CSE 3302 Notes 1: Introduction

(Last updated 11/11/15 8:41 AM)

References:

Gabbrielli-Martini: Intro., 1, 13.3, 13.4, 13.5, 13.6
Dybvig: 1
Steele, *Growing a Language:* `http://ranger.uta.edu/~weems/steele.pdf`

1.1. Language Design

Early IBM Tradition

Embedded Systems Attitude

> We'll never code in C, we need assembler. (Late 80s)
> We'll never code in Java, we need (some parts of) C++.

(`http://www.ee.ryerson.ca/~elf/hack/realmen.html`)

The inevitability of converting code . . .

Why so many languages? (Why so few "good" ones?)

(`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?id=365257`)

What makes a language popular?

Steele: "I stand on this claim: I should not design a small language, and I should not design a large one. I need to design a language that can grow."

"A language design can no longer be a thing. It must be a pattern—a pattern for growth—a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal."

Specific concepts and features are more useful (in academic study) than bundles.

How about domain-specific languages and end-user programming? (programming vs. configuration?)

(`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?id=1922649.1922658`)
(`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?id=2602695.2605205`)

Steele: "We need to put tools for language growth in the hands of the users."

What language should be given to "babies"?
(`https://apstudent.collegeboard.org/apcourse/ap-computer-science-a/about-the-exam/java-subset`)
How long should it take to learn a language?
Is UML a PL?

1.2. THE PROGRAMMING LANGUAGE DESIGN SPECTRUM

ACM Turing Award Winners (http://amturing.acm.org/) with Strong Connections to PLs:

Year	Name	Contribution
1966	Perlis	Programming and compilers “When someone says ‘I want a programming language in which I need only say what I wish done,’ give him a lollipop.”
1971	McCarthy	LISP
1972	Dijkstra	Philosophy of programming “If FORTRAN has been called an infantile disorder, PL/I must be classified as a fatal disease.”
1973	Bachman	COBOL, navigational DB
1974	Knuth	Language implementation http://www.youtube.com/embed/gAXdDEQveKw “compiler research was certainly intensive, representing roughly one third of all computer science in the 60s”
1977	Backus	FORTRAN, formalisms
1978	Floyd	Parsing, semantics https://en.wikipedia.org/wiki/Cycle_detection
1979	Iverson	APL
1980	Hoare	CSP “Inside every large language is a small language struggling to get out ... ” http://i.stanford.edu/pub/cstr/reports/cs/tr/73/403/CS-TR-73-403.pdf
1981	Codd	Relational DB
1983	Thompson	UNIX, scripting
1983	Ritchie	UNIX, C
1984	Wirth	Philosophy of PLs
1987	Cocke	RISC, code optimization
1991	Milner	ML, type inference
1999	Brooks	Systems design “The worst mistake we made was JCL” http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2838899.2822519
2001	Dahl/Nygard	Simula, O-O
2003	Kay	O-O, Smalltalk, MVC “Actually I made up the term ‘object-oriented’, and I can tell you I did not have C++ in mind” https://en.wikiquote.org/wiki/Talk:Edsger_W._Dijkstra#nano-Dijkstras
2005	Naur	Algol 60
2006	Allen	FORTRAN optimization
2008	Liskov	Abstraction, distributed computing

ACM Software System Award: http://awards.acm.org/software_system/year.cfm

Greatest Common Denominator Example:

C, Pascal, PL/0, Scheme, JavaScript, SML, Prolog code on webpage
(http://ranger.uta.edu/~weems/NOTES3302/GCD/)

2320: Where do want to be in 10 years?

What is design? (juggling {correctness, resource requirements, development cost}
or {quality, cost, schedule})

3302: To get ahead, what do you want to *manipulate*?

First Class: Passed as argument, returned from function, assigned to variable, (Constructible?)
Second Class: Passed as argument
Third Class: None of the above

How about functions, threads/processes, processors, messages, channels/pipes?

First-class *classes*?

1.4. Compilation and Interpretation

Practice vs. Possibilities vs. Details

Like . . . the nature of names, bindings, and symbol table(s)

Assembler:

Op Codes
Labels
Data Types (corresponding to hardware capabilities)
Macros
Control of Assembly

Compiler:

Maximizes checking that can be performed without execution of the source program.
May produce code for a machine whose level of similarity to ideal "language machine" may vary.
Traditional languages (as opposed to scripting languages) have symbol tables only in the compiler and use static (AKA lexical) scoping.

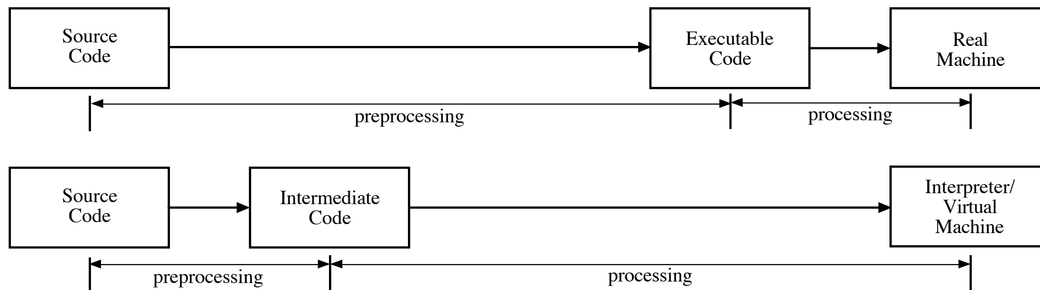
Interpreter/Virtual Machine:

Ranges from general hardware machine to "language machine" (with features such as strings, hashing for object property names, memory management, and dynamic scope).
In recent years, Just-In-Time "compilers" translate instructions for virtual machine to real machine code.

Linker/Loader:

Resolves external references in several object files
Possibly commences execution or just produces executable

From D. Grune, et.al., *Modern Compiler Design*, Wiley, 2000.



And, going meta:

(`https://en.wikipedia.org/wiki/Metamodeling`)

`http://www.scheme.com/tspl4/examples.html#./examples:h7`

`https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-25.html#%_chap_4`

`https://en.wikipedia.org/wiki/Bootstrapping_(compilers)`

The Little JavaScripter

And, run-time at compile-time:

`https://msdn.microsoft.com/en-us/library/Dn956974.aspx`

1.5. An Overview of Compilation

Preprocessor (e.g. C)

Include files
Macros
Conditional text
Compiler directives

Scanning/Lexing/Tokenizing/Lexical Analysis

Remove comments / white space

Collect “minimal” meaningful substrings

Identifiers/reserved words
Operators
Constants (strings, numbers)

Strong connections to regular expressions and finite-state automata (CSE 3315)
(`https://en.wikipedia.org/wiki/Lex_(software)`)

Parsing/Syntax Analysis

Construct tree representing nesting structure of language constructs and expressions
Practical languages allow both bottom-up and top-down (e.g. recursive descent) approaches
(`https://en.wikipedia.org/wiki/Yacc`)

Symbol Tables

Data structures
Scope/namespaces

Semantic Analysis

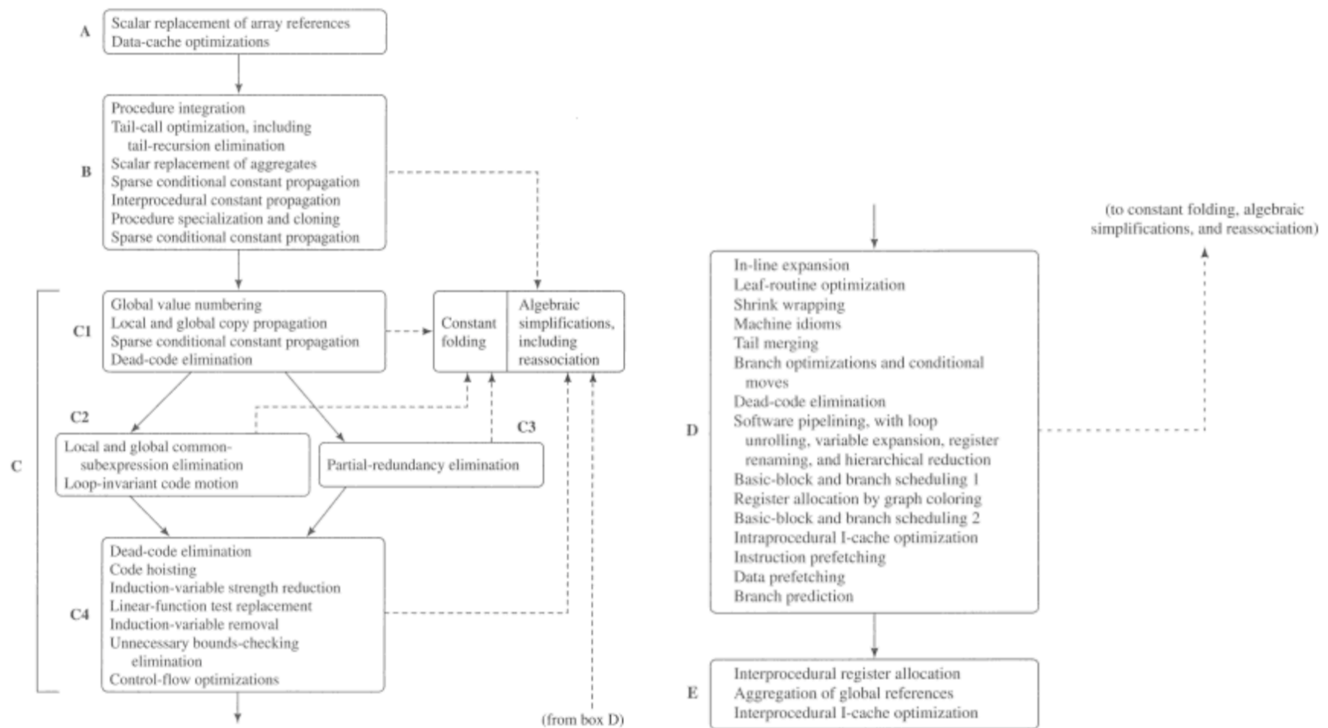
Attribute grammar (Notes 4) evaluation - functions for sending semantic information through abstract syntax tree

Type checking/inference (`http://www.amazon.com/dp/0262162091`)

Intermediate Code Generation

High-level machine language (`http://adriansampson.net/blog/llvm.html`)

Code Improvement (`http://www.amazon.com/dp/1558603204`)



To get a handle on this . . . (over the entire semester)

`http://ranger.uta.edu/~weems/NOTES3302/BASELINE/`

PL/0 - Pascal subset (see syntax diagrams: `http://ranger.uta.edu/~weems/NOTES3302/BASELINE/DIAGRAMS/`)
One-pass recursive-descent compiler, no intermediate code, no optimization
Scanner (`getsym`) is hand coded
Symbol `table` uses linear search
Stack-based interpreter

1.6. THE CHOSEN FEW . . .

Scheme (Lisp, SML, Haskell)
Scripting (JavaScript)
Pascal, C
C++
Java
Prolog (ASP: `http://potassco.sourceforge.net/clingo.html`)
Algol 60
Algol 68
Smalltalk