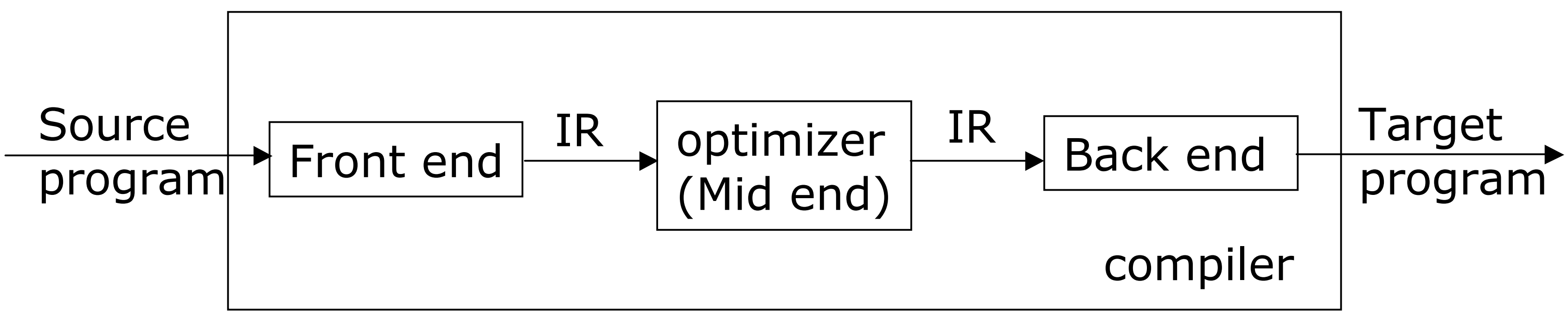
CS5363 Final Review

Programming language implementation

- Programming languages
  - Tools for describing data and algorithms
    - Instructing machines what to do
    - Communicate between computers and programmers
  - Different programming languages
    - FORTRAN, Pascal, C, C++, Java, Lisp, Scheme, ML, …
- Compilers/translators
  - Translate programming languages to machine languages
  - Translate one programming language to another
- Interpreters
  - Interpret the meaning of programs and perform the operations accordingly

Objectives of compilers

- Fundamental principles
  - Compilers shall preserve the meaning of the input program --- it must be correct
    - Translation should not alter the original meaning
  - Compilers shall do something of value
    - Optimize the performance of the input application



Front end

- Source program

  for (w = 1; w < 100; w = w * 2);

- Input: a stream of characters
  - ‘f’ ‘o’ ‘r’ ‘(’ `w’ ‘=’ ‘1’ ‘;’ ‘w’ ‘<’ ‘1’ ‘0’ ‘0’ ‘;’ ‘w’…
- Scanning--- convert input to a stream of words (tokens)
  - “for” “(“ “w” “=“ “1” “;” “w” “<“ “100” “;” “w”…
- Parsing---discover the syntax/structure of sentences

  forStmt: “for” “(” expr1 “;” expr2 “;” expr3 “)” stmt

  expr1 : localVar(w) “=” integer(1)

  expr2 : localVar(w) “<” integer(100)

  expr3: localVar(w) “=” expr4

  expr4: localVar(w) “*” integer(2)

  stmt: “;”

Lexical analysis/Scanning

- Called by the parser each time a new token is needed
  - Each token has a "type" and an optional "value"
- Regular expression: compact description of composition of tokens
  - Alphabet ∑: the set of characters that make up tokens

  A regular expression over ∑ could be

  the empty string, a symbol $s \in \Sigma$, or

  (α), $\alpha\beta$, α | ß, or α*, where α and ß are regular expressions.
- Finite automata
  - Include an alphabet ∑, a set of states S (including a start state s0 and a set of final states F), and a transition function δ
  - DFA δ: S * ∑ → S; NFA δ: S * ∑ → power(S)
- Regular expressions and finite automata
  - Describing and recognizing an input language
  - From R.E to NFA to DFA
  - Examples: comments, identifiers, integers, floating point numbers, ……

Context-free grammar

- Describe how to recursively compose programs/sentences from tokens
  - Loops, statements, expressions, declarations, …….
- A context-free grammar includes (T,NT,S,P)
  - BNF: each production has format A ::= B (or A→B) where a is a single non-terminal; B is a sequence of terminals and non-terminals
  - Using CFG to describe regular expressions
    - n ::= dn | d
    - d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- Given a CFG G=(T,NT,P,S), a sentence s belongs to L(G) if there is a derivation from S to s
- Derivation: top-down replacement of non-terminals
  - Each replacement follows a production rule
  - Left-most vs. right-most derivations
  - Example: derivations for 5 + 15 * 20

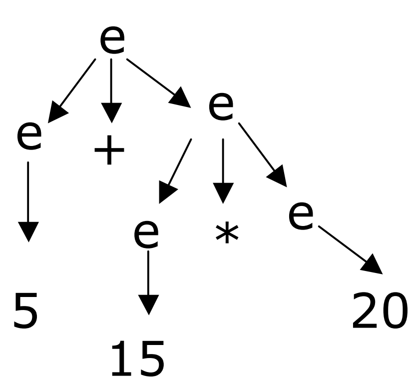
  e=>**e*e**=>**e+e*e**=>**5+e*e**=>**5+15*e=>5+15*20**

  e=>**e+e**=>**5+e**=>**5+e*e**=>5**+15*e**=>**5+15*20**
- Writing grammars for languages
  - E.g., the set of balanced parentheses

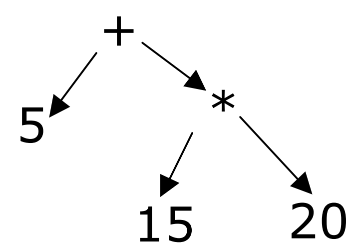
Parse trees and abstract syntax trees

- Parse tree: graphical representation of derivations
  - Parent: left-hand of production; children: right-hand of production
- A grammar is syntactically ambiguous if
  - some program has multiple parse trees
  - Rewrite an ambiguous grammar: identify source of ambiguity, restrict the applicability of some productions
    - Standard rewrite for defining associativity and precedence of operators
- Abstract syntax tree: condensed form of parse tree
  - Operators and keywords do not appear as leaves
  - Chains of single productions may be collapsed

Parse tree:



Abstract syntax tree:

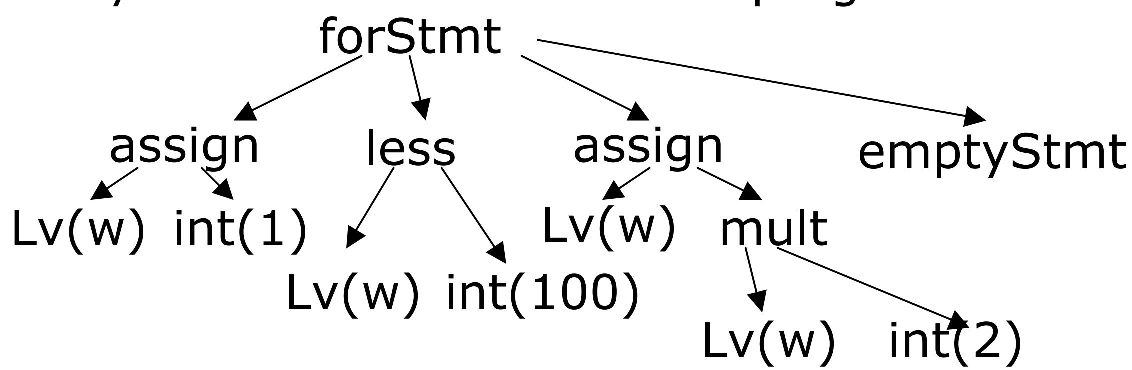


Top-down and bottom-up parsing

- Top-down parsing: start from the starting non-terminal, try to find a left-most derivation
  - Recursive descent parsing and LL(k) predictive parsers
  - Transformation to grammars: eliminate left-recursion and Left-factoring
  - Build LL(1) parsers: compute First for each production and Follow for each non-terminal
- Bottom-up parsing: start from the input string, try to reduce the input string to the starting non-terminal
  - Equivalent to the reverse of a right-most derivation
  - Right-sentential forms and their handles
  - Shift-reduce parsing and LR(k) parsers
    - The meaning of LR(1) items; building DFA for handle pruning; canonical LR(1) collection
    - How to build LR(1) parse table and how to interpret LR(1) table
- Top-down vs. bottom-up parsers: which is better?

Intermediate representation

- Source program
  for (w = 1; w < 100; w = w * 2);
- Parsing --- convert input tokens to IR
  - Abstract syntax tree --- structure of program



- Context sensitive analysis --- the surrounding environment
  - Symbol table: information about symbols
    - V: local variable, has type “int”, allocated to register
  - At least one symbol table for each scope

Context-sensitive analysis

- Attribute grammar (syntax-directed definition)
  - Associate a collection of attributes with each grammar symbol
  - Define actions to evaluate attribute values during parsing
- Synthesized and inherited attribute
  - Dependences in attribute evaluation
  - Annotated parse tree and attribute dependence graph
  - Bottom-up parsing and L-attribute evaluation
  - Translation scheme: define attribute evaluation within the parsing of grammar symbols
- Type checking
  - Basic types and compound types
  - Types of variables and expressions
    - Type environment (symbol table)
  - Type system, type checking and type conversion
    - Compile-time vs. runtime type checking
    - Type checking and type inference

Variation of IR

- IR: intermediate language between source and target
  - Source-level IR vs. machine-level IR
  - Graphical IR vs. linear IR
  - Mapping names/storages to variables
- Translating from source language to IR --- syntax-directed translation
- IR for the purpose of program analysis
  - Control-flow graph
  - Dependence graph
  - Static single assignment (SSA)

Execution model of programs

- Procedural abstraction: scope and storage management
  - Nested blocks and namespaces
  - Scoping rules
    - static/lexical vs. dynamic scoping
    - Local vs. global variables
  - Parameter passing: pass-by-value vs pass-by-reference
  - Activation record for blocks and functions: what are the necessary fields?
- The simplified memory model
  - Runtime stack, heap and code space
    - program pointer and activation record pointer
  - Allocating activation records on stack
    - how to set up the activation record?
  - Allocating variables in memory
    - base address and offset; local vs. static/global variables
    - Coordinates of variables: nesting level of variable scope
  - Access link and global display

Mid end --- improving the code

Original code

```
int j = 0, k;
while (j < 500) {
    j = j + 1;
    k = j * 8;
    a[k] = 0;
}
```

Improved code

```
int k = 0;
while (k < 4000) {
    k = k + 8;
    a[k] = 0;
}
```

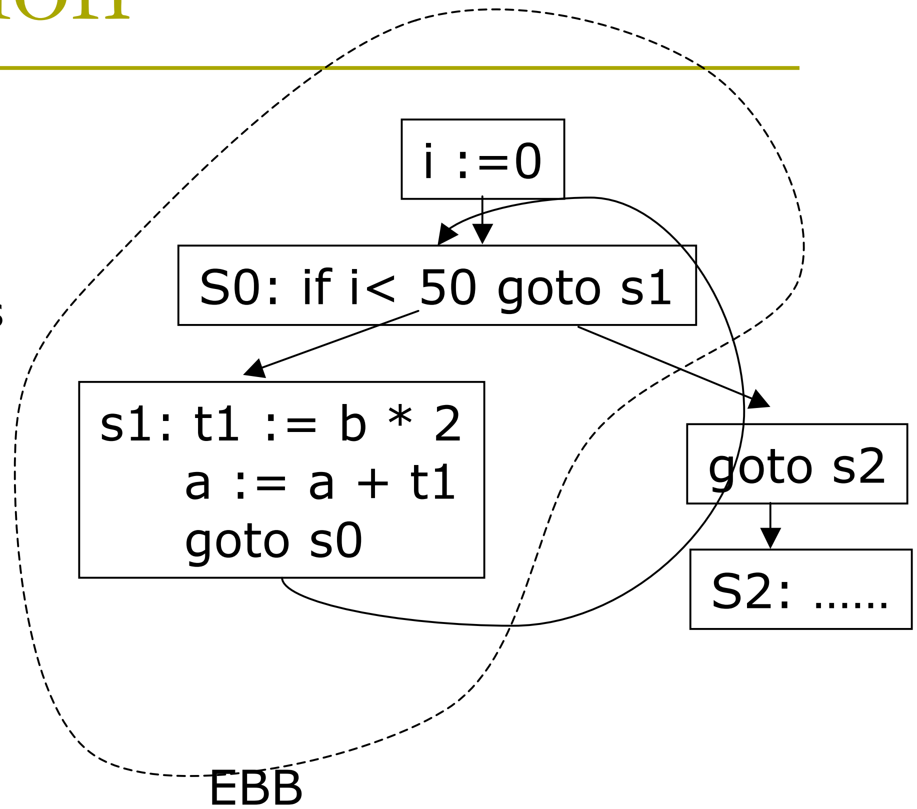
- Program analysis --- recognize optimization opportunities
  - Data flow analysis: where data are defined and used
  - Dependence analysis: when operations can be reordered
- Transformations --- improve target program speed or space
  - Redundancy elimination
  - Improve data movement and instruction parallelization

Data-flow analysis

- Program analysis: statically examines input computation to ensure safety and profitability of optimizations
- Data-flow analysis: reason about flow of values on control-flow graph
  - Forward vs. backward flow problem
    - Define domain of analysis; build the control-flow graph
    - Define a set of data-flow equations at each basic block
    - Evaluate local data-flow sets at each basic block
    - Iteratively modify result at each basic block until reaching a fixed point
  - Traversal order of basic blocks: (reverse) postorder
  - Example: available expression analysis, live variable analysis, reaching definition analysis, dominator analysis
- SSA (static single assignment)
  - Two rules that must be satisfied
  - Insertion of ∅ functions; rewrite from SSA to normal code
  - Computing dominance relations and dominance frontiers

Scope of optimization

- Local methods
  - Applicable only to basic blocks
- Superlocal methods
  - Operate on extended basic blocks (EBB)
  - B1,B2,B3,…,Bm, where Bi is the single predecessor of B(i+1)
- Regional methods
  - Operate beyond EBBs, e.g. loops, conditionals
- Global (intraprocedural) methods
  - Operate on entire procedure (subroutine)
- Whole-program (interprocedural) methods
  - Operate on entire program



Program optimizations

- Redundant expression elimination
  - Value numbering
    - Simulate runtime evaluation of instruction sequence
    - Use an integer number to unique identify each runtime value
    - Map each expression to a value number
    - Scope of optimization: local, EBB, dominator based
  - Global redundancy elimination
    - Find available expressions at the entry of each basic block
    - Remove expressions that are redundant
  - Naming of variables change availability of expressions
- Dead code elimination
  - Mark instructions that are necessary to evaluation of program; remove expressions with never-used results
  - Computing control dependence among basic blocks

Back end --- code generation

- Memory management
  - Every variable must be allocated with a memory location
  - Address stored in symbol tables during translation
- Instruction selection
  - Assembly language of the target machine
  - Abstract assembly (three/two address code)
- Register allocation
  - Most instructions must operate on registers
  - Values in registers are faster to access
- Instruction scheduling
  - Reorder instructions to enhance parallelism/pipelining in processors

Example of code generation

Code for w ← w * 2 * x * y * z in ILOC

```
loadAI rarp, @w → rw      // load 'w'
loadI  2        → r2      // constant 2 into r2
loadAI rarp, @x → rx      // load 'x'
loadAI rarp, @y → ry      // load 'y'
loadAI rarp, @z → rz      // load 'z'
mult   rw, r2   → rw      // rw ← w * 2
Mult   rw, rx   → rw      // rw ← w*2*x
Mult   rw, ry   → rw      // rw ← w * 2 * x * y
Mult   rw, rz   → rw      // rw ← w * 2 * x * y * z
storeAI rw      → rarp, @w  // write rw back to 'w'
```

ILOC: Imtermediate language for an optimizing compiler
similar to the assembly language for a simple RISC machine

Machine code generation

- Assigning storage: register or memory
  - Every expression e must have
    - A type that determines the size/meaning of its value
    - A location to store its value (e.place)
  - A variable may require a permanent storage
    - Non-local variables or variables that might be aliased
- Translating to three-address code
  - Different code shapes may have different efficiency
  - Translating expressions
    - Mixed type expressions --- implicit type conversion
    - Arithmetic vs. boolean expressions; short-circuit translation
  - Translating variable access, arrays, and function calls
  - Translating control-flow statements

Register allocation and assignment

- Values in registers are easier and faster to access than memory
  - Reserve a few registers for memory access
  - Efficiently utilize the rest of general-purpose registers
- Register allocation: at each program point, select a set of values to reside in registers
- Register assignment: pick a specific register for each value, subject to hardware constraints
- Register-to-register vs. memory model
- Local register allocation: top-down vs. bottom-up
- Graph-coloring based register allocation
  - Construct global live ranges
  - Build interference graph
  - Coalesce live ranges to eliminate register copying
  - Rank all live ranges based on spilling cost
  - Color the interference graph

Instruction selection

- Table-based instruction selector
  - Create a description of target machine, use back-end generator to produce a pattern-matching table
- AST tiling: pattern-based instruction selection through tree-grammar
  - Bottom-up walk of the AST, for each node n, find all applicable tree patterns and select the one with lowest cost
- Peephole optimization
  - Use a simple scheme to translate IR to machine code
  - Discover local improvements by examining short sequences of adjacent operations: expand → simplify → match

Instruction scheduling

- Dependence/precedence graph G = (N,E)
  - Each node n ∈ N is a single operation
    - type(n) and delay(n)
  - Edge (n1,n2) ∈ N indicates n2 uses result of n1 as operand
    - What about anti-dependences?
  - G is acyclic within each basic block
- Given a dependence graph D = (N,E), a schedule S maps each node n ∈ N to the cycle number that n is issued.
  - Each schedule S must be well-formed, correct, and feasible.
  - Critical path: the longest path in the dependence graph
- List scheduling: greedy heuristic to scheduling operations in a single basic block
  - Build a dependence graph (rename to avoid anti-dependences)
  - Assign priorities to each operation n (the length of longest latency path from n to end)
  - Iteratively select an operation and schedule it