CS254: Programming Language Design and Implementation

Overview

Types of languages

- Imperative
  - Von Neumann (Fortran, Pascal, BASIC, C, …)
  - Object-oriented (C++/Java, Smalltalk, …)
- Declarative
  - Functional (Scheme, Lisp, ML, …)
  - Logic, constraint-based (Prolog, VisiCalc, …)

Phases of Compilation

- Scanning (lexical analysis)
- Parsing (syntax analysis)
- Semantic analysis
- Intermediate code generation
- Optimization
- Target code generation

Scanners

- Use regular expressions to express tokens
- Defines a set of strings over the characters contained in some alphabet, defining a language
- Three basic operations
  - Union – e.g., a|b
  - Concatenation – e.g., ab
  - Closure – (Kleene closure) – e.g., a* - where a can be a set concatenated with itself zero or more times

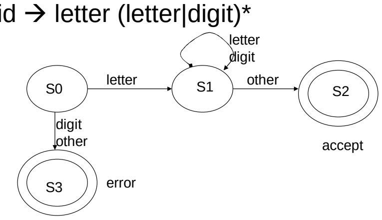
Recognizers

Identifier

letter → (a|b|c| … |z|A|B|C| … |Z)

digit → (0|1|2|3|4|5|6|7|8|9)

id → letter (letter|digit)*



Parsers

- Check input for grammatical correctness
- Determine syntax of token stream
- Constructs intermediate representation
- Produces meaningful error messages

Needs mathematical model of syntax→grammar

Needs algorithm for testing syntax → parsing

Types of parsers

- Top-down (LL) parsers
  - Left to right, leftmost derivation
  - Starts at the root of the derivation tree and fills in
  - Predicts next state with n lookahead
- Bottom-up (LR) parsers
  - Left to right, rightmost derivation
  - Starts at the leaves and fills in
  - Start with input string, end with start symbol
  - Starts in a state valid for legal first tokens
  - Change state to encode possibilities as input is consumed
  - Use a stack to store both state and sentential form

Attribute Grammars

- Generalization of context-free gammars
- Each grammar symbol has an associated set of attributes
- Augment grammar with rules that define values
  - Not allowed to refer to any variables or attributes outside the current production
- High-level specification, independent of evaluation scheme

Scope Rules

- Scope
  - a program region of maximal size in which no bindings change
  - name space that maps a set of names to a set of variables
- Scope of a binding
  - Static – can be determined based purely on textual rules at compile time
  - Dynamic – depends on the flow of execution at run time
- Lifetime of a variable – single execution of the scope or multiple executions of the scope (decides where data can be allocated)

Types

- Types
  - Values that share a set of common properties
  - Defined by language and/or programmer
- Type system
  - Set of types in a programming language
  - Rules that use types to specify program behavior
- Example type rules
  - If operands of addition are of type integer, then result is of type integer
  - The result of the unary & operator is a pointer to the object referred to by the operand
- Advantages of typed languages
  - Ensure run-time safety
  - Expressiveness (overloading, polymorphism)
  - Provide information for code generation

Basic Paradigms for Control Flow

- Sequencing
- Selection
- Iteration
- Procedural abstraction
- Recursion
- Non-determinacy
- concurrency

Code Generation

- Imperative programming models: compute via iteration and side effects
- Functional programming model: compute via recursion and substitution of parameters into functions
- Logic programming model: compute via resolution of logical statements, driven by the ability to unify variables and terms

Functional Programming

- E.g., Lisp, Scheme
- Formalism: Church's lambda calculus
- Key idea: no mutable state/side effects; everything done by composing functions

Functional Programming Design Features and Issues

- First-class and higher-order functions
- Polymorphism
- Recursion
- Garbage collection
- Control flow and evaluation order
- Support for list-based data

Logic Programming Model

- E.g., Prolog
- Formalism: Predicate calculus
- Key idea: collection of axioms from which theorems can be proven

Logic Programming Design Issues

- Horn clauses and terms
- Resolution and unification
- Search and execution order
- List manipulation
- High-order predicates for inspection and modification of the database

Data Abstractions

- Scopes and lifetime
  - Global variables (introduced by Basic)
    - Lifetime and scope spans program execution
  - Local variables (introduced by Fortran)
    - Lifetime and scope limited to execution of subroutine
  - Nested scopes (Algol 60)
    - Allows subroutines or blocks to themselves be local
  - Static variables (Fortran)
    - Lifetime spans execution, names visible in a single scope
  - Modules (Modula-2)
    - Allow a collection of subroutines to share a set of static variables
  - Module types (Euclid)
    - Allow instantiation of multiple instances of a given abstraction
  - Classes (Smalltalk, C++, Java)
    - Allow definition of families of related abstractions

Why abstractions?

- Reduce conceptual load
  - Hide implementation details
- Independence among program components
  - Replacement of pieces without rewriting others
  - Organizational compartmentalization
- Fault containment
  - Enforce division of labor
  - Prevent access to things you shouldn't see

Object-Oriented Programming

- Fundamental concepts
  - Encapsulation
  - Inheritance
  - Dynamic method binding
- Class – module as the abstract type including data and method definition
- Object – instance of a class

Concurrency in the form of

- Explicitly concurrent languages
  - e.g., Algol 68, Java
- Compiler-supported extensions
  - e.g., HPF, Power C/Fortran
- Library packages outside the language proper
  - e.g., pthreads

Code Improvement

- Peep-hole optimization
  - Short sequences of instructions
- Local optimization
  - Basic blocks
- Intra-procedural optimization
  - Across basic blocks but within a procedure/subroutine
- Inter-procedural optimization
  - Across procedures/subroutines

Peephole Optimization

- Constant folding – e.g., 3X2 → 6
- Constant propagation
- Elimination of redundant loads and stores
- Common subexpression elimination
- Copy propagation
- Strength reduction
- Load and branch delays
- Elimination of useless instructions
- Exploitation of instruction set

Redundancy Elimination

- Can be done at the basic block and intraprocedural level as well
- Requires data flow analysis
  - Static single assignment (SSA) form
  - Global value numbering
    - Assign the same name (number) to any two or more symbolically equivalent computations

Loop optimizations

- Loop invariants – move out of body
- Loop unrolling and software pipelining
  - Helps improve instruction scheduling
- Loop reordering
  - Requires dependence analysis
  - Can improve locality and parallelism

Elements of a Compiler

- Scanner (lexical analysis) (input – character stream, ouput – token stream)
- Parser (syntax analysis) (output – parse tree)
- Semantic analysis (output – AST with annotations)

- Machine-independent
  - Intermediate code generation (output – CFG with basic block pseudo-code)
  - Local redundancy elimination (output - modified CFG)
  - Global redundancy elimination (output - modified CFG)
  - Loop improvement (output - modified CFG)

- Machine-specific
  - Target code generation (output – assembly language)
  - Preliminary instruction scheduling (output – modified assembly)
  - Register allocation (output – modified assembly)
  - Final instruction scheduling (output – modified assembly)
  - Peephole optimization (output – modified assembly)