CSE 504

Course Summary

Symbol Tables

- Bindings
- Attributes
- Binding Time
- Scopes
- Visibility
- Lexical scoping
- Implementation of symbol tables
- Static Vs Dynamic scoping

Organization of a Compiler

- Lexical analysis
- Parsing (syntax analysis)
- Abstract Syntax Tree (AST)
- Semantic Analysis (type checking etc.)
- Syntax-directed definitions (attribute grammars)
- Intermediate code generation
- Code optimization
- Final code generation
- Runtime Environment

Lexical Analysis: Foundations

- Token, Lexeme, Pattern, String
- Regular expressions
  - Syntax, semantics
  - Finite-state automata
    - NFA vs DFA
    - Recognition using NFA
    - NFA to DFA translation
    - Optimization of DFAs
  - Properties of regular languages
    - Closed under complementation, union, intersection
  - RE to FSA translation
    - RE → NFA → DFA → optimal DFA
    - Direct construction of DFA

Lexical Analysis

- Goal: convert character stream to token stream
  - Recognize “words” in language
    - Keywords, identifiers, constants (literals), ..
  - Ignore “irrelevant” input
    - White spaces, comments, …
  - Maintain association between lexer output and input
    - Line numbers, column numbers, …
- Flex: A lexical analyzer generator
  - Use of Flex in compilers
  - Use of regular expressions as well as ***start*** states
    - Ability to freely intermix automata-based and RE based specifications of lexical analysis
    - Very powerful capability, makes Flex a very versatile tool for any application requiring efficient recognition of REs

Syntax Analysis: CFGs

- Types of grammars
  - Regular, context-free, context-sensitive, unrestricted
- CFGs
  - Terminals, Nonterminals, Productions, Start symbol
  - Sentence, Sentential form, String
  - Notational conventions
  - L(G)
  - Equivalence of grammars
  - Two sides of grammars: generation and acceptance

CFGs

- Derivations
  - Single-step, multistep
  - Left-most, right-most, canonical
- Parse trees
- Ambiguity
- Disambiguation rules
  - Operator precedence
  - dangling-else and shift/reduce conflict

CFGs (continued)

- Equivalence of grammars (and how to establish this)
- Recognizing grammars
  - Push-down automata (PDA)
  - NPDA Vs DPDA
- Properties
  - Closed under union, but not complementation or intersection
  - Note: CFGs recognizable using DPDAs are closed under all these operations.

Top-Down Parsing

- **Derive sentence from start symbol**
  - Next step in derivation is guided by input
- Predictive Parsing
  - Left-recursion elimination and left-factoring
  - Parsing with back-tracking
  - Recursive descent parsing
- Non-recursive parsing
  - Table-driven
  - Construction of LL(1) parsing tables
    - FIRST and FOLLOW
- LL(1) grammars

Bottom-Up Parsing

- **Reduce sentence to start symbol**
  - Next reduction is guided by PDA stack and input
- Handles
- Shift-Reduce parsing
  - Structure and operation of an SR parser
- Identification of handles
- Viable prefixes

LR Parsing

- Structure and operation of an LR parser
- Action and Goto tables
- LR Vs LL parsing
- Construction of SLR(1) parsing tables
  - Items and Item sets
  - Viable prefixes
  - DFA for recognizing viable prefixes
  - Generation of LR parsing tables from DFA
- LR(1) and LALR(1) parsing

Parser Generators

- Bison/Yacc
  - LALR(1) Parser generator
  - Integrates nicely with Lex/Flex
- Use of Bison to specify a parser
- Conflicts
  - How to interpret them
  - How to fix them
    - Operator precedence
- Bison is a versatile tool
  - Can be used for a variety of language processing applications
- Error recovery

Syntax-Directed Translation

- The concept and its use
- Syntax-directed translation using Bison
- Attribute grammars --- acceptance by AG
- Synthesized Vs inherited attributes
  - Flow of attribute information
- L-attributed definitions
- S-attributed definitions
- Maintaining attributes during parsing
  - Top-down parsing
  - Bottom-up parsing

Semantic Analysis

- Semantic analysis takes place during
  - AST construction
  - Type-checking
  - Intermediate code generation
- ASTs vs Parse trees
- Syntax-directed construction of AST using Bison/C++

Types

- What is a type
- Data types in modern languages
  - Simple types
  - Compound types
    - Products, unions (tagged Vs untagged), arrays, functions, pointers
  - Type expressions
- Polymorphism
  - Parametric polymorphism Vs overloading
  - Code reuse
- Type equivalence
  - Structural Vs Name based Vs declaration based
- Type compatibility
- Type checking Vs type inference
- Type conversions
  - Explicit, implicit, coercion
- Static Vs Dynamic typing
- Strong Vs Weak typing

Type-Checking

- Syntax-directed definitions for type-checking
  - Expressions
  - Assignment
  - Function calls/returns
  - Other statements
- Subtype principle
- Name resolution
  - Overloading resolution
  - Resolution of methods in OO languages
- Type-checker for E--

Expression Evaluation

- Semantics of Expressions
  - Order of evaluation
  - Use of properties of arithmetic operators
  - Problems with side-effects
- Boolean expression evaluation
  - Short-cirtcuit evaluation
- Control-flow statement evaluation
  - Switch-statement
  - While statement
  - For statement

Procedure calls

- Parameter-passing mechanisms
  - Call-by-Value
  - Call-by-Reference
  - Call-by-Name
  - Call-by-Need
  - Macros
  - Difficulties with parameter passing mechanisms
- Semantics of parameter passing
- Implementation of procedure calls
  - Stack, activation records
  - Caller Vs Callee responsibilities
- Exception-handling

Memory allocation

- Simple types Vs structures and arrays
- Global/static variables
- Stack allocation
  - How local variables and parameters are accessed
  - Accessing nonlocal variables
- Structure of activation records
- Heap allocation
  - Explicit Vs Automatic management
  - Fragmentation
  - Garbage collection
    - Reference-counting Vs mark/sweep Vs copying collection
    - Conservative GC

Implementation Aspects OO Languages

- Layout of structures and objects
  - Accessing data members
- Efficient implementation of virtual functions
- Subtype principle and how it dictates the implementation of OO languages

Code Generation

- Intermediate code formats
- Syntax-directed definition for IC generation
  - Declarations
  - Expressions
  - Assignments
    - l- and r-values
    - accessing arrays and other complex data types
  - Function calls
  - Conditionals
    - Short-circuit evaluation of boolean expressions and handling of conditionals
    - Loops

Machine Code Generation

- Assembly code versus machine code generation issues
  - Linkers, shared libraries, executables, symbol tables, etc.
- Register allocation
  - Cost savings due to use of registers
  - Graph-coloring based algorithm and heuristics
  - Works well in practice, no sense in using “register” declarations in your program, which will likely lead to less efficient code
- Instruction selection
  - Instruction set specification
  - Automated generation of assembly code from specifications
  - Optimal code generation using dynamic programming
    - Combines register allocation with assembly code generation

Code Optimization

- High-level, intermediate code and low-level optimizations
- High-level optimizations
  - Inlining, partial evaluation, tail call elimination, loop reordering, ...
- Intermediate code optimizations
  - CSE
  - constant and copy propagation
  - strength reduction, loop-invariant code motion
  - dead-code elimination
  - jump-threading

Code Optimizations

- Low-level optimizations
  - Register allocation
  - Instruction scheduling
    - loop-unrolling, instruction reordering
    - delay-slot filling and branch-prediction
    - RISC Vs CISC processors
  - Peep-hole optimization
    - redundant instructions
    - flow-of-control
    - algebraic simplification
  - Profile-based optimization

Dataflow Analysis

- Formulation
- Setting-up dataflow equations
- Approximation, direction of approximation, and soundness
- Recursion and fixpoint iteration
- Applications
  - Reaching definitions
  - Available expressions (CSE)
  - Live variables
- Difficulties
  - Procedure calls
  - Aliasing