Compiler Design

Lec1:

Compilers:
source->compiler->program
input->program->output

Interpreters: Python
program+input->interpreter->output

C,C++: compiled
Java: Javc->Java Bytecode->JVM->Machine code via JiT
JavaScript:blur

C/C++:
preproc compile linking
source1->pre-source1->object1.o->| f( ) call (0xadr)
source2->pre-source2->object2.o->| int f( )

Compiler passes/phases

“Front End”(language specific)
1.Lexical analysis: chunk of codes meaningful for compiler
2.Parsing: use grammar to create a parse tree
3.Semantic analysis: type checking and operations doing on them make sense
4.Intermediate Language Generation (internal language for compiler)

“Back End”(Universal for languages and aimed for different architectures)
5.Code generation(generate machine code from I.L)
6.Optimisation on machine code

Clang/Clang++ : C->LLVMIR LLI
LLVM->LLVMIR

LLVM: The **LLVM** compiler infrastructure project is a "collection of modular and reusable compiler and toolchain technologies” used to develop compiler front ends and back ends.

LLVMIR: LLVM can provide the middle layers of a complete compiler system, taking intermediate representation (IR) code from a compiler and emitting an optimised IR. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform. LLVM can accept the IR from the GNU Compiler Collection(GCC) toolchain, allowing it to be used with a wide array of extant compilers written for that project.

LLVM can also generate relocatable machine code at compile-time or link-time or even binary machine code at run-time.

IR: An **Intermediate representation** (**IR**) is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation.[1] A "good" IR must be *accurate* – capable of representing the source code without loss of information[2] – and *independent* of any particular source or target language.[1] An IR may take one of several forms: an in-memory data structure, or a special tuple- or stack-based code readable by the program.[3] In the latter case it is also called an *intermediate language*.

Lexical analysis
position = initial + rate * 60 ;
<id1> <=> <id2> <+> <id3> <*> <60>;

Symbol Table
position: id1 float
initial: id2 float
rate: id3 float

identifier: [a-zA-Z][a-zA-Z0-9]+
assign: identifier = expr
expression: identifier | expr binop expr
binop: +|-|*|/|^|~|

Parsing

```
          =
         / \
      id1   +
           / \
         id2  *
             / \
           id3 60 /// (int_to_float(60)
```

reverse polar(SCALA fold right hhh)

3 address code:
t1 = int_to_float(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
=>
t1 = id3 * 60.0
id1 = id2 + t1

Code Generation:

```
    dest, src
LDF R2, id3                0x03 0x02 0x10…./
      dest, src, operand
MULTF R2, R2, 60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1,R1
```

Register Allocation:
When less available register than CPU actually have

Compiler Construction Tools
- Lexing: Lex
- Parsing: Yacc CFG->parser (parser generator)
- Dataflow Analysis Engines:
- TableGen:

Finite State machines -> regular expr (anything written in regular expressions -> finite closed form instructions without stack)
Context-free grammars-> push-down automaton (finite state machine which has a stack) (can solve balancing parentheses)
Trees-> Graphs

High Level Languages
Can be optimised in many different ways
Access ram use more energy move between registers

Optimisation
- Correct
- Compose
- Decidability
- Hotspot, dynamic optimisation

Parallelism
Thread-level parallelism
ILP
- Single Instruction Multiple Data(SIMD) MMX/SSE(intel) Neon(arm)
- VLIW(Very Long Instruction Word) (Itanium, Elbrus)

Memory Hierarchy
Register (go down on this list, it gets slower and larger)
 |L1 L1-I, L1-D
 |L2
 |L3
 |RAM
 |Disk
V
Prefetcher

Binary Translators
- Transmeta
- QEMU
  ARM->TCG(tiny code generator)->x86
- Apple PowerPC->Intel (can use app for powerpc mac on intel mac)

Bug finding
- Linters
- Static analysers
- Fuzzers

CodeEditing Tools
- Code completion
- Error messages

Concept recap of PL for building compilers
Static: anything can be known before running the program
Dynamic

Environment and state   x.y name (x, y identifiers)
- Names: refer to specific locations
- Identifier: name for an entity
- Variables: underlying location where data is stored
names—environment->variables—state-> values

Parameter passing:
f (intx, int y) formal parameters
f (10,35)   actual parameters

<u>Call by Value</u>
```
f (int x) {
  x=x+1
}
z=10
f(z)
z is still 10
```

Call by Reference

```
f(int x){
 x=x+1
}
z=10
f(z)
//z=11
```

Call by Name

deliver expr itself

in functional programming Scala

```
f(int x){
 x=x+1
}
f(x+y) = x+y+1
```