Learning Goals

- Understand the internal structure of a real-world compiler
- Understand the effectiveness and limitations of code analysis and optimization techniques
- Be able to construct a full compiler for a toy language, generating assembly code for a RISC architecture

There are less than 1.5 compiler construction expert for every *1000* software engineers, but almost 2 jobs in compilers for every 100 in software engineering! (Data from LinkedIn, collected in 2012)

Where to find me.
Room & Time of lectures: we can change them if needed.

Introduction to Compiler Construction

- Why compiling? Compilers (performance, avoid writing fundamental code in asm) vs interpreters (ease of deployment/portability). Performance in a cost model, performance as a proxy of power consumption.
- When to compile? JIT, AOT and static compilers
- What to compile? Compilation units: from narrow (single-line or so) to expression to function-scope to interprocedural to whole-program. Speed/resources vs optimization.
- Where to compile? Cross-compilation and split compilation; re-hosting vs re-targeting
- Issues: maintainance (debugging), portability, retargetability, support for multiple languages
- Overview of a compiler framework
    - Lexical analysis & parsing (review): LL vs LR.
        - LR: declarative style, express rules in a DSL
        - LL: programming style, express rules as procedures
    - Statement and Data Structure Lowering
        - Lowering of loops: for → while → goto
    - Optimization: machine independent and machine-dependent, language dependent vs language independent
    - Code Generation
- *Reading*: Compiler Construction
- Tools (C compilers):
    - GCC
    - Clang/LLVM
    - Open64/ORC, Rose (source to source)
    - icc, CoSy

Topics for the next lectures

Intermediate Representations
Semantic Analysis & Type Checking
Code Generation
Dataflow Optimization
Register Allocation
Parallelization and other optimization techniques
Introduction to LLVM

Projects with LLVM
Python Bindings + tests
Register Allocation in SSA
Identify pure functions
Simple alias analysis pass
Profile guided transformations
Improve OpenMP support
Partial redundancy elimination
Loop unrolling on machine code

```
def factor(symtab) :
      if accept('ident') : return Var(var=symtab.find(value), symtab=symtab)
      if accept('number') : return Const(value=value, symtab=symtab)
      elif accept('lparen') :
            expr = expression()
            expect('rparen')
            return expr
      else :
            error("factor: syntax error")
            getsym()

def term(symtab) :
      op=None
      expr = factor(symtab)
      while new_sym in [ 'times', 'slash'] :
            getsym()
            op = sym
            expr2 = factor(symtab)
            expr = BinExpr(children=[ op, expr, expr2 ], symtab=symtab)
      return expr

def expression(symtab) :
      op=None
      if new_sym in [ 'plus' or 'minus' ] :
            getsym()
            op = sym
      expr = term(symtab)
      if op : expr = UnExpr(children=[initial_op, expr], symtab=symtab)
      while new_sym in [ 'plus' or 'minus' ] :
            getsym()
            op = sym
            expr2 = term(symtab)
            expr = BinExpr(children=[ op, expr, expr2 ], symtab=symtab)
      return expr
```