Compiler Construction
1. Introduction

Oscar Nierstrasz
# Compiler Construction

<table>
<thead>
<tr>
<th>Lecturers</th>
<th>Prof. Oscar Nierstrasz, Dr. Mircea Lungu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assistants</td>
<td>Jan Kurš, Boris Spasojević</td>
</tr>
<tr>
<td>Lectures</td>
<td>E8 001, Fridays @ 10h15-12h00</td>
</tr>
<tr>
<td>Exercises</td>
<td>E8 001, Fridays @ 12h00-13h00</td>
</tr>
<tr>
<td>WWW</td>
<td>scg.unibe.ch/teaching/cc</td>
</tr>
</tbody>
</table>
MSc registration Spring 2015

JMCS students
• Register on Academia for teaching units by March 13, 2015
• Register on Academia for exams by May 15, 2015
• Request reimbursement of travel expenses by June 30, 2015

**NB:** Hosted JMCS students (*e.g.* CS bachelor students etc.) must additionally:
• Request for Academia access by February 28, 2015
Roadmap

- Overview
- Front end
- Back end
- Multi-pass compilers
- Example: compiler and interpreter for a toy language

See *Modern compiler implementation in Java* (Second edition), chapter 1.
Roadmap

> Overview
> Front end
> Back end
> Multi-pass compilers
> Example: compiler and interpreter for a toy language

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.

http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/
Other recommended sources

> **Compilers: Principles, Techniques, and Tools**, Aho, Sethi and Ullman

> **Parsing Techniques**, Grune and Jacobs
  — [http://www.cs.vu.nl/~dick/PT2Ed.html](http://www.cs.vu.nl/~dick/PT2Ed.html)

> **Advanced Compiler Design and Implementation**, Muchnik
<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20-Feb-15</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>27-Feb-15</td>
<td>Lexical Analysis</td>
</tr>
<tr>
<td>3</td>
<td>06-Mar-15</td>
<td>Parsing</td>
</tr>
<tr>
<td>4</td>
<td>13-Mar-15</td>
<td>Parsing in Practice</td>
</tr>
<tr>
<td>5</td>
<td>20-Mar-15</td>
<td>Semantic Analysis</td>
</tr>
<tr>
<td>6</td>
<td>27-Mar-15</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>03-Apr-15</td>
<td><strong>Good Friday</strong></td>
<td></td>
</tr>
<tr>
<td>10-Apr-15</td>
<td><strong>Spring break</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>17-Apr-15</td>
<td>Optimization</td>
</tr>
<tr>
<td>8</td>
<td>24-Apr-15</td>
<td>Code Generation</td>
</tr>
<tr>
<td>9</td>
<td>01-May-15</td>
<td>Bytecode and Virtual Machines</td>
</tr>
<tr>
<td>10</td>
<td>08-May-15</td>
<td>PEGs, Packrats and Parser Combinators</td>
</tr>
<tr>
<td>11</td>
<td>15-May-15</td>
<td>Program Transformation</td>
</tr>
<tr>
<td>12</td>
<td>22-May-15</td>
<td><em>Project Presentations</em></td>
</tr>
<tr>
<td>13</td>
<td>29-May-15</td>
<td><strong>Final Exam</strong></td>
</tr>
</tbody>
</table>
What is a compiler?

A program that translates an *executable* program in one language into an *executable* program in another language.

Translates “source code” into “target code”.

*We expect the program produced by the compiler to be “better”, in some way, than the original.*
What is an interpreter?

A program that reads an executable program and produces the results of running that program.

Usually, this involves executing the source program in some fashion.
This picture offers a high-level overview of the different approaches to implementing languages.
Why do we care?

Compiler construction is a microcosm of computer science

<table>
<thead>
<tr>
<th>artificial intelligence</th>
<th>greedy algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>learning algorithms</td>
</tr>
<tr>
<td>algorithms</td>
<td>graph algorithms</td>
</tr>
<tr>
<td></td>
<td>union-find</td>
</tr>
<tr>
<td></td>
<td>dynamic programming</td>
</tr>
<tr>
<td>theory</td>
<td>DFAs for scanning</td>
</tr>
<tr>
<td></td>
<td>parser generators</td>
</tr>
<tr>
<td></td>
<td>lattice theory for analysis</td>
</tr>
<tr>
<td>systems</td>
<td>allocation and naming</td>
</tr>
<tr>
<td></td>
<td>locality</td>
</tr>
<tr>
<td></td>
<td>synchronization</td>
</tr>
<tr>
<td>architecture</td>
<td>pipeline management</td>
</tr>
<tr>
<td></td>
<td>hierarchy management</td>
</tr>
<tr>
<td></td>
<td>instruction set use</td>
</tr>
</tbody>
</table>

*Inside a compiler, all these things come together*
Isn’t it a solved problem?

>Machines are constantly changing
— Changes in architecture ⇒ changes in compilers
— new features pose new problems
— changing costs lead to different concerns
— old solutions need re-engineering

> Innovations in compilers should prompt changes in architecture
— New languages and features

For example, computationally expensive but simpler scannerless parsing techniques are undergoing a renaissance.
What qualities are important in a compiler?

- Correct code
- Output runs fast
- Compiler runs fast
- Compile time proportional to program size
- Support for separate compilation
- Good diagnostics for syntax errors
- Works well with the debugger
- Good diagnostics for flow anomalies
- Cross language calls
- Consistent, predictable optimization

Each of these shapes your feelings about the correct contents of this course
<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1952</td>
<td>First compiler</td>
<td>First compiler (linker/loader) written by Grace Hopper for A-0 programming language</td>
</tr>
<tr>
<td>1957</td>
<td>First complete compiler</td>
<td>First complete compiler for FORTRAN by John Backus and team</td>
</tr>
<tr>
<td>1960</td>
<td>COBOL compilers</td>
<td>COBOL compilers for multiple architectures</td>
</tr>
<tr>
<td>1962</td>
<td>First self-hosting compiler</td>
<td>First self-hosting compiler for LISP</td>
</tr>
</tbody>
</table>

http://en.wikipedia.org/wiki/Compiler ;-}
A compiler was originally a program that “compiled” subroutines [a link-loader]. When in 1954 the combination “algebraic compiler” came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

— Bauer and Eickel [1975]
Abstract view

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agree on format for object (or assembly) code

Big step up from assembler — higher level notations
A classical compiler consists of a front end that parses the source code into an intermediate representation, and a back end that generates executable code.
A fallacy!

- must encode all the knowledge in each front end
- must represent all the features in one IR
- must handle all the features in each back end

*Limited success with low-level IRs*
Roadmap

> Overview
> **Front end**
> Back end
> Multi-pass compilers
> Example: compiler and interpreter for a toy language
Front end

- scanner
  - tokens
  - errors

- parser
  - IR

- recognize legal code
- report errors
- produce IR
- preliminary storage map
- shape code for the back end

*Much of front end construction can be automated*

- preliminary storage map => not only prepare symbol table, but decide what part of storage different names (entities) should be mapped to (local, global, automatic etc)
- shape code for the back end => decide how different parts of code are organized (in the IR)
• map characters to *tokens*
• character string value for a token is a *lexeme*
• eliminate white space

```
x = x + y
<id,x> = <id,x> + <id,y>
```

• character string value for a *token* is a *lexeme*
• *Typical tokens: id, number, do, end …*
• *Key issue is speed*
Parser

- recognize context-free syntax
- guide context-sensitive analysis
- construct IR(s)
- produce meaningful error messages
- attempt error correction

*Parser generators mechanize much of the work*
### Context-free grammars

**Context-free syntax** is specified with a grammar, usually in Backus-Naur form (BNF)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>&lt;goal&gt; := &lt;expr&gt;</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>&lt;expr&gt; := &lt;expr&gt; &lt;op&gt; &lt;term&gt;</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>&lt;term&gt;</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>&lt;term&gt; := number</code></td>
</tr>
<tr>
<td>5.</td>
<td><code>&lt;term&gt;</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>&lt;op&gt; := +</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>&lt;op&gt; := -</code></td>
</tr>
</tbody>
</table>

A grammar $G = (S, N, T, P)$
- $S$ is the start-symbol
- $N$ is a set of non-terminal symbols
- $T$ is a set of terminal symbols
- $P$ is a set of productions $P: N \rightarrow (N \cup T)^*$

Called “context-free” because rules for non-terminals can be written without regard for the context in which they appear.
Given a grammar, valid sentences can be derived by repeated substitution.

To recognize a valid sentence in some CFG, we reverse this process and build up a parse.

<table>
<thead>
<tr>
<th>Production</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;goal&gt;</td>
<td></td>
</tr>
<tr>
<td>1 &lt;expr&gt;</td>
<td></td>
</tr>
<tr>
<td>2 &lt;expr&gt; &lt;op&gt; &lt;term&gt;</td>
<td></td>
</tr>
<tr>
<td>5 &lt;expr&gt; &lt;op&gt; y</td>
<td></td>
</tr>
<tr>
<td>7 &lt;expr&gt; - y</td>
<td></td>
</tr>
<tr>
<td>2 &lt;expr&gt; &lt;op&gt; &lt;term&gt; - y</td>
<td></td>
</tr>
<tr>
<td>4 &lt;expr&gt; &lt;op&gt; 2 - y</td>
<td></td>
</tr>
<tr>
<td>6 &lt;expr&gt; + 2 - y</td>
<td></td>
</tr>
<tr>
<td>3 &lt;term&gt; + 2 - y</td>
<td></td>
</tr>
<tr>
<td>5 x + 2 - y</td>
<td></td>
</tr>
</tbody>
</table>

The parse is the sequence of productions needed to parse the input. The parse tree is something else …
Parse trees

A parse can be represented by a tree called a parse tree (or syntax tree).

Obviously, this contains a lot of unnecessary information
So, compilers often use an *abstract syntax tree* (AST).
Roadmap

- Overview
- Front end
- Back end
  - Multi-pass compilers
  - Example: compiler and interpreter for a toy language
Back end

- translate IR into target machine code
- choose instructions for each IR operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

*Automation has been less successful here*
Instruction selection

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - ad hoc techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming
Register allocation

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult

*Modern allocators often use an analogy to graph coloring*
Roadmap

> Overview
> Front end
> Back end
> **Multi-pass compilers**
> Example: compiler and interpreter for a toy language
Traditional three-pass compiler

- analyzes and changes IR
- goal is to reduce runtime *(optimization)*
- must preserve results

Code improvement and optimization
Modern optimizers are usually built as a set of passes

- constant expression propagation and folding
- code motion
- reduction of operator strength
- common sub-expression elimination
- redundant store elimination
- dead code elimination

- constant propagation and folding (evaluate and propagate constant expressions at compile time)
- code motion (move code that does not need to be reevaluated out of loops)
- reduction of operator strength (replace slow operations by equivalent faster ones)
- common sub-expression elimination (evaluate once and store)
- redundant store elimination (detect when values are stored repeatedly and eliminate)
- dead code elimination (eliminate code that can never be executed)
The MiniJava compiler

Cf. MCIJ 2d edn p 4
## Compiler phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lex</td>
<td>Break source file into individual words, or tokens</td>
</tr>
<tr>
<td>Parse</td>
<td>Analyse the phrase structure of program</td>
</tr>
<tr>
<td>Parsing Actions</td>
<td>Build a piece of abstract syntax tree for each phrase</td>
</tr>
<tr>
<td>Semantic Analysis</td>
<td>Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase</td>
</tr>
<tr>
<td>Frame Layout</td>
<td>Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way</td>
</tr>
<tr>
<td>Translate</td>
<td>Produce intermediate representation trees (IR trees), a notation that is not tied to any particular source language or target machine</td>
</tr>
<tr>
<td>Canonicalize</td>
<td>Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases</td>
</tr>
<tr>
<td>Instruction Selection</td>
<td>Group IR-tree nodes into clumps that correspond to actions of target-machine instructions</td>
</tr>
<tr>
<td>Control Flow Analysis</td>
<td>Analyse sequence of instructions into control flow graph showing all possible flows of control program might follow when it runs</td>
</tr>
<tr>
<td>Data Flow Analysis</td>
<td>Gather information about flow of data through variables of program; e.g., liveness analysis calculates places where each variable holds a still-needed (live) value</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Choose registers for variables and temporary values; variables not simultaneously live can share same register</td>
</tr>
<tr>
<td>Code Emission</td>
<td>Replace temporary names in each machine instruction with registers</td>
</tr>
</tbody>
</table>
Roadmap

> Overview
> Front end
> Back end
> Multi-pass compilers
> Example: compiler and interpreter for a toy language
A straight-line programming language (no loops or conditionals):

<table>
<thead>
<tr>
<th>Stm</th>
<th>→</th>
<th>Stm ; Stm</th>
<th>CompoundSt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stm</td>
<td>→</td>
<td>id := Exp</td>
<td>AssignStm</td>
</tr>
<tr>
<td>Stm</td>
<td>→</td>
<td>print ( ExpList )</td>
<td>PrintStm</td>
</tr>
<tr>
<td>Exp</td>
<td>→</td>
<td>id</td>
<td>IdExp</td>
</tr>
<tr>
<td>Exp</td>
<td>→</td>
<td>num</td>
<td>NumExp</td>
</tr>
<tr>
<td>Exp</td>
<td>→</td>
<td>Exp Binop Exp</td>
<td>OpExp</td>
</tr>
<tr>
<td>Exp</td>
<td>→</td>
<td>( Stm , Exp )</td>
<td>EseqExp</td>
</tr>
<tr>
<td>ExpList</td>
<td>→</td>
<td>Exp , ExpList</td>
<td>PairExpList</td>
</tr>
<tr>
<td>ExpList</td>
<td>→</td>
<td>Exp</td>
<td>LastExpList</td>
</tr>
<tr>
<td>Binop</td>
<td>→</td>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>Binop</td>
<td>→</td>
<td>–</td>
<td>Minus</td>
</tr>
<tr>
<td>Binop</td>
<td>→</td>
<td>×</td>
<td>Times</td>
</tr>
<tr>
<td>Binop</td>
<td>→</td>
<td>/</td>
<td>Div</td>
</tr>
</tbody>
</table>

a := 5 + 3; b := (print(a,a–1),10×a); print(b)

prints 8 7
80
Tree representation

```latex
\texttt{a := 5 + 3; b := (print(a, a-1), 10\times a); print(b)}
```
Straightline Interpreter and Compiler Files

Source files

- «Grammar spec»
  - slpl.jj

- «Compiler source»
  - CompilerVisitor ...

- «Interpreter source»
  - InterpreterVisitor ...

- «Abstract Machine for Interpreter»
  - Machine

Generated files

- «Parser source»
  - StraightLineParser ...

- «Default visitors and interfaces»
  - Visitor ...

- «Syntax Tree Nodes»
  - Goal ...

- «Grammar spec with actions»
  - jtb.out.jj

- «Bytecode»
  - StraightLineParser ...

Key
- generates

40
Java classes for trees

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;}
}
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;}
}
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;}
}
Straightline Interpreter and Compiler Runtime

- StraightLine source code
- Examples

- Syntax Tree
- Goal

- InterpreterVisitor...
  - Machine
  - output

- CompilerVisitor...
  - Bytecode generation library
    - bccl.jar
  - bytecode
  - output

Key:
generates
What you should know!

- What is the difference between a compiler and an interpreter?
- What are important qualities of compilers?
- Why are compilers commonly split into multiple passes?
- What are the typical responsibilities of the different parts of a modern compiler?
- How are context-free grammars specified?
- What is “abstract” about an abstract syntax tree?
- What is intermediate representation and what is it for?
- Why is optimization a separate activity?
Can you answer these questions?

- Is Java compiled or interpreted? What about Smalltalk? Ruby? PHP? Are you sure?
- What are the key differences between modern compilers and compilers written in the 1970s?
- Why is it hard for compilers to generate good error messages?
- What is “context-free” about a context-free grammar?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[http://creativecommons.org/licenses/by-sa/4.0/](http://creativecommons.org/licenses/by-sa/4.0/)