strong></th>
<th><strong>Issues</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• IL’s are useful because lower levels expose features hidden by higher levels</td>
<td>• Compiling is almost this simple, but there are many pitfalls</td>
</tr>
<tr>
<td>- registers</td>
<td>• Example: How are erroneous programs handled?</td>
</tr>
<tr>
<td>- memory/frame layout</td>
<td>• Language design has big impact on compiler</td>
</tr>
<tr>
<td>- etc.</td>
<td>- Determines what is easy and hard to compile</td>
</tr>
<tr>
<td>• But lower levels obscure high-level meaning</td>
<td>- Course theme: many trade-offs in language design</td>
</tr>
</tbody>
</table>
### 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:
    - lexical analysis and parsing are well-understood and cheap
    - semantic analysis and optimization dominate
    - focus on concurrency/parallelism and interactions with the memory model of the underlying platform
    - optimization for code size and energy consumption

### Current Trends in Compilation

- Compilation for speed is less interesting. However, there are exceptions:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures, GPUs)
- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting data races
  - security properties
  - ...

### 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
  - **Web programming**: scripts that run everywhere
  - **Multicores**: concurrency and parallelism
  - Other special purpose languages...
### 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>Readability</td>
<td>Writeability</td>
</tr>
<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></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 your knowledge of common programming constructs and their properties
- Improve your understanding of program execution
- Increase your ability to learn new languages
- Learn how languages are implemented
- Learn new (programming) techniques
- See many basic CS concepts at work