Compiler construction

Lecture 1: Introduction and project overview

Alex Gerdes
Vårtermin 2017
Chalmers University of Technology — Gothenburg University

Today

- Course info
- Introduction to compiling
- Some examples
- Project description

Course info

Compiler Construction 2017

What is it?

Hands-on, learning-by-doing course, where you implement your own compiler

Related course

Companion course to (and optional continuation of) Programming Language Technology in period 3

Focus

Compiler backend and runtime issues

Why learn to write a compiler?

Few people ever write (or extend, or maintain) compilers for real programming languages.

But knowledge of compiler technology is useful anyhow:

- Tools and techniques are useful for other applications – including but not limited to small-scale languages for various purposes
- Understanding compiling gives deeper understanding of programming language concepts – and thus makes you a more efficient programmer

Course aims

After this course you will:

- Have experience of implementing a complete compiler for a simple programming language, including
  - Lexical and syntactic analysis (using standard tools)
  - Type checking and other forms of static analysis
  - Code generation and optimization for different target architectures (LLVM, x86, ...)
- Understand basic principles of run-time organisation, parameter passing, memory management, etc. in programming languages
- Know the main issues in compiling imperative and object-oriented languages

Course organisation

Teachers[1]

- Alex Gerdes (course responsible, lectures, supervision, grading)
- Magnus Myreen (examiner, grading)
- Anton Ekblad (assistant, grading)

Lectures Fridays 13–15 (and later Tuesdays 13–15). Lots of holidays in this period, so check schedule!

Supervision On demand via email (anytime) or visit during my office hours, Thursdays 13–15

Group There is a Google group for announcements, asking questions and finding lab partners; make sure to sign up

[1]Email addresses and offices on course website

Examination

Grading

- U/3/4/5 and U/G/VG scale is used
- Your grade is entirely based on your project; there are several alternative options, detailed in the project description
- Need not decide on ambition level in advance
- Individual oral exam in exam week
- Details on the course website

Project groups

- We recommend that you work in groups of two
- Individual work is permitted but discouraged
- The course's Google group can be used to find project partner

Course evalutation

Evaluation the course

The course will be evaluated according to Chalmers course evaluation policy.

Student representatives

We have randomly selected a number of course representatives. Their names will be listed on the course webpage. If you do not want to be one, let me know. (we plan an introduction meeting after the lecture)

Introduction to compiling

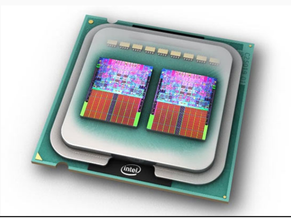
Compiler technology

- Very well-established field of computing science, with mature theory and tools for some subproblems and huge engineering challenges for others
- Compilers provide a fundamental infrastructure for all of computing; crucial to make efficient use of resources
- Advances in computer architecture lead to new challenges both in programming language design and in compiling

Current grand challenge

Multi-core processors.

How should programmers exploit parallellism?



What is a compiler?

A compiler is a translator

A compiler translates programs in one language (the source language) into another language (the target language).

Typically, the target laguage is more "low-level" than the source language.

Examples:

- C++ into assembly language
- Java into JVM bytecode
- JVM bytecode into x86 assembly
- Haskell into C

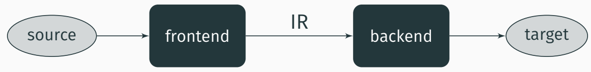
Why is compiling difficult?

The semantic gap

- The source program is structured into (depending on language) classes, functions, statements, expressions, …
- The target program is structured into instruction sequences, manipulating memory locations, stack and/or registers and with (conditional) jumps

Source code	x86 assembly	JVM assembly
`8*(x+5)-y`	`movl 8(%ebp), %eax`	`bipush 8`
	`sall $3, %eax`	`iload_0`
	`subl 12(%ebp), %eax`	`iconst_5`
	`addl $40, %eax`	`iadd`
		`imul`
		`iload_1`
		`isub`

Basic structure of a compiler



Intermediate representation

A notation separate from source and target language, suitable for analysis and improvement of programs.

Examples:

- Abstract syntax trees
- Three-address code
- JVM assembly

Front and back end

Front end: Source to IR

- Lexing
- Parsing
- Type-checking

Back end: IR to Target

- Analysis
- Code improvement
- Code emission

Some variations

One-pass or multi-pass

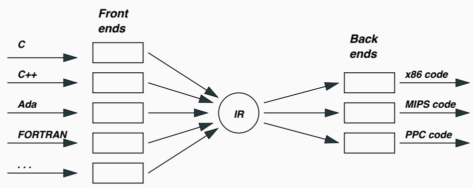
Already the basic structure implies at least two passes, where a representation of the program is input and another is output.

- For some source languages, one-pass compilers are possible
- Most compilers are multi-pass, often using several IRs

Pros and cons of multi-pass compilers

- − Longer compilation time
- − More memory consumption
- + SE aspects: modularity, portability, simplicity, …
- + Better code improvement
- + More options for source language

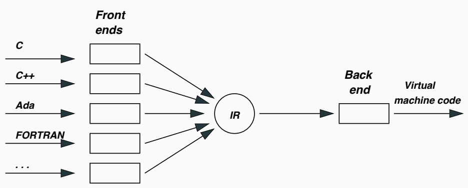
Compiler collections



More compilers with less work

- Compilers for m languages and n architectures with $m + n$ components
- Requires an IR that is language and architecture neutral
- Well-known example: GCC

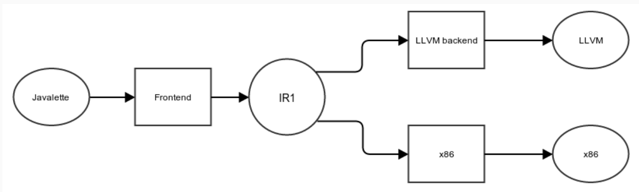
Compiling for virtual machines



Target code for virtual (abstract) machine

- Interpreter for virtual machine code written for each (real) architecture
- Can be combined with JIT compilation to native code
- Was popular 40 years ago but fell out of fashion
- Strongly revived by Java's JVM, Microsoft's .NET, LLVM

Our course project



Many options

- One or more backends: LLVM/x86 code
- Various source language extensions

More details follow in this lecture. See also the course website.

Front end tasks

```
if (x > 100) y = 1;
```

Lexing

Converts source code char stream to token stream.

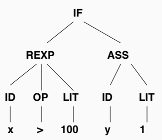
Good theory and tools.

```
IF LPAR ID/x GT LIT/100
RPAR ID/y EQ LIT/1 SEMI
```

Parsing

Converts token stream to abstract syntax trees (ASTs).

Good theory and tools.



Type-checking

Checks and annotates AST.

Good theory and programming patterns.

Back end tasks

Some general comments

- Not as well-understood, hence more difficult
- Several sub-problems are inherently difficult (e.g., NP-complete); hence heuristic approaches necessary
- Large body of knowledge, using many clever algorithms and data structures
- More diverse; many different IRs and analyses can be considered
- Common with many optimization passes; trade-off between compilation time and code quality

Compiling and linking

Why is linking necessary?

- With separate compilation of modules, even native code compiler cannot produce executable machine code
- Instead, object files with unresolved external references are produced by the compiler
- A separate linker combines object files and libraries, resolves references and produces an executable file

Separate compilation and code optimization

- Code improvement is easy within a basic block (code sequence with one entry, one exit and no internal jumps)
- More difficult across jumps
- Still more difficult when interprocedural improvement is tried
- And seldom tried across several compilation units

Examples

The beginning: FORTRAN 1954 – 57

Target machine: IBM704

- $\leq$ 36kb primary (magnetic core) memory
- One accumulator, three index registers
- $\approx$ 0.1 – 0.2 ms/instruction

Compiler phases

1. (Primitive) lexing, parsing, code generation for expressions
2. Optimization of arrays/DO loop code
3. Code merge from previous phases
4. Data flow analysis, preparing for next phase
5. Register assignment
6. Assembly

GCC: Gnu Compiler Collection 1985 –

Goals

- Free software
- Key part of the GNU operating system

Status

- 2.5 million lines of code, and growing
- Many front- and backends
- Very widespread use
- Monolithic structure, difficult to learn internals
- Up to 26 passes

LLVM (Low Level Virtual Machine) 2002 –

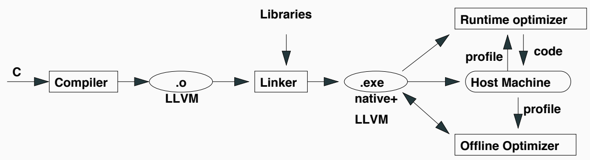
Goals

- Multi-stage code improvement, throughout life cycle
- Modular design, easy to grasp internal structure
- Practical, drop-in replacement for other compilers (e.g. GCC)
- LLVM IR: three-address code in SSA form, with type information

Status

- Front end (CLANG) released (for C, C++ and Obj. C)
- GCC front end adapted to emit LLVM IR
- LLVM back ends of good quality available

LLVM optimization architecture



Code optimization opportunities

- During compilation to LLVM (as in all compilers)
- When linking modules and libraries
- Recompilation of hot-spot code at run-time, based on run-time profiling (LLVM code part of executable)
- Off-line, when computer is idle, based on stored profile info

CompCert 2005 –

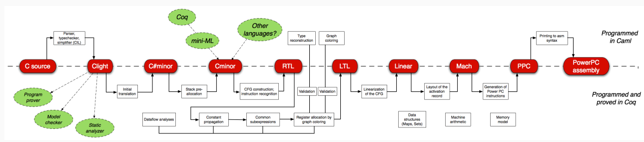
Program verification

- For safety-critical software, formal verification of program correctness may be worth the cost
- Such verification is typically done of the source program. So what if the compiler is buggy?

Use a certified compiler!

- CompCert is a compiler for a large subset of C, with PowerPC assembler as target language
- Written in Coq, a proof assistant for formal proofs
- Comes with a machine-checked proof that for any program, which does not generate a compilation error, the source and target programs behave identically

CompCert architecture

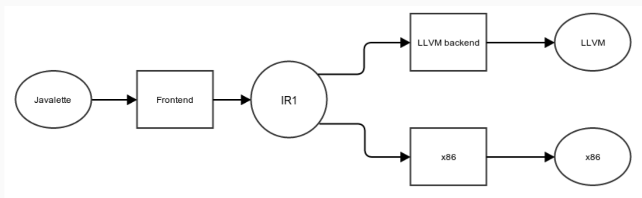


Intermediate constructions

- Eight intermediate languages
- Six type systems
- Thirteen passes

JavaLette

Project languages



Recall

- Two or more backends; JVM/LLVM/x86 code
- Various source language extensions

Today we will discuss the languages involved.

Source language

JavaLette

- A simple imperative language in C-like syntax
- A JavaLette program is a sequence of function definitions, that may be (mutually) recursive
- One of the functions must be called `main`, have result type `int` and no parameters

Restrictions

Basic language is very restricted: no arrays, no pointers, no modules, ...

Program environment

External functions

- Procedures:

```
void printInt(int i)
void printDouble(double d)
void printString(string s)
void error()
```

- Functions:

```
int readInt()
double readDouble()
```

One file programs

Except for calling the above routines, the complete program is defined in one file.

Types and literals

Types

JavaLette has the types

- `int`, with literals described by `digit+`
- `double`, with literals `digit+.digit+[(e|E)[+|−]digit+]`
- `boolean`, with literals `true` and `false`

In addition, the type `void` can be used as return type for “functions” to be used as statements.

Notes

- The type-checker may profit from having an internal type of functions
- String literals can be used as argument to `printString`; otherwise, there is no type of strings

Function definitions

Syntax

- A function definition has a result type, a name, a parameter list in parentheses and a body, which is a block (see below)
- A parameter list consists of parameter declarations separated by commas, which may be empty
- A parameter declaration is a type followed by a name

Return statements

- All functions must `return` a result of their result type
- Procedures may `return` without a value and may also omit the `return` statement (“fall off the end”)

Example of a function definition

```
int fact(int n) {
  int i, r;
  i = 1;
  r = 1;
  while (i < n + 1) {
    r = r * i;
    i++;
  }
  return r;
}
```

Statements

The following statements forms exist in JavaLette (details in project description):

- Empty statement
- Variable declaration
- Assignment statement
- Increment and decrement
- Return-statement
- Procedure call
- If-statement (with and without else-part)
- While-statement
- Block (a sequence of statements enclosed in braces)

The first six statement forms end with semicolon, blocks do not

Identifiers, declarations and scope

Identifiers

An identifier (a name) is a letter, optionally followed by letters, digits and underscores.

Reserved words (`else if return while`) are not identifiers.

Declarations

A variable (a name) must be declared before it is used.

Otherwise, declarations may be anywhere in a block.

Scope

A variable may only be declared once within a block.

A declaration shadows possible other declarations of the same variable in enclosing blocks.

Expressions

The following expression forms exist in JAVALETTE:

- Variables and literals
- Binary operator expressions with operators
  `+ - * / % < > >= <= == != && ||`
- Unary operator expressions with operators `-` and `!`
- Function calls

Notes

- `&&` and `||` have lazy semantics in the right operand
- Arithmetic operators are overloaded in types `int` and `double`, but both operands must have the same type (no casts!)

Part A of the project

Compiler front end, including

- Lexing and parsing
- Building an IR of abstract syntax trees
- Type-checking and checking that functions always 'return'
- BNFC source file for JAVALETTE offered for use

Deadline

You must submit part A at the latest Sunday, April 9 at midnight.

Late submissions will only be accepted if you have a really good reason.

Part B of the project

LLVM backend

Back end for LLVM. Typed version of three-address code (virtual register machine).

Submission deadline Sunday, April 30 at midnight.

If you plan to implement many extensions, then try to finish early and continue with part C.

Part C of the project

Extensions

One or more language extensions to JAVALETTE.

Submission deadline Sunday, May 21 at midnight.

Possible extensions

- JAVALETTE language extensions. One or more of the following:
  - For loops and arrays, restricted forms (two versions)
  - Dynamic data structures (lists, trees, etc.)
  - Classes and objects (two versions)
- Native code generator (support offered only for x86), needs complete treatment of function calls
- See full list in the project description on the course web page

LLVM

LLVM: a virtual register machine

Not so different from JVM

- Instead of pushing values onto a stack, store them in registers (assume unbounded supply of registers)
- Control structures similar to Jasmin
- High-level function calls with parameter lists

LLVM can be interpreted/JIT-compiled directly or serve as input to a retargeting step to real assembly code.

LLVM example

```
define i32 @main() {
entry: %t0 = call i32 @f(i32 7)
       call void @printInt(i32 %t0)
       ret i32 0
}

define i32 @f(i32 %__p__n) {
entry: %n = alloca i32
       store i32 %__p__n , i32* %n
       %i = alloca i32
       %r = alloca i32
       store i32 1 , i32* %i
       store i32 1 , i32* %r
       br label %lab0
```

LLVM example

```
lab0:  %t0 = load i32* %i
       %t1 = load i32* %n
       %t2 = icmp sle i32 %t0 , %t1
       br i1 %t2 , label %lab1 , label %lab2
lab1:  %t3 = load i32* %r
       %t4 = load i32* %i
       %t5 = mul i32 %t3 , %t4
       store i32 %t5 , i32* %r
       %t6 = load i32* %i
       %t7 = add i32 %t6 , 1
       store i32 %t7 , i32* %i
       br label %lab0
lab2:  %t8 = load i32* %r
       ret i32 %t8
}
```

What does `@f` calculate?

Optimization of LLVM code

Many possibilities

Important optimizations can be done using this IR, many based on data flow analysis (later lecture). LLVM tools are great for studying effects of various optimizations.

Examples:

- Constant propagation
- Common subexpression elimination
- Dead code elimination
- Moving code out of loops

You should generate straightforward code and rely on LLVM tools for optimization.

LLVM optimization example

```
proj> cat myfile.ll | llvm-as | opt -std-compile-opts
 > myfileopt.bc
proj> llvm-dis myfileopt.bc
proj> cat myfileopt.ll
```

```
declare void @printInt(i32)
define i32 @main() {
entry:
  tail call void @printInt(i32 5040)
  ret i32 0
}
```

continues on next slide

LLVM optimization example

```
define i32 @fact(i32 %__p__n) nounwind readnone {
entry:
  %t23 = icmp slt i32 %__p__n, 1
  br i1 %t23, label %lab2, label %lab1
lab1:
  %t86 = phi i32 [ %t5, %lab1 ], [ 1, %entry ]
  %t05 = phi i32 [ %t7, %lab1 ], [ 1, %entry ]
  %t5 = mul i32 %t86, %t05
  %t7 = add i32 %t05, 1
  %t2 = icmp sgt i32 %t7, %__p__n
  br i1 %t2, label %lab2, label %lab1
lab2:
  %t8.lcssa = phi i32 [ 1, %entry ], [ %t5, %lab1 ]
  ret i32 %t8.lcssa
}
```

From LLVM to (x86) assembly

The main tasks

- Instruction selection
- (Register allocation)
- (Instruction scheduling)
- Function calls: explicit handling of activation records, calling conventions, special registers, …

Final words

How to choose implementation language?

- Haskell is very well suited for these kind of problems. Data types and pattern-matching makes for efficient programming. State is handled by monadic programming; the second lecture will give some hints.
- Java and C++ are more mainstream, but will require a lot of code. But you get a visitor framework for free when using BNFC. BNFC patterns for Java are more powerful than for C++.

Testing

On the web site you can find a moderately extensive testsuite of JavaLette programs. Test at every stage!

You have a lot of code to design, write and test; it will take more time than you expect. Plan your work and allow time for problems!

What next?

- Find a project partner and choose implementation language
- Read the project instruction
- Get started!
- Really, get started!!!
- If you reuse front end parts, e.g., from Programming Language Technology, make sure you conform to JavaLette definition
- Front end should ideally be completed next week
- Do not wait

Good luck!