Introduction to Programming Languages and Compilers

CS164

11:00-12:00 MWF

10 Evans

Notes by G. Necula, with additions by P. Hilfinger

Administrivia

- Course home page:
  http://www-inst.eecs.berkeley.edu/~cs164
- Concurrent enrollment: see me after class
- Pick up class accounts at the end of lecture Wednesday
- Pick a partner
- We're understaffed. Those in 10-11, 3-4 sections might consider moving to 9-10, 4-5; will discuss more in section meetings.

Course Structure

- Course has theoretical and practical aspects: analysis and translation of programming languages uses both.
- Regular homework = theory, should be individual work.
- Programming assignments = practice, in teams
- All submissions will be electronic

Academic Honesty

- Don't use work from uncited sources
  - Including old code
- We use plagiarism detection software
  - 6 cases in last few semesters

The Course Project

- Course has hidden agenda: programming design and experience.
- Substantial project in parts.
- Provides example of how complicated problem might be approached.
- Validation (testing) is part of the project.
- Also a chance to introduce important tool: version control, which we'll use to monitor your progress
- General rule: start early!

How are Languages Implemented?

- Two major strategies:
  - Interpreters (older, less studied)
  - Compilers (newer, more extensively studied)
- Interpreters run programs "as is"
  - Little or no preprocessing
- Compilers do extensive preprocessing
  - Most implementations use compilers
- New trend is hybrid: "Just-In-Time" compilation, interpretation+compilation

(Short) History of High-Level Languages

- Initially, programs "hard-wired" or entered electro-mechanically: Analytical Engine, Jacquard Loom, ENIAC, punched-card-handling machines
- Programs encoded as numbers (machine language) stored as data: Manchester Mark I, EDSAC.
- 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
- 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

After FORTRAN

- Lisp, late 1950s: dynamic, symbolic data structures.
- Algol 60: Europe's answer to FORTRAN: modern syntax, block structure, explicit declaration. Set standard for language description. Dijkstra: "A marked improvement on its successors."
- COBOL: late 1950's. Business-oriented. Records.

The 60s Language Explosion

- APL (arrays), SNOBOL (strings), FORMAC (formulae), and many more.
- 1967-68: Simula 67, first "object-oriented" language.
- Algol 68: Combines FORTRAish numerical constructs, COBOLish records, pointers, all described in rigorous formalism. Remnants remain in C, but Algol68 deemed too complex.
- 1968: "Software Crisis" announced. Trend towards simpler languages: Algol W, Pascal, C

The 1970s

- Emphasis on "methodology": modular programming, CLU, Modula family.
- Mid 1970's: Prolog. Declarative logic programming.
- Mid 1970's: ML (Metalanguage) type inference, pattern-driven programming.
- Late 1970's: DoD starts to develop Ada to consolidate >500 languages.

And on into the present

- Complexity increases with C++.
- Then decreases with Java.
- Then increases again (C#).
- Proliferation of little or specialized languages and scripting languages: HTML, PHP, Perl, Python, Ruby, …

Problems to address

- How to *describe* language clearly for programmers, precisely for implementors?
- How to implement description, and know you're right?
  - *Automatic conversion* of description to implementation
  - Testing
- How to save implementation effort?
  - Multiple languages to multiple targets: can we *re-use* effort?
- How to make languages *usable?*
  - Handle errors reasonably
  - Detect questionable constructs
  - Compile quickly

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 letters and words.
  - Words are 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;

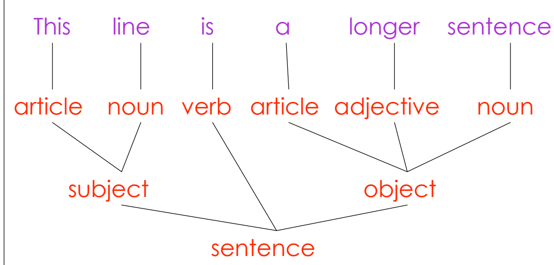
- Tokens:

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

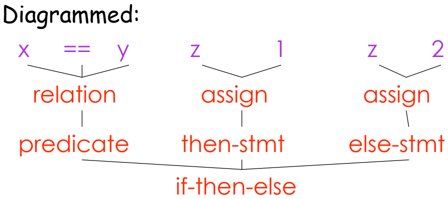


Parsing Programs

- 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

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
- Not an emphasis in the project.

Optimization Example (in C)

```
for (int i = 0; i < N; i += 1) {
    for (int j = 0; j < N; j += 1) {
        A[i][j] += B[i]*C[j]
    }
}
```

```
for (i = 0; i < N; i += 1) {
  double tmp1 = B[i];
  double* tmp2 = &A[i];
  for (int j = 0; j  < N; j +=1)
     tmp2[j] += tmp1*C[j];
}
```


Optimization is tricky

```
for (i = 0; i < N; i += 1)
   A[i] *= D/A[0];
```

is NOT

```
tmp1 = D/A[0];
for (i = 0; i < N; i += 1)
   A[i] *= tmp1;
```


Code Generation

- Produces assembly code (usually)
  - which is then assembled into executables by an assembler
- A translation into another language
  - Analogous to human translation

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

- Optimization for speed is less interesting. But:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures)
  - Small devices where speed = longer battery life
- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting concurrency errors (data races)
  - ...

Trends, contd.

- Compilers
  - More needed and more complex
  - Driven by increasing gap between
    - new languages
    - new architectures
  - Venerable and healthy area

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