CS2210 Compiler Design

Prof. Dr. Markus Mock
University of Pittsburgh

Administrative Stuff

- Webpage
  http://www.cs.pitt.edu/~mock/cs2210/
- Class email - be sure to be on list
- Office hours: Monday & Wednesday 4-6pm and by appointment

Why Study Compilers?

- Meeting area of programming languages, architecture
  - Capability of compilers greatly influences design of both
- Program analysis
  - Widely useful
    - Software engineering tools
    - DB query optimizers
    - Programmable graphics renderers
    - Safety checking of code (extensible OS)
Language Implementation Goals

- Correctness
- Efficiency
  - Time, data space, code space
  - At compile & run-time
- Support expressive language features e.g. dynamic dispatching
- Support desirable programming environments
  - Fast turnaround time
  - Separate compilation
  - Source level debugging

Language Implementation Techniques

- Interpreter
  - Easy to implement
  - Built-in run-time support
  - Slow
- Compiler
  - Significant implementation effort
  - Requires run-time support (e.g., garbage collector)
  - Fast, optimized target code
- Hybrid
  - Example: bytecode interpreters for Perl, Java

Interpreted vs. compiled languages

- C, C++, Fortran
  - Usually compiled
- Lisp, Haskell, Perl, Python
  - Usually interpreted
- Why?
Compiler Overview

Source Program → Compiler → Target Program

↓
Error messages

Different Compilers
- Text formatters, e.g., Latex
- Silicon compilers
- Query interpreters

Compilation Tools

source → preprocessor → compiler → assembler

libraries → linker/loader → object
Compiler Phases

- Lexical analysis
- Syntactic analysis
- Semantic analysis / type checking
- Intermediate code generation
- Code optimization
- Target code generation

Lexical Analysis / Scanning

- Break input character stream into lexical units called tokens
- Example:
  position := initial + rate * 60
  <id> <asOp> <id> <addOp> <id> <mulOP> <number>

Syntax Analysis / Parsing

- Breaks token stream into grammatical groups (parse trees)
- Example (on board)
Semantic Analysis

- Typechecking
- Decorate parse tree with semantic information
- Symbol Table(s): used to record semantic information

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>real</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>rate</td>
<td>real</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>initial</td>
<td>real</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Intermediate Code Generator

- Translate parse tree to intermediate form
- Three address code
  - t1 = t2 + t3
- SSA (static single assignment form)
- PDG (program dependence graph)
- Many others

Code Optimizer

- "optimize code"
  - Goal: improve performance, usually time
    - Can degrade performance
  - Significant time spent here
Optimization
- Analyze
  - Learn useful facts about the program
- Transform
  - Faster, smaller, less power etc.
- Requirement
  - Don't change semantics!
- Program Analysis determines when transformations are
  - legal
  - Profitable
- Scope
  - Peephole, local, global, interprocedural, whole-program

Code Generator (Back end)
- Generate machine code
- Instruction selection
  - Easy for RISC
- Register Allocation
- Code scheduling
- Backend optimizations
  - Machine-specific optimization

Language Design & Overview of COOL

CS2210
Lecture 2
Lecture 2 Outline

- Today’s topic: language design
- Why are there new languages?
- Design Principles
  - Abstraction
  - Types
  - Reuse
- Cool
- The Course Project

Programming Languages

- Languages are adopted to fill a void
  - Enable a previously difficult/impossible application
  - Orthogonal to language design quality (almost)
- Programmer training is the dominant cost
  - Languages with many users are replaced rarely
  - Popular languages become ossified
- But easy to start in a new niche . . .

Why So Many Languages?

- Application domains have distinctive (and conflicting) needs
- Examples:
  - Scientific Computing: high performance
  - Business: report generation
  - Artificial Intelligence: symbolic computation
  - Systems programming: low-level access
  - Web based programming
  - Special purpose languages
Topic: Language Design

- No universally accepted metrics for design
- Is there a best language?
- Does widespread use determine “best” language?
  - Is C the best language?
- Good language design is hard
- But good design principles

Abstraction

- Abstraction = detached from concrete details
- Abstraction necessary to build software systems
- Modes of abstraction
  - Via languages/compilers:
    - Higher-level code, few machine dependencies
  - Via subroutines
    - Abstract interface to behavior
  - Via modules
    - Export interfaces; hide implementation
  - Via abstract data types
    - Bundle data with its operations

Types

- Originally, few types
  - FORTRAN: scalars, arrays
  - LISP: no static type distinctions
- Realization: Types help
  - Allow the programmer to express abstraction
  - Allow the compiler to check against many frequent errors
  - Sometimes to the point that programs are guaranteed “safe”
- More recently
  - Lots of interest in types
  - Experiments with various forms of parameterization
  - Best developed in functional programming
Reuse

- Reuse = exploits common patterns in software systems
- Goal: mass-produced software components
- Reuse is difficult
- Two popular approaches (combined in C++)
  - Type parameterization (List[int], List[double])
  - Classes and inheritance: C++ derived classes
- Inheritance allows
  - Specialization of existing abstraction
  - Extension, modification, hiding behavior

Trends

- Language design
  - Many new special-purpose languages
  - Popular languages to stay
- Compilers
  - More needed and more complex
  - Driven by increasing gap between
    - new languages
    - new architectures
  - Venerable and healthy area

Cool Overview

- Classroom Object Oriented Language
- Designed to
  - Be implementable in one semester
  - Give a taste of implementation of modern
    - Abstraction
    - Static typing
    - Reuse (inheritance)
    - Memory management
    - And more...
- But many things are left out
A Simple Example

```plaintext
class Point {
    x : Int ← 0;
    y : Int ← 0;
};
```

- Cool programs are sets of class definitions
  - A special class `Main` with a special method `main`
  - No separate notion of subroutine
- `class = a collection of attributes and methods`
- Instances of a class are objects

Cool Objects

```plaintext
class Point {
    x : Int ← 0;
    y : Int; (* use default value *)
};
```

- The expression "new Point" creates a new object of class `Point`
  - An object can be thought of as a record with a slot for each attribute

```
x  y
0  0
```

Methods

- A class can also define methods for manipulating the attributes
  ```plaintext
  class Point {
      x : Int ← 0;
      y : Int ← 0;
      movePoint(newx : Int, newy : Int): Point {
          x ← newx;
          y ← newy;
          self;
      } -- close block expression
  } -- close method
  ```
- Methods can refer to the current object using `self`
Information Hiding in Cool

- Methods are global
- Attributes are local to a class
  - They can only be accessed by the class’s methods
- Example:

  ```
  class Point {
    . . .
    x() : Int { x; }
    setx(newx : Int) : Int { x ← newx; }
  }
  ```

Method Implementation

- Each object knows how to access the code of a method
- As if the object contains a slot pointing to the code

In reality implementations save space by sharing these pointers among instances of the same class

```
x y movePoint
```

Inheritance

- We can extend points to colored points using subclassing

```
class ColorPoint inherits Point {
    color : Int ← 0;
    movePoint(newx : Int, newy : Int) : Point {
        { color ← 0;
          x ← newx; y ← newy;
        }
    }
}
```
Cool Types

- Every class is a type
- Base classes:
  - Int: for integers
  - Bool: for boolean values: true, false
  - String: for strings
  - Object: root of the class hierarchy
- All variables must be declared
  - compiler infers types for expressions

Cool Type Checking

- Is well typed if A is an ancestor of B in the class hierarchy
- Anywhere an A is expected a B can be used
- Type safety:
  - A well-typed program cannot result in runtime type errors

Method Invocation and Inheritance

- Methods are invoked by dispatch
- Understanding dispatch in the presence of inheritance is a subtle aspect of OO languages

```
p : Point;
p ← new ColorPoint;
p.movePoint(1,2);
```
- p has static type Point
- p has dynamic type ColorPoint
- p.movePoint must invoke the ColorPoint version
Method Invocation

Example: invoke one-argument method \( m \)

1. Eval. \( e \)
2. Find class of \( e \)
3. Find code of \( m \)
4. Eval. Argum
5. Bind self and \( x \)
6. Run method

Other Expressions

- Expression language (every expression has a type and a value)
  - Loops: \( \text{while} \ E \ \text{loop} \ E \ \text{pool} \)
  - Conditionals \( \text{if} \ E \ \text{then} \ E \ \text{else} \ E \ \text{fi} \)
  - Case statement \( \text{case} \ E \ \text{of} \ x : \text{Type} \Rightarrow E; \ldots \text{esac} \)
  - Arithmetic, logical operations
  - Assignment \( x \leftarrow E \)
  - Primitive I/O \( \text{out} \_\text{string}(s), \text{in} \_\text{string}(), \ldots \)

- Missing features:
  - Arrays, Floating point operations, Interfaces, Exceptions, ...

Cool Memory Management

- Memory is allocated every time \texttt{new} is invoked
  - Memory is deallocated automatically when an object is not reachable anymore
  - Done by the garbage collector (GC)
  - There is a Cool GC
Course Project

- A complete compiler
  - Cool $\Rightarrow$ MIPS assembly language
  - No optimizations
- Split in 5 programming assignments (PAs)
- There is adequate time to complete assignments
  - But start early and please follow directions
  - Turn in early to test the turn-in procedure
- Individual project

Programming Assignment I

- Write an interpreter for a stack machine
  - ... in Cool
  - Due on 9/8