Review and Preview

Lecture 12
Further Topics in Compilers

- **Advanced Language features**
  - Object-Oriented Languages
    - objects, classes
  - Functional Languages
    - function closures
    - lazy evaluation
  - advanced type systems
    - parametric polymorphism (ML, Haskell)
    - subtyping (O-O languages)
  - modules
  - continuations, coroutines, threads
Further Topics in Compilers

- data flow and control flow analysis
- constant folding
- inlining functions
- loop optimization
  - hoisting code out of loops, loop unrolling
- instruction selection
- instruction scheduling
- array bounds checks
- delay slots, speculative execution
- instruction level parallelism (ILP), pipelining, function units
Intermediate Languages

- Static Single-Assignment form
  - IR as functional language
- Continuation-Passing Style (CPS)
  - A-normal form
  - making important structure explicit
- Typed intermediate languages
  - FLINT, TILT, TAL, ...
- Virtual machine systems (JVM, MSIL)
  - JIT compilers
Runtime Systems

- **Memory management**
  - garbage collectors
    - mark-sweep (classical)
    - copying, generational, incremental, compacting
- **Traps and interrupts**
- **Input/Output**
- **OS services**
- **Threads**
Review

- **Lexical Analysis**
  - turning characters into tokens
  - regular expressions
  - NFAs and DFAs
  - `lex (ml-lex)`
    - r.e. patterns, actions
    - start states
    - handling strings with escapes
    - handling nested comments
Parsing

- Context-free grammars
- Top-down parsers
  - recursive descent
- Bottom-up parsers
  - LR grammars (LR(k), SLR, LALR(k))
  - constructing parser tables
- Yacc (ml-yacc)
  - terminals and nonterminals
  - grammar rules and actions
  - shift-reduce and reduce-reduce conflicts
  - precedence and associativity declarations
Abstract Syntax

- Simple tree representation of logical structure
  - types
  - expressions
  - statements
  - declarations
- Naturally expressed using ML datatypes
- Basis for semantic (or static) analysis
Type Checking

- **Types**
  - express structure of data
  - interface of functions

- **Type Environments**
  - map names (variables, functions) to their types

- **Typing rules**
  - relate expressions and types

- **Type checking**
  - check consistency
  - synthesize types of expressions
Semantic Analysis

- Type checking
- Determining scope of names (types, variables, functions)
- Escape analysis
  - global vs local variables
- Forward references
  - recursive types
  - recursive functions
  - two pass analysis (prebind names, then analyze definitions)
Intermediate Representation

- **Intermediate language or IR (tree.sig/sml)**
  - a lower-level tree representation of program structure
    - constructs similar to machine language
  - unlimited supply of temps, or abstract registers
  - temps and labels replace variables and functions
  - conditional and unconditional jumps express control flow constructs (if-then-else, while, for, break)
  - type \textit{lexp} expresses memory accesses (l-values, r-values)
  - MOVE represents assignment, indexing, selection
Translation of Absyn to IR

- *translation environments*
  - maps names (variables and functions) to access info

- *separate expressions and statements*
  - gexp reunifies expressions, statements, and conditionals
  - coercions between different forms to satisfy context

- *recursive traversal of abstract syntax (similar to type checking, escape analysis)*
  - two pass treatment of recursive function declarations
  - types not involved

- *units of translation are “fragments”*
  - representing single function body (or top-level program)
Function Call Frames

• **Call Frames (aka Activation Records)**
  
  • store local information associated with a function call
    
    • arguments and local variables that “escape”
    
    • saved $fp and $ra registers
    
    • space for spilled temps and callee saves registers
    
    • space for excess outgoing arguments (beyond first 4)

• **frame record**
  
  • records information about function and its frame layout during compilation
  
  • manages allocation of slots for arguments, locals, spills
  
  • could store info on use of global variables and need for static link
Static Links

• Need to compute access to nonlocal variables

• static link is frame pointer of frame of statically enclosing function

• passed to function as additional, implicit parameter
  • not always needed
  • first argument, treated as escaping (found in 0($fp))

• computed with the aid of “level” type
  • chain of statically nested functions
  • translation environment maps function to its parent’s level
Basic Blocks & Trace Scheduling

- **Linearize code**
  - move statements (including calls) out of expressions
    - no side effects in expressions
  - flatten to statement list (possibly followed by final expression)
    - eliminates SEQ and ESEQ IR tree forms

- **Split into basic blocks**
  - straight-line code segments
  - start with label, end with (conditional or unconditional) jump
  - can be reordered without changing behavior

- **Trace scheduling**
  - sort basic blocks and concatenate them
  - arrange so jumps are followed by target labels when possible
  - arrange for false branch of cond. jump to follow jump
Liveness Analysis

- **live range**
  - a temp is live over a sequence of instructions between a definition and a use of that temp

- **liveness analysis determines the live ranges of temps**
  - calculates live-in and live-out sets of temps at each instruction

- **two temps interfere if they are both live at the same point**
  - they then have to coexist, and therefore can’t occupy the same register
  - t1 and t2 interfere if t1 is defined at an instr and the other is live-out at that instr

- **interference graph records interference relation**
  - nodes represent temps, edges represent interference
Register Allocation

- Assign registers (strings) to temps (allocation)
- Avoid assigning same register to temps that interfere
  - color the interference graph using registers as colors
    - successively remove nodes of insignificant degree (*simplify*)
    - color them as they are restored
  - pick colors to maximize number of moves between temps of same color (these moves can then be eliminated)
- If no nodes of insignificant degree
  - choose a node of minimal *spill cost* to *spill* (store in frame)
  - rewrite code to accomplish spilling
  - redo liveness analysis and interference graph coloring

- Register coallescing
  - attempt ot coallesc nodes that are move related, if it doesn’t make coloring harder
Final Exam

- Open book, open notes, open code
- Wednesday, June 11, 10:30am-12:30pm