Compiler Theory

001 - Introduction and Course Outline
Sandro Spina
Department of Computer Science
Books *(needed during this course)*

- My slides are based on the three books:
  - Compilers: Principles, techniques and tools.
    - Aho, Lam, Sethi, Ullman
    - Keith Cooper and Linda Torczon
  - Modern compiler implementation in Java.
    - Andrew Appel

- but there are other very good books on compiler theory

- As always, the Internet is another great source of information ...
Course Sequence

- Compiling Theory ... (with practice)
  - Front end
  - Back end

- Assignment – You shall be building parts of a compiler (it is very important that you have good programming skills)
Computing

- Computers are everywhere ... not just in server rooms and offices !!!

- On automobiles, telephones (mobiles), televisions, musical instruments, traffic lights, etc.

- The software that runs on them provides services such as communication, security, entertainment (eg gaming), amongst others.

- Fundamentally everything is based on the theory of computation.
Heterogeneous Computing

- Computing hardware has evolved along many directions.

- Today we speak about multi-core micro processors, traditional CPUs, FPGAs (reconfigurable hardware), GPUs.

- Application software can either be
  - Control intensive (eg searching, parsing)
  - Data intensive (eg image processing, data mining)
  - A mix of both !! (most often)

- Nowadays software needs to execute across a range of hardware devices.
What is a compiler?

- It is a language processor !!

- It is a program that can read a program in one language (Java, C, Lisp, C#, Pascal, C for CUDA, OpenCL, GLSL, HLSL, etc.) – the source language – and translate it into an equivalent program in another language – target language.

- Also - a compiler needs to report any errors in the source program that it detects during the translation process. (e.g. a missing semicolon at the end of a statement)
Brief compiler history (based on Cooper)

- Appeared first in the 1950s for FORTRAN.

- Early 1960s – Algol 60

- Complex machine instruction sets were created in order to mimic closely programming language features. CISC architectures.

- RISC architectures (1980s) shifted this balance towards the compiler.

- Both architectures are still widely used today ... E.g. Ipad is based on RISC and Intel’s x86 on CISC.
Compiler Fundamental Principle (Cooper)

THE COMPILER MUST PRESERVE THE MEANING OF THE PROGRAM BEING COMPILED. CORRECTNESS MUST BE GUARANTEED (PROVED)
What is an interpreter?

- An interpreter is also a language processor

- However - instead of producing a target program, it directly executes the operations specified in the source program on inputs supplied by the user.

- A typical example of an interpreter is the JVM.
  - Java compiler translates Java source code into byte code ... which is then interpreted by a virtual machine (JIT compilers which can during runtime compile to machine code directly)
Clearly a compiler **must** understand the syntax (valid forms) and semantics (meaning) of the input language.

It also needs to understand the rules, syntax and meaning of the output language.
Stages of a compiler (in brief) …

- Source Program (Character Stream)
  - Lexical Analysis

- Tokens (Token Stream)
  - Parser, Syntax Analyzer

- Abstract Syntax (Syntax Tree)
  - Semantic Analysis – Symbol Table
Stages of a compiler (in brief) …

- Syntax Tree
  - Intermediate Code Generator

- Intermediate Representation
  - Machine independent Optimisations

- Intermediate Representation
  - Code Generator > Target Machine Code
Eg from Aho showing compiler phases

- IMAGE FROM AHO Pg 5/7

- Expression (in source code)
  - $position = initial + rate \times 60$
The 1st phase of the compiler is lexical analysis (scanning)

It reads a stream of characters (source program) and groups the characters into meaningful sequences called lexemes.

For each lexeme, the lexical analyzer produces output tokens of the form
- < token-name, attribute-value>
- Eg <id,1> <=> <id,2> <+> <id,3> <*> <60>

id indicates to the syntax analysis phase that we have an identifier (variable) and it’s value is located in symbol table at position 1.
Syntax Analysis (parsing)

- The parser uses the first component of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

- Syntax trees are usually used where each interior node represents an operation and the children of the node represent the arguments of the operation.
Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

- **Type Checking** !!

- Type conversions (coercions) ... eg if operator is applied to int and float, the compiler may convert (coerce) the integer into a floating-point.
Intermediate Code Generation

- Many compilers generate an explicit low-level (machine-like) intermediate representation.

- Properties
  - Easy to produce
  - Easy to translate into target machine

- Eg. Three-address code (linear representation)
  - \(< X = y \textbf{op} z >\) format
  - \(T2 = id3 \times t1\)
  - \(T3 = id2 + t2\)

- We’ll see how this step is useful for code optimization
This stage of compilation attempts to improve the intermediate code so that better target code is generated.

Better = faster, smaller !!

But could also mean that target consumes less power (eg mobile phone, pda apps)

We shall eventually look at machine-dependent and machine-independent optimizations in some detail.
Compiler optimization must meet the following design objectives:

- It must be correct, that is, preserve the meaning of the compiled program.
- It must improve the performance of many programs (not just a few !!)
- Compilation time must be kept reasonable, and
- Engineering effort required must be manageable
The code generator takes as input an intermediate representation of the source program and maps it into the target language.

If target language is machine code, registers or memory locations are selected for each of the variables ... the assignment of registers to hold variables is very important!

Output might look like:

- LDF R2, id3
- MULF R2, R2, #60.0 (# means 60.0 is a constant)
- LDF R1, id2
- ADDF R1, R1, R2
Instruction Scheduling

- An important task during code generation is instruction scheduling.

- This is because the execution time of different operations can vary according to the target machine’s specific performance constraints.

- Clearly, an operation cannot begin to execute until its operands are ready ... But can start multiple new operations which do not need the result of currently executing operations.

- The intuition here is that by re-ordering certain operations, the compiler minimizes the number of cycles wasted waiting for operands ...

- E.g. \([\text{load } r1, \text{add } r1, \text{load } r2, \text{mult } r2, \text{load } r3, \text{add } r3]\) can be reordered to \([\text{load } r1, \text{load } r2, \text{load } r3, \text{add } r1, \text{mult } r2, \text{add } r3]\)
Some important concepts (i)

- So far we have seen (very very briefly) the phases of a compiler !!

- In the next few slides we shall look at important concepts you should be aware of with respect to programming languages + compilers
Some important concepts (ii) – Symbol Table

- Compilers need to record the variable names used in the source program together with various attributes of each name (e.g. type, storage allocated, scope).

- Procedure names are also stored together with attributes such as number of parameters, type of arguments and method of passing each argument (by value or by reference).

- It is ‘clearly’ very important that the data structure which stores this information is efficient in terms of retrieval and storage of data.
Concepts (iii) – Parameter Passing

- All programming languages have a notion of a procedure (in Java we have methods, functions in C)

- Most programming languages use either call by value, call by reference or both

- Call by Value: - the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). Both C and Java use call by value however with C we can pass a pointer to a variable and with Java many variables are really references (pointers) to arrays, strings and objects !!
All programming languages have a notion of a procedure (in Java we have methods, functions in C)

Call by reference is usually an option in many programming languages (call by value is much more used)

Call by Reference :- Changes to the formal parameter appear as changes in the actual parameter !! Eg Pascal incr ( var x : int )
A memory hierarchy consists of several levels of storage with different speeds and sizes, with the levels closest to the processor (registers, caches) being the fastest but smallest.

A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes and beyond.

Using registers effectively is probably the single most important problem in optimizing a program.

The compiler can improve the effectiveness of the memory hierarchy by changing the layout of the data.
Concepts (vi) – Software Tools

- **Lexical Analysis** – LeX, FLeX, JLeX
- **Syntax Analysis** – JavaCC, SableCC
- **Semantic Analysis** – JavaCC, SableCC
- **MiniJava programming language** (Appel Book)
The language of straight-line programs!

- Check Appel Pg 7
- Made up of simple statements and expressions (no loops or if stmts)
- Bottom up:
  - Binop,
  - Exp,
  - ExpList,
  - Stm.
An embedded compiler!!

- We know what a compiler is.
- Embedding with respect to language ... for example one can easily embed a parser in Haskell or Java.
- A compiler would ‘slightly’ be more hairy to embed because you would need to generate executable at run-time using it.
- We usually think of compile-time and run-time are two separate stages but really one can be compiling at run-time‼️
- Let’s take for example Janino ... (next slide)
ExpressionEvaluator ee = new ExpressionEvaluator(
    "c > d ? c : d", // expression
    int.class, // expressionType
    new String[] { "c", "d" }, // parameterNames
    new Class[] { int.class, int.class } // parameterTypes
);

// Evaluate it with varying parameter values; very fast.
Integer res = (Integer) ee.evaluate(
    new Object[] { // parameterValues
        new Integer(10),
        new Integer(11),
    }
);

System.out.println("res = " + res);
Janino embedded in Sunflow (GI Rendering Engine)

- Janino is used in Sunflow in order to compile shaders at run-time.

- Different shaders describing different reflection models can be loaded at run-time ... then compiled using Janino ... and executed in the same memory space of the currently executing VM.

- DirectX and OpenGL also have compilers which compile at run-time vertex and pixel shaders before loading them on the GPU.

- The point to understand here is that compilers have an extremely vast range of applicability. Not just the traditional source -> compile -> execute trace.

- Although we’ll be looking mainly at those because the main concepts remain the same.
Compiler Construction Topics

- **Compiler construction is a complex task!!!** It combines together many aspects of Computer Science
  - Formal language theory
  - Artificial intelligence (greedy algorithms and heuristic techniques)
  - Computer architecture

- We shall cover many aspects, but will focus mostly on the front end.

- At the end of this course you will be able to take a formal specification of a language (using e.g. EBNF) and be able to use tools (compiler compilers) to generate a source-to-source compiler.
Next ...

- Chapter 2 from Dragon Book

- A simple syntax-directed translator will be used to briefly introduce
  - Lexical Analysis
  - Syntax Analysis
  - Parsing
  - Intermediate Code