Introduction to Programming Languages and Compilers

CS164

11:00-12:30 TT

10 Evans

ICOM 4036 - Outline

- Prontuario
- Course Outline
- Brief History of PLs
- Programming Language Design Criteria
- Programming Language Implementation

Programming Assignments Highlights

- Implement a compiler in four phases
- Teams of two students (Choose your partner!)
- Development in C++ or Java for Linux
- Use Academic Computer Center (Amadeus)
- Can work on your personal computers
- Source Language = COOL (UC Berkeley CS164)
- Target Language = MIPS Assembly (SPIM)
- Each project must have some unique feature chosen by the development team
- Each compiler must pass a minimal set of tests in order to pass the class.

Homework for next week

- Read the COOL Reference Manual

  http://www-inst.eecs.berkeley.edu/~cs164/public_html-sp03
- Choose your partner
  - notify me by email
- Choose your development language
  - C++ or Java
- Read the Flex (C++) or JLex (Java) manual

  http://inst.eecs.berkeley.edu/~cs164/public_html-sp03/documentation.html

(Short) History of High-Level Languages

- 1953 IBM develops the 701
- All programming done in assembly
- Problem: Software costs exceeded hardware costs!
- John Backus: “Speedcoding”
  - An interpreter
  - Ran 10-20 times slower than hand-written assembly

FORTRAN I

- 1954 IBM develops the 704
- John Backus
  - Idea: translate high-level code to assembly
  - Many thought this impossible
    - Had already failed in other projects
- 1954-7 FORTRAN I project
- By 1958, >50% of all software is in FORTRAN
- Cut development time dramatically
  - (2 wks ! 2 hrs)

FORTRAN I

- The first compiler
  - Produced code almost as good as hand-written
  - Huge impact on computer science

- Led to an enormous body of theoretical work

- Modern compilers preserve the outlines of FORTRAN I

History of Ideas: 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

History of Ideas: 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

History of Ideas: 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

Programming Language Economics 101

- 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
  - Special purpose languages

Topic: Language Design

- No universally accepted metrics for design
- "A good language is one people use" ?
- NO !
  - Is COBOL the best language?
- Good language design is hard

Language Evaluation Criteria

Characteristic	Criteria		
	Readability	Writeability	Reliability
Simplicity	*	*	*
Data types	*	*	*
Syntax design	*	*	*
Abstraction		*	*
Expressivity		*	*
Type checking			*
Exception handling			*

Why Study Languages and Compilers ?

- Increase capacity of expression
- Improve understanding of program behavior
- Increase ability to learn new languages

- Learn to build a large and reliable system
- See many basic CS concepts at work

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

How are Languages Implemented?

- Two major strategies:
  - Interpreters (older, less studied)
  - Compilers (newer, much more studied)
- Interpreters run programs “as is”
  - Little or no preprocessing
- Compilers do extensive preprocessing

Language Implementations

- Batch compilation systems dominate
  - E.g., gcc
- Some languages are primarily interpreted
  - E.g., Java bytecode
- Some environments (Lisp) provide both
  - Interpreter for development
  - Compiler for production

The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend English.

Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

- Note the
  - Capital “T” (start of sentence symbol)
  - Blank “ “ (word separator)
  - Period “.” (end of sentence symbol)

More Lexical Analysis

- Lexical analysis is not trivial. Consider:

  ist his ase nte nce

- Plus, programming languages are typically more cryptic than English:

  `*p->f ++ = -.12345e-5`

And More Lexical Analysis

- Lexical analyzer divides program text into "words" or "tokens"

```
if x == y then z = 1; else z = 2;
```

- Units:

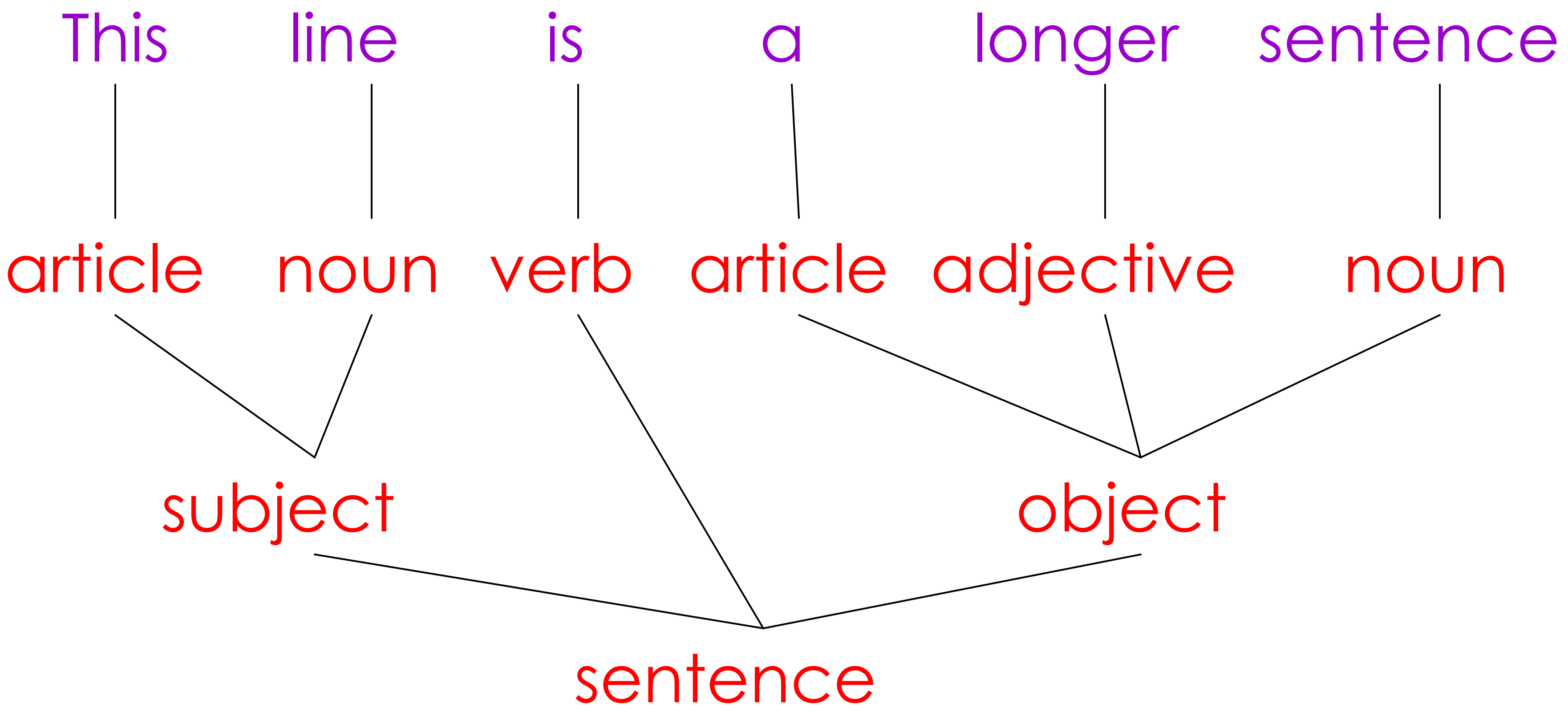
```
if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;
```

Parsing

- Once words are understood, the next step is to understand sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

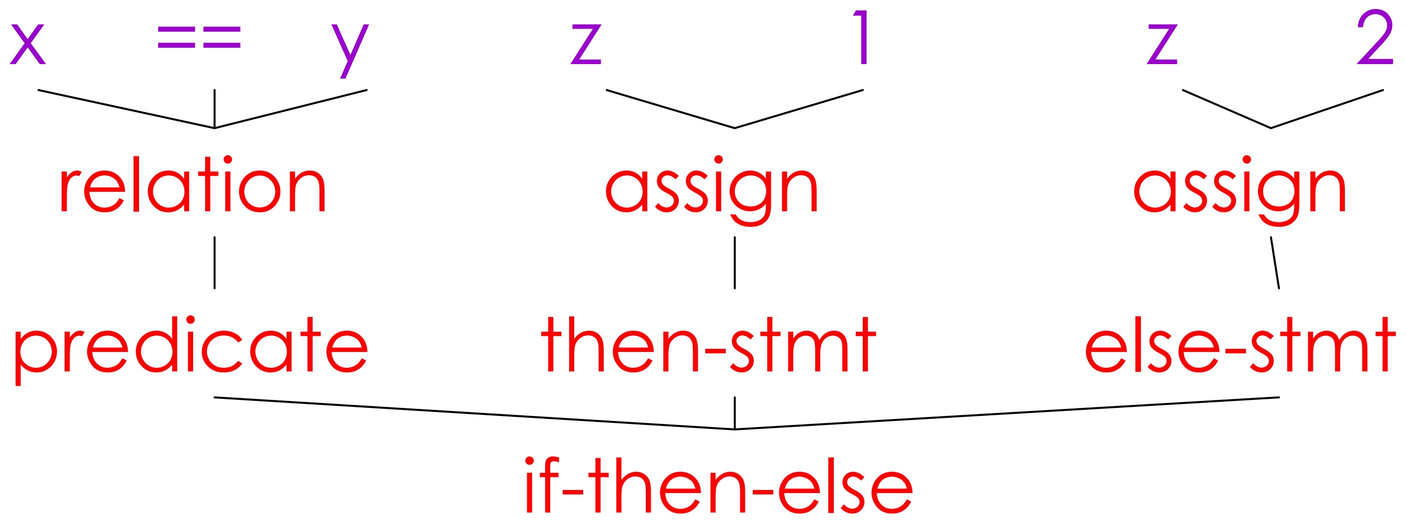


Parsing Programs

- Parsing program expressions is the same
- Consider:

  If x == y then z = 1; else z = 2;

- Diagrammed:



Semantic Analysis

- Once sentence structure is understood, we can try to understand "meaning"
  - But meaning is too hard for compilers
- Compilers perform limited analysis to catch inconsistencies
- Some do more analysis to improve the performance of the program

Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does "his" refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints "4"; the inner definition is used

```
{
  int Jack = 3;
  {
     int Jack = 4;
     cout << Jack;
  }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings

- Example:

  Jack left her homework at home.

- A "type mismatch" between her and Jack; we know they are different people
  - Presumably Jack is male

Examples of Semantic Checks in PLs

- Variables defined before used
- Variables defined once
- Type compatibility
- Correct arguments to functions
- Constants are not modified
- Inheritance hierarchy has no cycles
- …

Optimization

- No strong counterpart in English, but akin to editing
- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - In general, conserve some resource
- The project has no optimization component

Optimization Example

X = Y * 0 is the same as X = 0

NO!

Valid for integers, but not for floating point numbers

Examples of common optimizations in PLs

- Dead code elimination
- Evaluating repeated expressions only once
- Replace expressions by simpler equivalent expressions
- Evaluate expressions at compile time
- Inline procedures
- Move constant expressions out of loops
- …

Code Generation

- Produces assembly code (usually)
- A translation into another language
  - Analogous to human translation

Intermediate Languages

- Many compilers perform translations between successive intermediate forms
  - All but first and last are *intermediate languages* internal to the compiler
  - Typically there is 1 IL
- IL's generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly

Intermediate Languages (Cont.)

- IL's are useful because lower levels expose features hidden by higher levels
  - registers
  - memory layout
  - etc.

- But lower levels obscure high-level meaning

Issues

- Compiling is almost this simple, but there are many pitfalls.
- Example: How are erroneous programs handled?
- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design

Compilers Today

- The overall structure of almost every compiler adheres to our outline

- The proportions have changed since FORTRAN
  - Early: lexing, parsing most complex, expensive

  - Today: optimization dominates all other phases, lexing and parsing are cheap

Trends in Compilation

- Compilation for speed is less interesting. But:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures)
- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting concurrency errors (data races)
  - ...