Compiler Construction

Introduction and overview

Lennart Andersson

Revision 2012-01-17

2012

Agenda

- Course registration, etc.
- Course structure
- Course contents, overview
- Learning goals

Course registration

You need to

- Confirm registration by signing the Registration Form
- Sign up for the programming assignment at
  - `https://sam.cs.lth.se/Labs`
  - Deadline: Friday Jan 20, 13:00

To unregister

- Mail me

Prerequisites

- Object-oriented programming and Java
- Algorithms and Data structures (recursion, trees, lists, hash tables, . . .)

Course information

- Web page: `http://cs.lth.se/EDA180`
  - will be updated during the course
- Literature
  - Course material, will be made available on the web site.
    - Lectures, seminars, labs, project
    - Not handed out - print yourself.
  - Textbook
    - A.W. Appel: Modern Compiler Implementation in Java, 2nd Edition, Cambridge University Press, 2002, ISBN: 0-521-82060-X.
    - Available as an e-book via `http://lu.summon.serialssolutions.com/`

Course structure

- 14 lectures
  - Tuesday 15-17, E:1406
  - Wednesday 10-12, E:1406
- 5 seminars (give extra points on exam)
  - Thursday 8-10, E:C
  - Start next week
- 6 computer assignments (mandatory)
  - Thursday or Friday 8-10 (Mars)
  - Start next week
  - Must be completed before exam and course project
- Written exam, March 7th
- Course project, VT2

People helping with the course

- Lectures: Lennart Andersson
- Guest lectures:
  - Emma Söderberg (semantic computations)
  - Jonas Skeppstedt (code optimization)
  - Roger Henriksson (automatic memory management)
- Seminars and Programming assignments:
  - Emma Söderberg

Seminars

Active participation gives extra points at the exam

- Before the seminar
  - Try to solve the problems at home. Write down your solutions, bring them with you, be prepared to present them at the seminar.
  - You are encouraged to cooperate when solving problems.
- At the seminar
  - Mark the problems you are willing to present solutions for.
  - The seminar leader selects some students to present their solutions. Discussion of the solutions.
- At the written exam (NB! Only for exams this year)
  - Your markings will give you a maximum of 15% extra points at the written exam.

Programming assignments

- Work in pairs
- Use the lecture break to form pairs!
- Make the preparations. You may complete the assignment before appearing at the session.
- Computer room session
  - Presentation
  - Supervisor feedback
  - Supplementary instruction - if needed

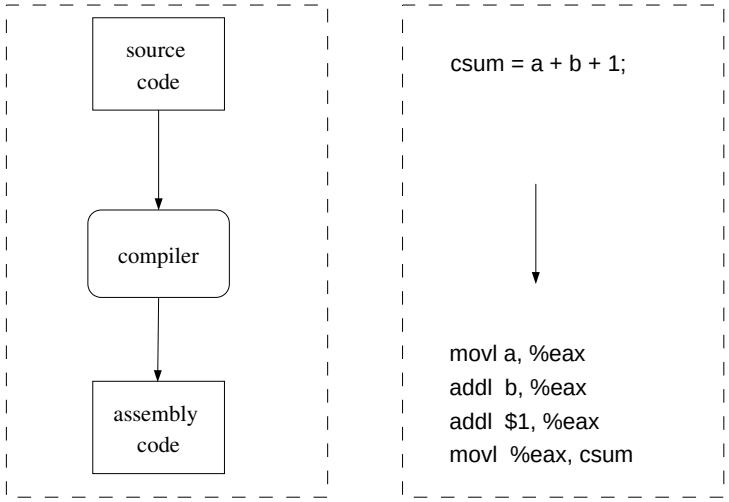
Examination

- Exams take place
  - Wednesday, March 7, 8-13, evak:3
  - Saturday, April 14, 8-13, E:2116 (Registration required, date may be changed)
- Prerequisites
  - Completed programming assignments

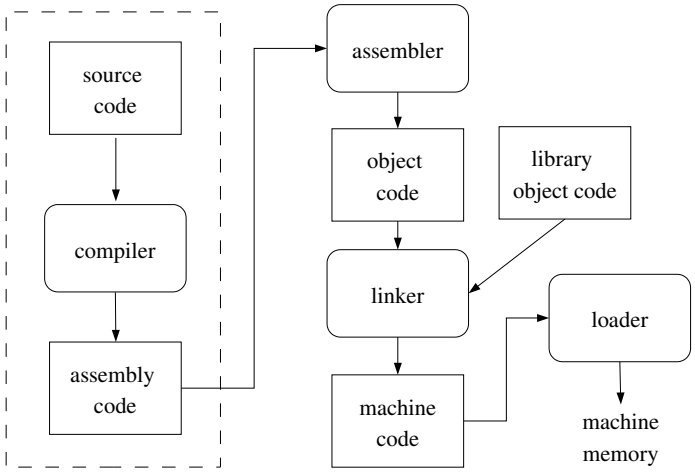
Project

- Standard project
  - Design of a small procedural language
  - Implementation of a compiler from source text to Intel assembly code
- Work in pairs
- Deadlines
  - Intermediate deadlines given later
  - Final deadline for completed and approved project: April 27th.

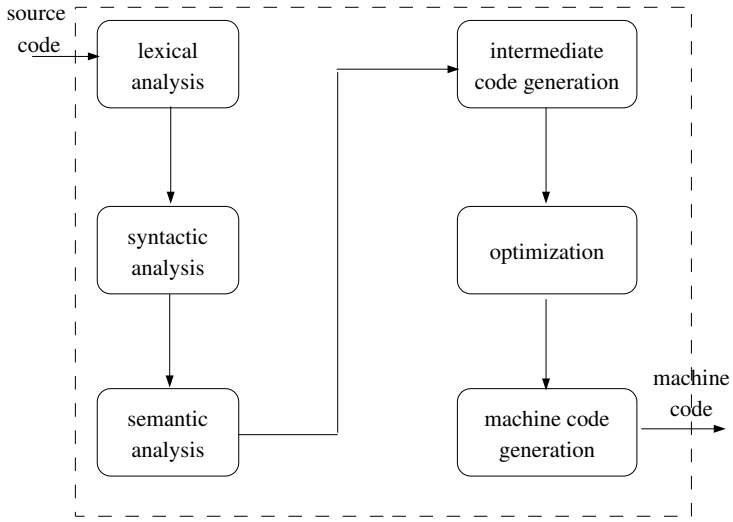
Project outcome



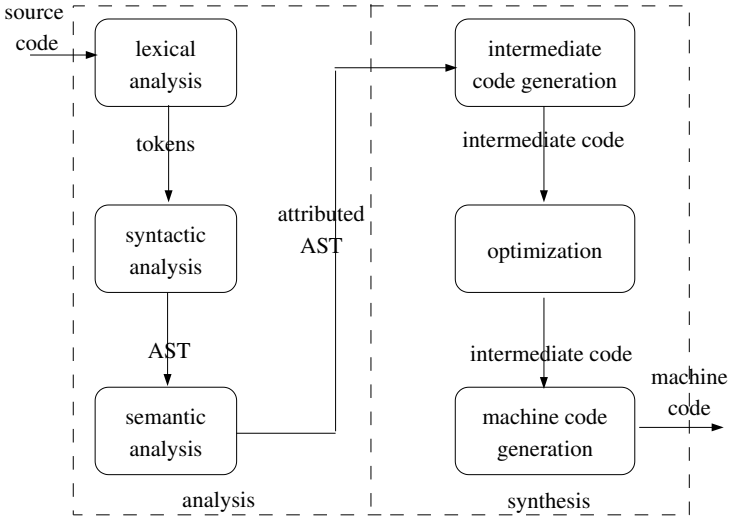
What happens after compilation?



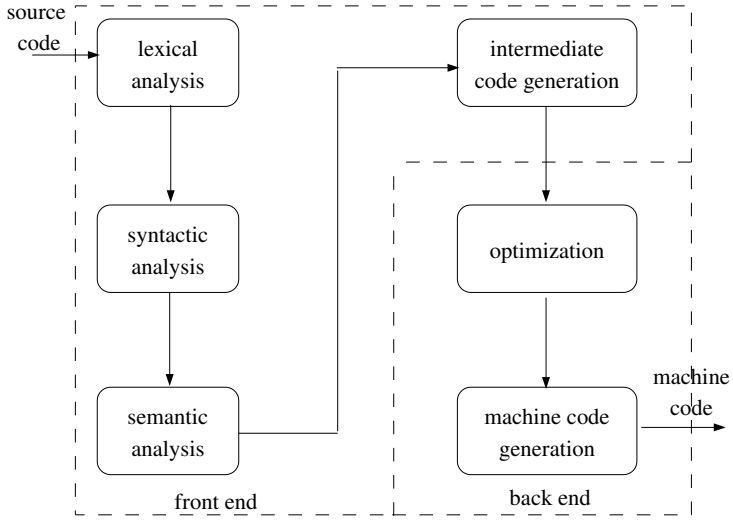
A closer look at the compiler



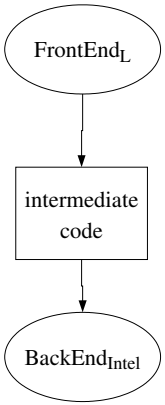
Intermediate representations



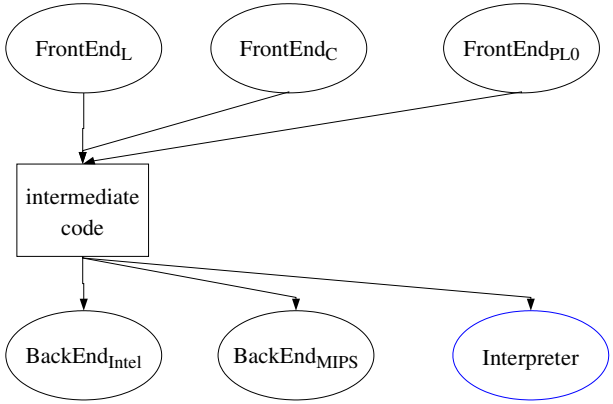
Front and back end



Intermediate code



Several front and back ends



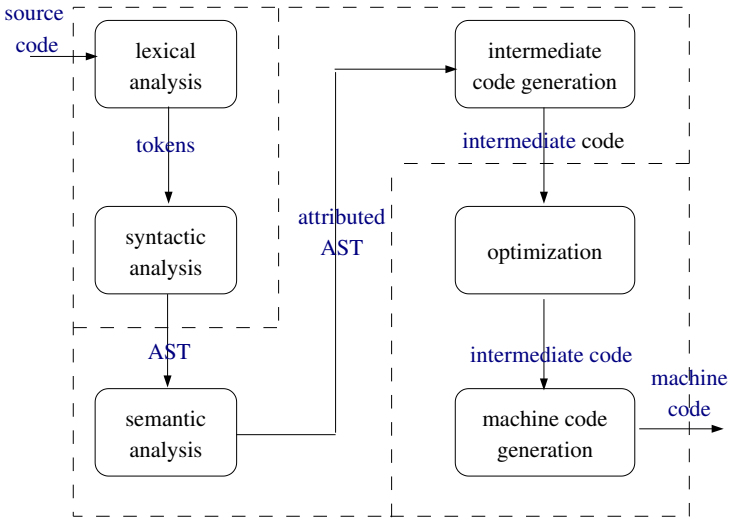
Why?

- It is more rational to implement m front ends and n back ends than m*n compilers.
- Many optimizations are best performed on intermediate code.
- It may be easier to debug the front end using an interpreter than a target machine.

Compilation and Interpretation

- A compiler translates a high level program to low level/machine code.
- An interpreter executes a high/low level program by calling one procedure for each program construct.
- An interpreter may use a JIT ("just in time") compiler to compile all or parts of the the program into machine code during execution.

Program representations



Lexical analysis (scanning)

```
Source text          Tokens

while (k<=n) {
  sum=sum+k;
  k=k+1;
}
```

A *token* is a symbolic name, sometimes with an attribute.
A *lexeme* is a string corresponding to a token.

Grammatik

statement → *assignment* | *whileStmt* | *compoundStmt*
assignment → *ID EQUALS expr*
whileStmt → *WHILE LPAR expr RPAR statement*
compoundStmt → *LBRACE statements RBRACE*
statements → ...
expr → ...

Syntactic analysis (parsing)

```
while ( k <= n )   { sum = sum + k  ;  k = k + 1 ; }
```

Abstract Syntax Tree

- An *Abstract Syntax Tree*, AST, is an object oriented model of a program/statement/expr etc.
- It may be similar to the parse tree but has only essential nodes.
- Every node is an object and sub-trees correspond to attributes.

AST – while (k<=n) {sum=sum+k; k=k+1;}

`WhileStmt`

AST class hierarchy

- Create class hierarchies for statements and expressions!
- Invent names for suitable abstract classes!
- Which methods are required to traverse the AST?

Program representations

Semantic analysis

Analyze the AST, e.g.

- Which variable corresponds to which declaration?
- What is the type of an expression?
- Are there compile time errors in the program?

Formalisms

- Regular expression for
  - defining lexemes
  - automatic generation of scanners
- Context-free grammars for
  - defining concrete grammars
  - automatic generation of parsers
- ast grammars for
  - defining abstract grammars
  - automatic generation of Java classes
- jadd specifications for
  - defining attributes and methods for AST classes
  - automatic distribution into Java classes

Tools

- JavaCC (Sun/Open source)
  - Scanner and parser generator
- JJTree (Sun/Open source)
  - adds AST building to JavaCC
  - implements the Visitor pattern for ASTs
- JJDoc (Sun/Open source)
  - creates a BNF grammar from JavaCC data
- JastAdd (LTH/Open source)
  - generates Java classes
  - supports aspect oriented programming
- as (GNU/Open source)
  - translates assembly code to machine code

Other tools

- Ant (Apache/Open source)
  - Software system builder
- JUnit (Object Mentor/Open source)
  - testing framework
- Gdb (GNU/Open source)
  - debugger

Synthesis

- Runtime systems
  - How are variables accessed and procedures called?
  - How are objects and classes represented?
  - How is memory reused?
- Intermediate code generation
  - Straight-forward mapping from AST
  - Use unlimited number of registers (temporaries)
- Optimization
  - Not covered
- Machine code generation
  - Instruction selection
  - Register allocation

Paradigms

- Imperative languages
  - Procedural
  - Object oriented
  - Aspect oriented
- Declarative languages
  - Functional
  - Logical
  - Constraint
- Hybrid languages

Applications of compiler construction

- Traditional compilers from source to assembly
- Source-to-source translators, preprocessors
- Interpreters and virtual machines
- Integrated programming environments
- Analysis tools
- Refactoring tools
- Domain-specific languages

Related research at LTH

- Compiler tools (Görel Hedin)
- Natural language processing (Pierre Nugues)
- Constraint solvers (Krzysztof Kuchcinski)
- Real-time garbage collection (Roger Henriksson)
- Code optimization for multiprocessors (Jonas Skeppstedt)
- Real-time Java (Anders Nilsson)
- Domain specific languages (Johan Åkesson)

Course goals

- The student should be able to
  - use regular expressions
  - use context-free grammars
  - use abstract grammars
  - describe runtime systems and garbage collection
- The student should be able to
  - use parser generators
  - make semantic analysis
  - do code generation
- The student should be able to use
  - static aspect oriented programming
  - the visitor pattern

Related courses

- Optimizing compilers, Jonas Skeppstedt, HT1 2012
- Language technology, Pierre Nugues, HT1, 2012

Readings

- F1: Introduction
  - Appel, chapter 1
- F2: Regular expressions
  - Appel, chapter 2

Review questions

- Which are the major compilation phases?
- What is the difference between the analysis and syntheses phases?
- Why do we use intermediate code?
- What is the advantage of separating the front and back ends?
- What is
  - a lexeme,
  - a token,
  - a parse tree,
  - an abstract syntax tree,
  - intermediate code,
  - assembly code?