Compiler Design

Chapter 1

Introduction
Outline

• Scope of the course
• Disciplines involved in it
• Abstract view for a compiler
• Front-end and back-end tasks
• Modules
• Straight line compiler
Course scope

• **Aim:**
  - To learn techniques of a modern compiler

• **Main reference:**
  - Modern compiler construction in Java 2\textsuperscript{nd} edition

• **Supplementary references:**
  - Compilers -Principles, Techniques and Tools by Alfred V. Aho and et. al.
  - Advanced Compiler Design and Implementation by Muchnick
Subjects

• Lexical analysis
• Syntax Analysis (Parsing)
• Abstract Syntax
• Semantic Analysis
• Activation Record
• Translation to intermediate code
• Basic blocks and traces
• Instruction selection
• Liveness analysis
• Register allocation
• Object oriented languages
Compiler learning

• Isn’t it an old discipline?
  - Yes, it is a well-established discipline
  - Algorithms, methods and techniques are researched and developed in early stages of computer science growth
  - There are many compilers around and many tools to generate them automatically

• So, why we need to learn it?
  - Although you may never write a full compiler
  - But the techniques we learn is useful in many tasks like writing an interpreter for a scripting language, validation checking for forms and so on
Terminology

• **Compiler:**
  - a program that translates an *executable* program in one language into an *executable* program in another language
  - we expect the program produced by the compiler to be better, in some way, than the original

• **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

• Our course is mainly about compilers but many of the same issues arise in interpreters
Disciplines involved

- Algorithms
- Languages and machines
- Operating systems
- Computer architectures
Recognizes legal (and illegal) programs
Generate correct code
Manage storage of all variables and code
Agreement on format for object (or assembly) code
Front-end, Back-end division

- Front end maps legal code into IR
- Back end maps IR onto target machine
- Simplify retargeting
- Allows multiple front ends
- Multiple passes $\rightarrow$ better code
Front end

- Recognize legal code
- Report errors
- Produce IR
- Preliminary storage maps
Front end

- **Scanner:**
  - Only phase to interact directly with the source program
  - Maps characters into tokens - the basic unit of syntax
    - $x = x + y$ becomes $\langle \text{id}, x \rangle = \langle \text{id}, x \rangle + \langle \text{id}, y \rangle$
  - Typical tokens: number, id, +, -, *, /, do, end
  - Eliminate white space (tabs, blanks, comments)

- **A key issue is speed so instead of using a tool like LEX it sometimes needed to write your own scanner**
Front end

- **Parser:**
  - Analyzes the phrase structure of the program
  - Adds hierarchical (tree) structure to the flat stream of tokens
  - Construct IR
  - Produce meaningful error messages
  - Attempt error correction
- There are parser generators like YACC and JavaCC which automate much of the work

Source code → Scanner → tokens → Parser → IR → errors
Context free grammars are used to represent programming language syntaxes:

\[
\begin{align*}
\text{<expr>} &::= \text{<expr>} \text{ <op> <term>} \\
& \quad | \text{ <term>} \\
\text{<term>} &::= \text{ <number>} | \text{ <id>} \\
\text{<op>} &::= + | -
\end{align*}
\]
Front end

- A parser tries to map a program to the syntactic elements defined in the grammar
- A parse can be represented by a tree called a parse or syntax tree
Front end

- A parse tree can be represented more compactly referred to as Abstract Syntax Tree (AST)
- AST is often used as IR between front end and back end
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
Back end

- Produce compact fast code
- Use available addressing modes
Back end

- Have a value in a register when used
- Limited resources
- Optimal allocation is difficult
Traditional three pass compiler

- Code improvement analyzes and change IR
- Goal is to reduce runtime
Middle end (optimizer)

- Modern optimizers are usually built as a set of passes
- Typical passes
  - Constant propagation
  - Common sub-expression elimination
  - Redundant store elimination
  - Dead code elimination
Example modules
# Example modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lex</td>
<td>Break source file into individual words, or <em>tokens</em></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 <em>abstract syntax tree</em> 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 <em>intermediate representation trees</em> (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 <em>control flow graph</em> 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., <em>liveness analysis</em> calculates places where each variable holds a still-needed (<em>live</em>) value</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Choose registers for variables and temporary values; variables not simultaneously live can share the same register</td>
</tr>
<tr>
<td>Code Emission</td>
<td>Replace temporary names in each machine instruction with registers</td>
</tr>
</tbody>
</table>
**Straight line programming (SLP) language (no loops or conditions)**

\[
\begin{align*}
Stm & \rightarrow Stm ; Stm & \text{CompoundStmt} \\
Stm & \rightarrow \text{id} := Exp & \text{AssignStmt} \\
Stm & \rightarrow \text{print ( ExpList )} & \text{PrintStmt} \\
Exp & \rightarrow \text{id} & \text{IdExp} \\
Exp & \rightarrow \text{num} & \text{NumExp} \\
Exp & \rightarrow Exp Binop Exp & \text{OpExp} \\
Exp & \rightarrow ( Stm , Exp ) & \text{EseqExp} \\
ExpList & \rightarrow Exp , ExpList & \text{PairExpList} \\
ExpList & \rightarrow Exp & \text{LastExpList} \\
Binop & \rightarrow + & \text{Plus} \\
Binop & \rightarrow - & \text{Minus} \\
Binop & \rightarrow \times & \text{Times} \\
Binop & \rightarrow / & \text{Div}
\end{align*}
\]

- For example, \( a:=5+3; b:=(\text{print}(a, a-1), 10*a); \text{print}(b) \)
- Prints:
  - 87
  - 80
Java classes for SLP

abstract class Stm {}
class CompoundStmt extends Stm
    Stm stm1, stm2;
    CompoundStmt(Stm s1, Stm s2)
    { stm1=s1; stm2=s2; }
class AssignStmt extends Stm
    { String id; Exp exp;
    AssignStmt(String i, Exp e)
    { id=i; exp=e; }
}
class PrintStmt extends Stm {
    ExpList exps;
    PrintStmt(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;}
}
Object representation

// print(34)
Stm stm =
    new PrintStm(new LastExpList(new NumExp(34)));

// a:=5+3; b:=(print(a, a-1),10*a); print(b)
Stm prog =
    new CompoundStm(
        new AssignStm("a",
            new OpExp(new NumExp(5),
                OpExp.Plus, new NumExp(3))),
        new CompoundStm(
            new AssignStm("b",
                new EseqExp(new PrintStm(new PairExpList(
                    new IdExp("a"),
                    new LastExpList(new OpExp(new IdExp("a"),
                        OpExp.Minus, new NumExp(1)))),
                    new OpExp(new NumExp(10), OpExp.Times,
                        new IdExp("a")))),
                new PrintStm(new LastExpList(new IdExp("b"))))));
Tree representation

- \( a := 5 + 3; \quad b := (\text{print}(a, a - 1), 10 \times a); \quad \text{print}(b) \)
/**
 * Program.java
 *
 * A program to be interpreted. Abstract Syntax Tree classes are
defined in Straightline.java, and the interpreter is in
 * Interpreter.java. See Appel chapter 1 for details.
 */

public class Program {

    // a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )
    Stm prog =
        new CompoundStm(new AssignStm("a",
            new OpExp(new NumExp(5),
                OpExp.Plus, new NumExp(3))),
        new CompoundStm(new AssignStm("b",
            new EseqExp(new PrintStm(new PairExpList(new IdExp("a")),
                new LastExpList(new OpExp(new IdExp("a"),
                    OpExp.Minus, new NumExp(1)))),
            new OpExp(new NumExp(10), OpExp.Times,
                new IdExp("a"))),
        new PrintStm(new LastExpList(new IdExp("b")))));

    // Program
import java.util.Hashtable;

public class Interpreter {
    private boolean debug = true;

    java.util.Hashtable table = new java.util.Hashtable();

    public static void main(String[] args) {
        Interpreter interpreter = new Interpreter();
        System.out.println("Evaluating program.");
        interpreter.interpret(new Program().prog);
    }

    int interpret(Stm stm) {
        fDebug("interpret(Stm)");
        if (stm instanceof CompoundStm)
            return this.interpret((CompoundStm) stm);
        else if (stm instanceof AssignStm)
            return this.interpret((AssignStm) stm);
        else if (stm instanceof PrintStm)
            return this.interpret((PrintStm) stm);
        else {
            System.err.println("ERROR: Unknown Stm concrete class - " + stm);
            return -1;
        }
    }
An Interpreter for SLP

```java
int interpret(CompoundStmt sm) {
    fDebug("interpret(CompoundStmt) ");
    interpret(sm.stml);
    interpret(sm.stm2);
    return 0;
}

int interpret(AssignStmt sm) {
    fDebug("interpret(AssignStmt) ");
    table.put(sm.id, new Integer(interpret(sm.exp)));
    return 0;
}

int interpret(PrintStmt sm) {
    fDebug("interpret(PrintStmt) ");
    ExpList exp = sm.exps;
    while (exp instanceof PairExpList) {
        System.out.println(this.interpret(((PairExpList) sm.exps).head));
        exp = ((PairExpList) exp).tail;
    }
    System.out.println(this.interpret(exp));
    return 0;
}
```
An Interpreter for SLP

```java
int interpret(Exp exp) {
    fDebug("interpret(Exp) ");
    if (exp instanceof NumExp)
        return this.interpret((NumExp) exp);

    if (exp instanceof IdExp)
        return this.interpret((IdExp) exp);

    if (exp instanceof OpExp)
        return this.interpret((OpExp) exp);

    if (exp instanceof EseqExp)
        return this.interpret((EseqExp) exp);

    return 0;
}

int interpret(IdExp exp) {
    fDebug("interpret(IdExp) ");
    return (((Integer) table.get(exp.id)).intValue());
}

int interpret(NumExp exp) {
    fDebug("interpret(NumExp) ");
    return exp.num;
}
```
An Interpreter for SLP

```cpp
int interpret(OpExp exp) {
  fDebug("interpret(OpExp)"陛下，
  Exp e1 = exp.left;
  Exp e2 = exp.right;
  switch(exp.oper) {
    case 1: return interpret(e1) + interpret(e2));
    case 2: return interpret(e1) - interpret(e2));
    case 3: return interpret(e1) * interpret(e2));
    case 4: return interpret(e1) / interpret(e2));
  }
  return 0;
}

int interpret(EseqExp exp) {
  fDebug("interpret(EseqExp)"陛下，
  this.interpret(exp.stm);
  return this.interpret(exp.exp);
}

int interpret(ExpList list) {
  fDebug("interpret(ExpList)"陛下，
  if (list instanceof PairExpList)
    return this.interpret((PairExpList)list);
  return this.interpret((LastExpList)list);`
An Interpreter for SLP

```java
int interpret(PairExpList list) {
    fDebug("interpret(PairExpList) ");
    interpret((Exp)list.head);
    interpret((ExpList)list.tail);
    return 0;
}

int interpret(LastExpList list) {
    fDebug("interpret(LastExpList) ");
    return this.interpret(list.head);
}

void fDebug(String msg) {
    if (debug) System.err.println("DEBUG: " + msg);
}
```