Programming Languages

Session 1 – Main Theme
Programming Languages Overview & Syntax

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

1. Instructor and Course Introduction
2. Introduction to Programming Languages
3. Programming Language Syntax
4. Conclusion

Who am I?

- Profile -

- 31 years of experience in the Information Technology Industry, including thirteen years of experience working for leading IT consulting firms such as Computer Sciences Corporation
- PhD in Computer Science from University of Colorado at Boulder
- Past CEO and CTO
- Held senior management and technical leadership roles in many large IT Strategy and Modernization projects for fortune 500 corporations in the insurance, banking, investment banking, pharmaceutical, retail, and information management industries
- Contributed to several high-profile ARPA and NSF research projects
- Played an active role as a member of the OMG, ODMG, and X3H2 standards committees and as a Professor of Computer Science at Columbia initially and New York University since 1997
- Proven record of delivering business solutions on time and on budget
- Original designer and developer of jcrew.com and the suite of products now known as IBM InfoSphere DataStage
- Creator of the Enterprise Architecture Management Framework (EAMF) and main contributor to the creation of various maturity assessment methodology
- Developed partnerships between several companies and New York University to incubate new methodologies (e.g., EA maturity assessment methodology developed in Fall 2008), develop proof of concept software, recruit skilled graduates, and increase the companies' visibility

How to reach me?

	Cell	(212) 203-5004
	Email	jcf@cs.nyu.edu
	AIM, Y! IM, ICQ	jcf2_2003
	MSN IM	jcf2_2003@yahoo.com
	LinkedIn	http://www.linkedin.com/in/jcfranchitti
	Twitter	http://twitter.com/jcfranchitti
	Skype	jcf2_2003@yahoo.com

What is the course about?

- Course description and syllabus:
  - http://www.nyu.edu/classes/jcf/CSCI-GA.2110-001_su14
  - http://cs.nyu.edu/courses/summer14/G22.2110-001/index.html

- Textbook:
  - ***Programming Language Pragmatics (3rd Edition)***

Michael L. Scott
Morgan Kaufmann
ISBN-10: 0-12374-514-4, ISBN-13: 978-0-12374-514-4, (04/06/09)

Course goals

- Intellectual:
  - help you understand benefit/pitfalls of different approaches to language design, and how they work
- Practical:
  - you may need to design languages in your career (at least small ones)
  - understanding how to use a programming paradigm can improve your programming even in languages that don't support it
  - knowing how a feature is implemented helps understand time/space complexity

Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance

Alignment

Solution Approach

Agenda

1. Instructor and Course Introduction
2. Introduction to Programming Languages
3. Programming Language Syntax
4. Conclusion

Introduction to Programming Languages - Sub-Topics

- Introduction
- Programming Language Design and Usage Main Themes
- Programming Language as a Tool for Thought
- Idioms
- Why Study Programming Languages
- Classifying Programming Languages
- Imperative Languages
- PL Genealogy
- Predictable Performance vs. Writeability
- Common Ideas
- Development Environment & Language Libraries
- Compilation vs. Interpretation
- Programming Environment Tools
- An Overview of Compilation
- Abstract Syntax Tree
- Scannerless Parsing

Introduction (1/3)

- Why are there so many programming languages?
  - evolution -- we've learned better ways of doing things over time
  - socio-economic factors: proprietary interests, commercial advantage
  - orientation toward special purposes
  - orientation toward special hardware
  - diverse ideas about what is pleasant to use

Introduction (2/3)

- What makes a language successful?
  - easy to learn (BASIC, Pascal, LOGO, Scheme)
  - easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
  - easy to implement (BASIC, Forth)
  - possible to compile to very good (fast/small) code (Fortran)
  - backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)
  - wide dissemination at minimal cost (Pascal, Turing, Java)

Introduction (3/3)

- Why do we have programming languages? What is a language for?
  - way of thinking -- way of expressing algorithms
  - languages from the user's point of view
  - abstraction of virtual machine -- way of specifying what you want
  - the hardware to do without getting down into the bits
  - languages from the implementor's point of view

Programming Language Design and Usage Main Themes (1/2)

- Model of Computation (i.e., paradigm)
- Expressiveness
  - Control structures
  - Abstraction mechanisms
  - Types and related operations
  - Tools for programming in the large
- Ease of use
  - Writeability
  - Readability
  - Maintainability
  - Compactness – writeability/expressibility
  - Familiarity of Model
  - Less Error-Prone
  - Portability
  - Hides Details – simpler model
  - Early detection of errors
  - Modularity – Reuse, Composability, Isolation
  - Performance Transparency
  - Optimizability
- Note Orthogonal Implementation Issues:
  - Compile time: parsing, type analysis, static checking
  - Run time: parameter passing, garbage collection, method dispatching, remote invocation, just-in-time compiling, parallelization, etc.

Programming Language Design and Usage Main Themes (2/2)

- Classical Issues in Language Design:
  - Dijkstra, “Goto Statement Considered Harmful”,
    - http://www.acm.org/classics/oct95/#WIRTH66
  - Backus, “Can Programming Be Liberated from the von Neumann Style?”
    - http://www.stanford.edu/class/cs242/readings/backus.pdf
  - Hoare, “An Axiomatic Basis For Computer Programming”,
    - http://www.spatial.maine.edu/~worboys/processes/hoare%20axiomatic.pdf
  - Hoare, “The Emperor’s Old Clothes”,
    - http://www.braithwaite-lee.com/opinions/p75-hoare.pdf
  - Parnas, “On the Criteria to be Used in Decomposing Systems into Modules”,
    - http://www.acm.org/classics/may96/

Programming Language as a Tool for Thought

- Roles of programming language as a communication vehicle among programmers is more important than writeability
- All general-purpose languages are Turing Complete (i.e., they can all compute the same things)
- Some languages, however, can make the representation of certain algorithms cumbersome
- Idioms in a language may be useful inspiration when using another language

Idioms

- Copying a string q to p in C:
  - while (*p++ = *q ++) ;
- Removing duplicates from the list @xs in Perl:
  - my % seen = ();

    @xs = grep { ! $seen {$_ }++; } @xs ;
- Computing the sum of numbers in list xs in Haskell:
  - foldr (+) 0 xs

Is this natural? … It is if you're used to it!

Why Study Programming Languages? (1/6)

- Help you choose a language.
  - C vs. Modula-3 vs. C++ for systems programming
  - Fortran vs. APL vs. Ada for numerical computations
  - Ada vs. Modula-2 for embedded systems
  - Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  - Java vs. C/CORBA for networked PC programs

Why Study Programming Languages? (2/6)

- Make it easier to learn new languages some languages are similar; easy to walk down family tree
  - » concepts have even more similarity; if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum
    - Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European).

Why Study Programming Languages? (3/6)

- Help you make better use of whatever language you use
  - » understand obscure features:
    - In C, help you understand unions, arrays & pointers, separate compilation, varargs, catch and throw
    - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals

Why Study Programming Languages? (4/6)

- Help you make better use of whatever language you use (cont.)
  - understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
    - use simple arithmetic equal (use x*x instead of x**2)
    - use C pointers or Pascal "with" statement to factor address calculations
      - http://www.freepascal.org/docs-html/ref/refsu51.html)
    - avoid call by value with large data items in Pascal
    - avoid the use of call by name in Algol 60
    - choose between computation and table lookup (e.g. for cardinality operator in C or C++)

Why Study Programming Languages? (5/6)

- Help you make better use of whatever language you use (cont.)
  - figure out how to do things in languages that don't support them explicitly:
    - lack of suitable control structures in Fortran
    - use comments and programmer discipline for control structures
    - lack of recursion in Fortran, CSP, etc
    - write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)

Why Study Programming Languages? (6/6)

- Help you make better use of whatever language you use (cont.)
  - figure out how to do things in languages that don't support them explicitly:
    - lack of named constants and enumerations in Fortran
    - use variables that are initialized once, then never changed
    - lack of modules in C and Pascal use comments and programmer discipline
    - lack of iterators in just about everything fake them with (member?) functions

Classifying Programming Languages (1/2)

- Group languages by programming paradigms:
  - » imperative
    - von Neumann (Fortran, Pascal, Basic, C, Ada)
      - programs have mutable storage (state) modified by assignments
      - the most common and familiar paradigm
    - object-oriented (Simula 67, Smalltalk, Eiffel, Ada95, Java, C#)
      - data structures and their operations are bundled together
      - inheritance
    - scripting languages (Perl, Python, JavaScript, PHP)
  - » declarative
    - functional (applicative) (Scheme, ML, pure Lisp, FP, Haskell)
      - functions are first-class objects / based on lambda calculus
      - side effects (e.g., assignments) discouraged
    - logic, constraint-based (Prolog, VisiCalc, RPG, Mercury)
      - programs are sets of assertions and rules
    - Functional + Logical (Curry)
  - » Hybrids: imperative + OO (C++)
    - functional + object-oriented (O'Caml, O'Haskell)
    - Scripting (used to glue programs together) (Unix shells, PERL, PYTHON, TCL PHP, JAVASCRIPT)

Classifying Programming Languages (2/2)

- Compared to machine or assembly language, all others are high-level
- But within high-level languages, there are different levels as well
- Somewhat confusingly, these are also referred to as low-level and high-level
  - Low-level languages give the programmer more control (at the cost of requiring more effort) over how the program is translated into machine code.
    - C, FORTRAN
  - High-level languages hide many implementation details, often with some performance cost
    - BASIC, LISP, SCHEME, ML, PROLOG,
  - Wide-spectrum languages try to do both:
    - ADA, C++, (JAVA)
  - High-level languages typically have garbage collection and are often interpreted.
  - The higher the level, the harder it is to predict performance (bad for real-time or performance-critical applications)
  - Note other "types/flavors" of languages: fourth generation (SETL, SQL), concurrent/distributed (Concurrent Pascal, Hermes), markup, special purpose (report writing), graphical, etc.

Imperative Languages

- Imperative languages, particularly the von Neumann languages, predominate
  - They will occupy the bulk of our attention
- We also plan to spend a lot of time on functional, and logic languages

PL Genealogy

- FORTRAN (1957) => Fortran90, HP
- COBOL (1956) => COBOL 2000
  - » still a large chunk of installed software
- Algol60 => Algol68 => Pascal => Ada
- Algol60 => BCPL => C => C++
- APL => J
- Snobol => Icon
- Simula => Smalltalk
- Lisp => Scheme => ML => Haskell
- with lots of cross-pollination:
  e.g., Java is influenced by C++, Smalltalk, Lisp, Ada, etc.

Predictable Performance vs. Writeability

- Low-level languages mirror the physical machine:
  - » Assembly, C, Fortran
- High-level languages model an abstract machine with useful capabilities:
  - » ML, Setl, Prolog, SQL, Haskell
- Wide-spectrum languages try to do both:
  - » Ada, C++, Java, C#
- High-level languages have garbage collection, are often interpreted, and cannot be used for real-time programming.
  - » The higher the level, the harder it is to determine cost of operations.

Common Ideas

- Modern imperative languages (e.g., Ada, C++, Java) have similar characteristics:
  - large number of features (grammar with several hundred productions, 500 page reference manuals, . . .)
  - a complex type system
  - procedural mechanisms
  - object-oriented facilities
  - abstraction mechanisms, with information hiding
  - several storage-allocation mechanisms
  - facilities for concurrent programming (not C++)
  - facilities for generic programming (new in Java)

Language Mechanism & Patterns

- *Design Patterns*: Gamma, Johnson, Helm, Vlissides
  - » Bits of design that work to solve sub-problems
  - » What is mechanism in one language is pattern in another
    - Mechanism: C++ class
    - Pattern: C struct with array of function pointers
    - Exactly how early C++ compilers worked
- Why use patterns
  - » Start from very simple language, very simple semantics
  - » Compare mechanisms of other languages by building patterns in simpler language
  - » Enable meaningful comparisons between language mechanisms

Development Environment & Language Libraries (1/2)

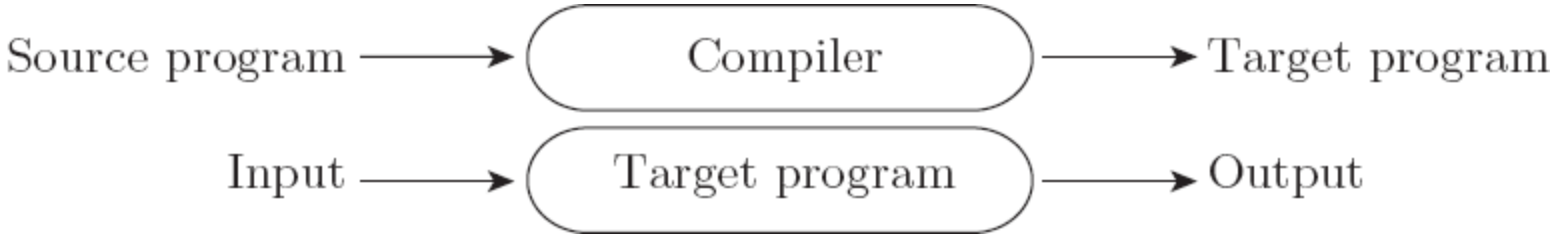
- Development Environment
  - Interactive Development Environments
    - Smalltalk browser environment
    - Microsoft IDE
  - Development Frameworks
    - Swing, MFC
  - Language aware Editors

Development Environment & Language Libraries (2/2)

- The programming environment may be larger than the language.
  - The predefined libraries are indispensable to the proper use of the language, and its popularity
  - Libraries change much more quickly than the language
  - Libraries usually very different for different languages
  - The libraries are defined in the language itself, but they have to be internalized by a good programmer
  - Examples:
    - C++ standard template library
    - Java Swing classes
    - Ada I/O packages
    - C++ Standard Template Library (STL)

Compilation vs. Interpretation (1/16)

- Compilation vs. interpretation
  - not opposites
  - not a clear-cut distinction
- Pure Compilation
  - The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



Compilation vs. Interpretation (2/16)

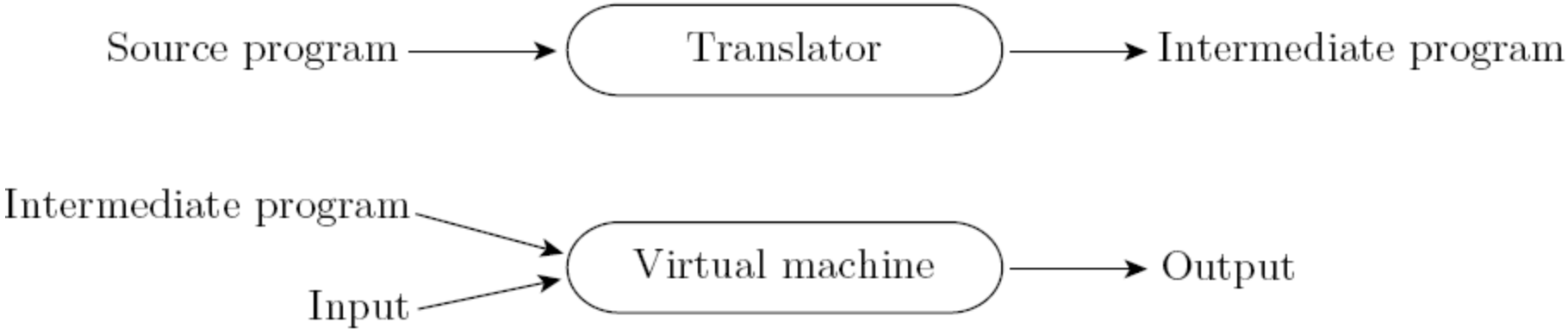
- Pure Interpretation
  - Interpreter stays around for the execution of the program
  - Interpreter is the locus of control during execution

Compilation vs. Interpretation (3/16)

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)
- Compilation
  - Better performance

Compilation vs. Interpretation (4/16)

- Common case is compilation or simple pre-processing, followed by interpretation
- Most language implementations include a mixture of both compilation and interpretation



Compilation vs. Interpretation (5/16)

- Note that compilation does NOT have to produce machine language for some sort of hardware
- Compilation is *translation* from one language into another, with full analysis of the meaning of the input
- Compilation entails semantic *understanding* of what is being processed; pre-processing does not
- A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

Compilation vs. Interpretation (6/16)

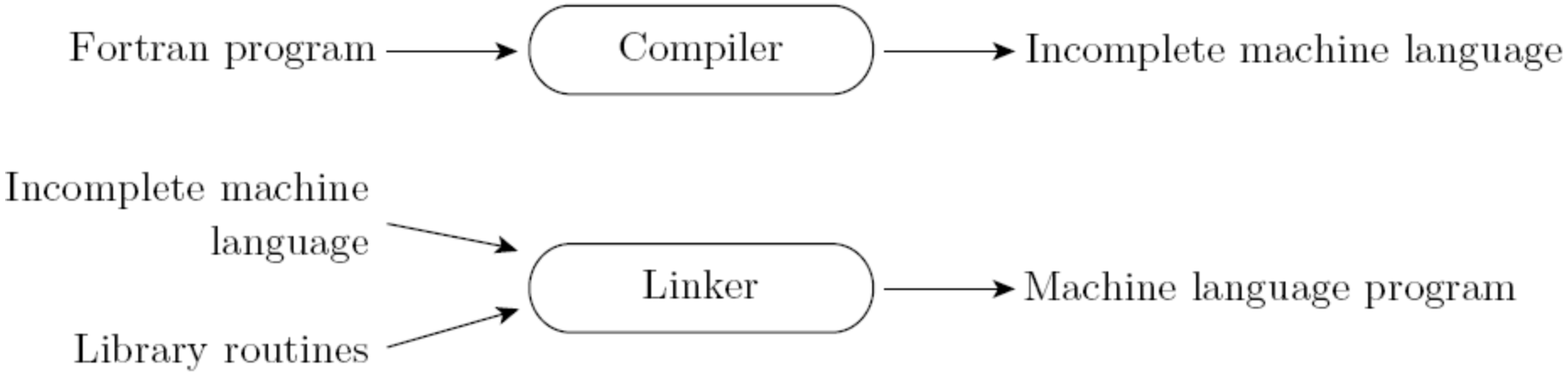
- Many compiled languages have interpreted pieces, e.g., formats in Fortran or C
- Most use “virtual instructions”
  - set operations in Pascal
  - string manipulation in Basic
- Some compilers produce nothing but virtual instructions, e.g., Pascal P-code, Java byte code, Microsoft COM+

Compilation vs. Interpretation (7/16)

- Implementation strategies:
  - Preprocessor
    - Removes comments and white space
    - Groups characters into *tokens* (keywords, identifiers, numbers, symbols)
    - Expands abbreviations in the style of a macro assembler
    - Identifies higher-level syntactic structures (loops, subroutines)

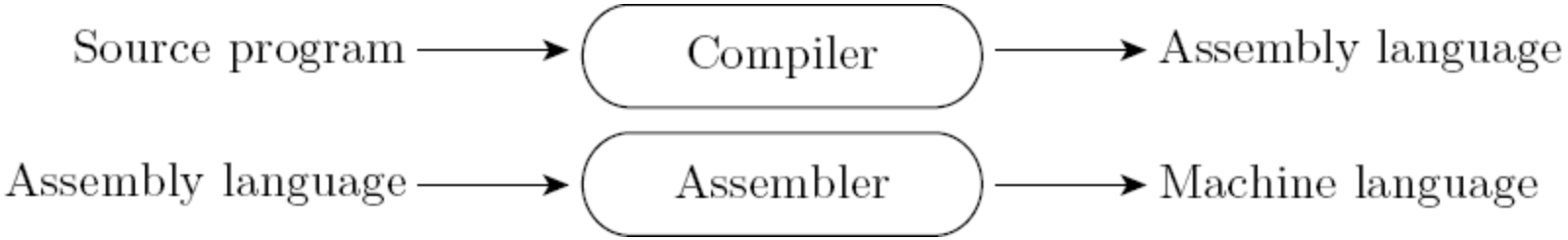
Compilation vs. Interpretation (8/16)

- Implementation strategies:
  - » Library of Routines and Linking
    - Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:



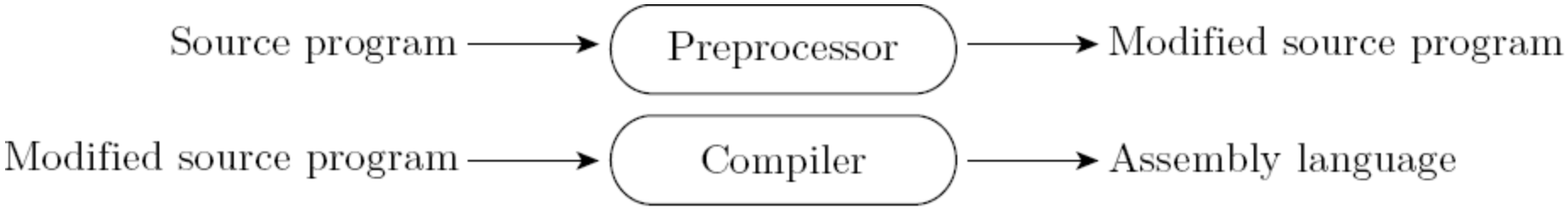
Compilation vs. Interpretation (9/16)

- Implementation strategies:
  - » Post-compilation Assembly
    - Facilitates debugging (assembly language easier for people to read)
    - Isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers)



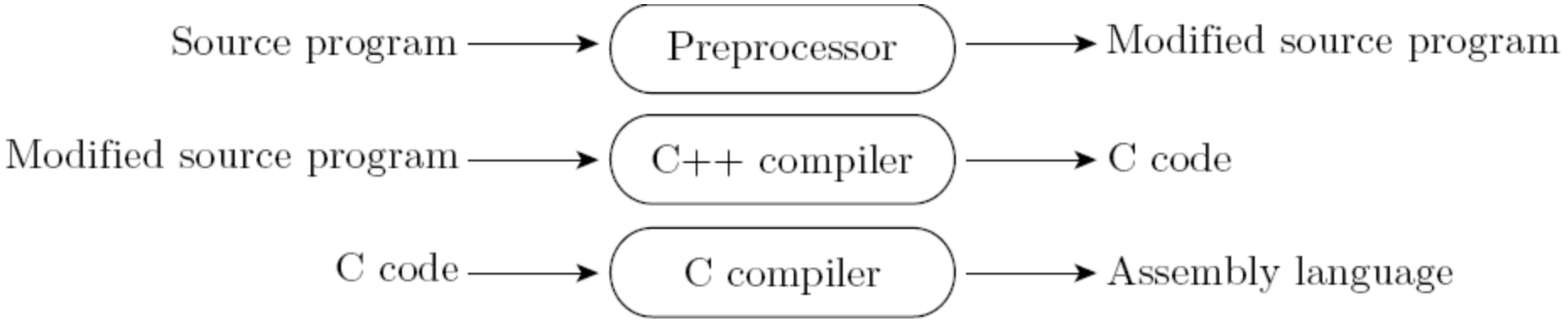
Compilation vs. Interpretation (10/16)

- Implementation strategies:
  - » The C Preprocessor (conditional compilation)
    - Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source



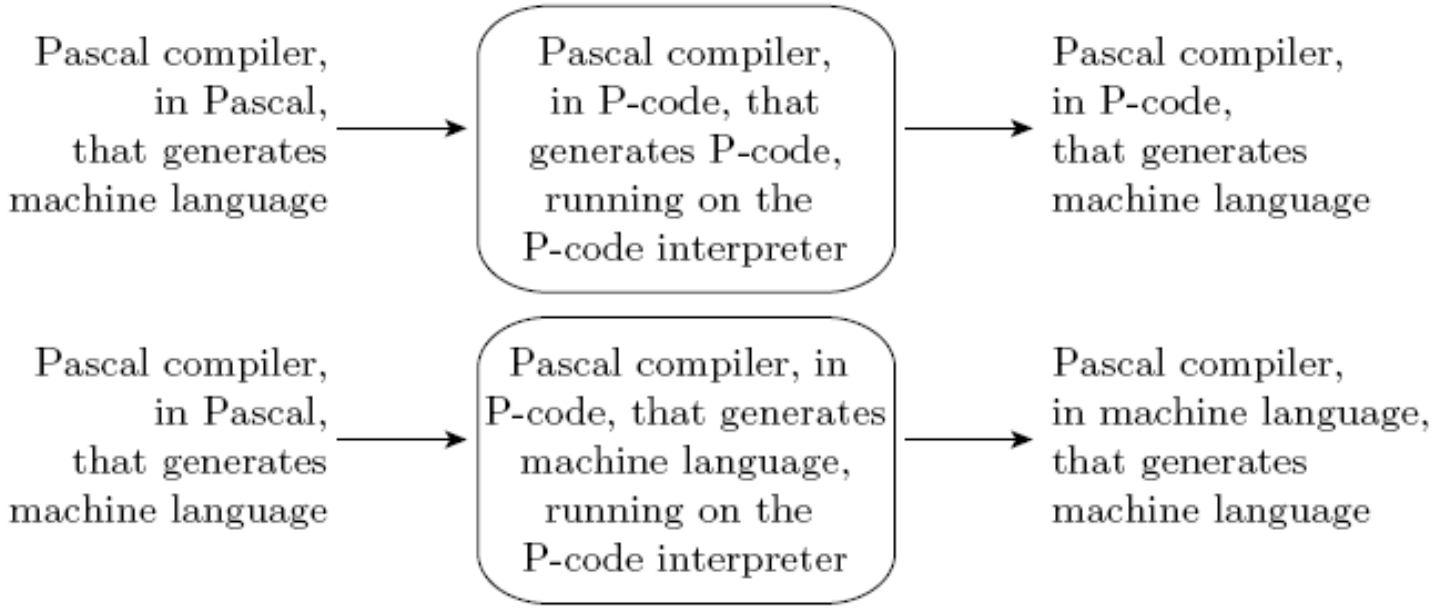
Compilation vs. Interpretation (11/16)

- Implementation strategies:
  - » Source-to-Source Translation (C++)
    - C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language:



Compilation vs. Interpretation (12/16)

- Implementation strategies:
  - » *Bootstrapping*



Compilation vs. Interpretation (13/16)

- Implementation strategies:
  - » Compilation of Interpreted Languages
    - The compiler generates code that makes assumptions about decisions that won't be finalized until runtime. If these assumptions are valid, the code runs very fast. If not, a dynamic check will revert to the interpreter.

Compilation vs. Interpretation (14/16)

- Implementation strategies:
  - » Dynamic and Just-in-Time Compilation
    - In some cases a programming system may deliberately delay compilation until the last possible moment.
      - – Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.
      - – The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs.
      - – The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

Compilation vs. Interpretation (15/16)

- Implementation strategies:
  - Microcode
    - Assembly-level instruction set is not implemented in hardware; it runs on an interpreter.
    - Interpreter is written in low-level instructions (*microcode* or *firmware*), which are stored in read-only memory and executed by the hardware.

Compilation vs. Interpretation (16/16)

- Compilers exist for some interpreted languages, but they aren't pure:
  - selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source.
  - Interpretation of parts of code, at least, is still necessary for reasons above.
- Unconventional compilers
  - text formatters
  - silicon compilers
  - query language processors

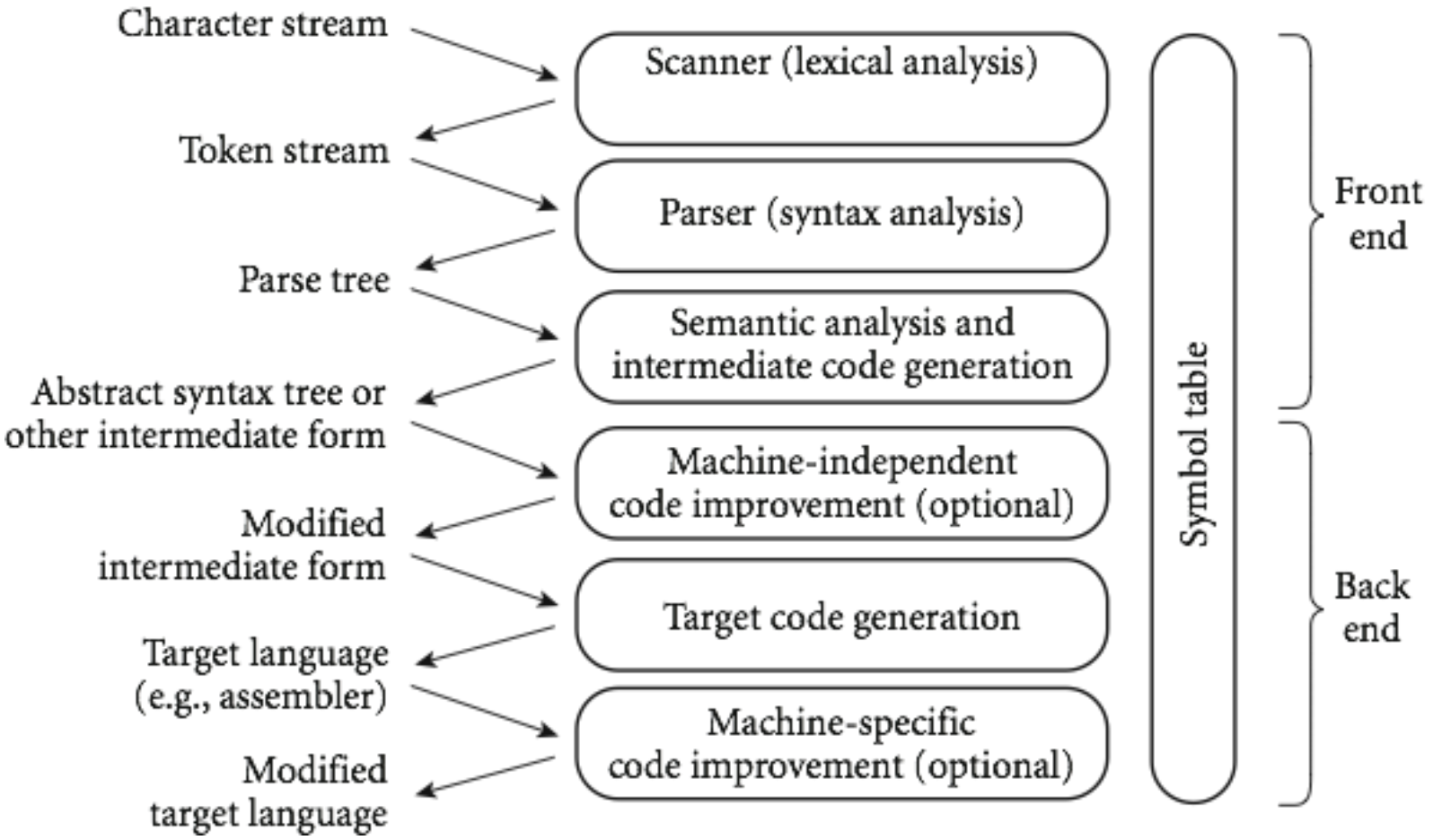
Programming Environment Tools

- Tools

Type	Unix examples
Editors	`vi`, `emacs`
Pretty printers	`cb`, `indent`
Pre-processors (esp. macros)	`cpp`, `m4`, `watfor`
Debuggers	`adb`, `sdb`, `dbx`, `gdb`
Style checkers	`lint`, `purify`
Module management	`make`
Version management	`sccs`, `rcs`
Assemblers	`as`
Link editors, loaders	`ld`, `ld-so`
Perusal tools	`More`, `less`, `od`, `nm`
Program cross-reference	`ctags`

An Overview of Compilation (1/15)

- Phases of Compilation



An Overview of Compilation (2/15)

- ***Scanning***:
  - divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
  - we can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
  - you can design a parser to take characters instead of tokens as input, but it isn't pretty
  - scanning is recognition of a *regular language*, e.g., via Deterministic Finite Automata (DFA)

An Overview of Compilation (3/15)

- ***Parsing*** is recognition of a *context-free language*, e.g., via Push Down Automata (PDA)
  - Parsing discovers the "context free" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)

An Overview of Compilation (4/15)

- ***Semantic analysis*** is the discovery of *meaning* in the program
  - The compiler actually does what is called STATIC semantic analysis. That's the meaning that can be figured out at compile time
  - Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's DYNAMIC semantics

An Overview of Compilation (5/15)

- ***Intermediate form*** (IF) done after semantic analysis (*if* the program passes all checks)
  - IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
  - They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
  - Many compilers actually move the code through more than one IF

An Overview of Compilation (6/15)

- ***Optimization*** takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - The term is a misnomer; we just *improve* code
  - The optimization phase is optional
- ***Code generation phase*** produces assembly language or (sometime) relocatable machine language

An Overview of Compilation (7/15)

- Certain ***machine-specific optimizations*** (use of special instructions or addressing modes, etc.) may be performed during or after ***target code generation***
- ***Symbol table***: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
  - » This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

An Overview of Compilation (8/15)

- Lexical and Syntax Analysis
  - GCD Program (in C)

```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```

An Overview of Compilation (9/15)

- Lexical and Syntax Analysis
  - GCD Program Tokens
    - Scanning (*lexical analysis*) and parsing recognize the structure of the program, groups characters into *tokens*, the smallest meaningful units of the program

```
int     main   (   )        {
int     i      =   getint   (   )   ,   j   =   getint   (   )   ;
while   (      i   !=       j   )   {
if      (      i   >        j   )   i   =   i   -       j   ;
else    j      =   j        -   i   ;
}
putint  (      i   )        ;
}
```

An Overview of Compilation (10/15)

- Lexical and Syntax Analysis
  - Context-Free Grammar and Parsing
    - Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents
    - Potentially recursive rules known as *context-free grammar* define the ways in which these constituents combine

An Overview of Compilation (11/15)

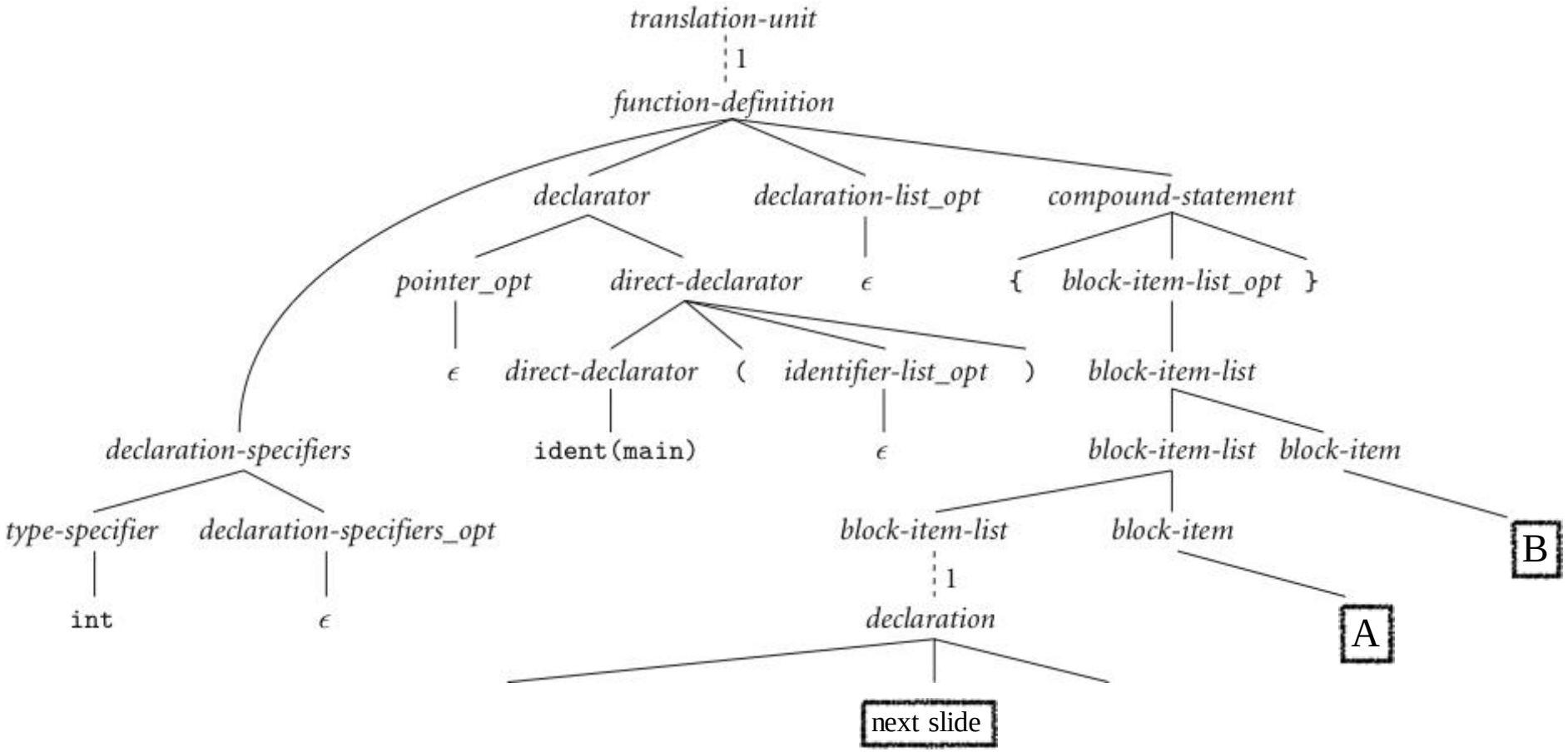
- Context-Free Grammar and Parsing
  - Example (`while` loop in C)

iteration-statement → *while (expression) statement*

statement, in turn, is often a list enclosed in braces:
statement → *compound-statement*
compound-statement → *{ block-item-list opt }*
where
block-item-list opt → *block-item-list*
or
block-item-list opt → ϵ
and
block-item-list → *block-item*
block-item-list → *block-item-list block-item*
block-item → *declaration*
block-item → *statement*

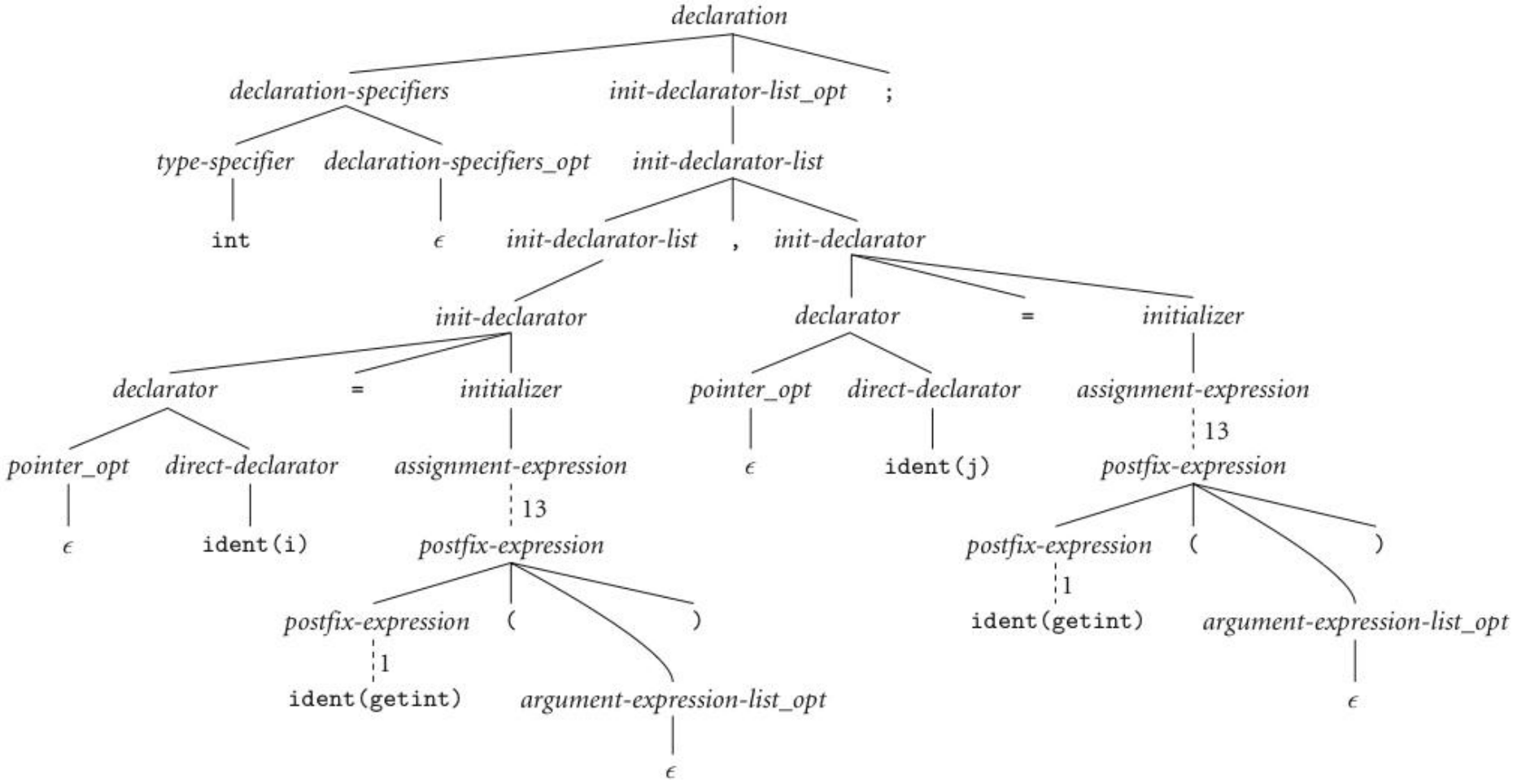
An Overview of Compilation (12/15)

- Context-Free Grammar and Parsing
  - » GCD Program Parse Tree

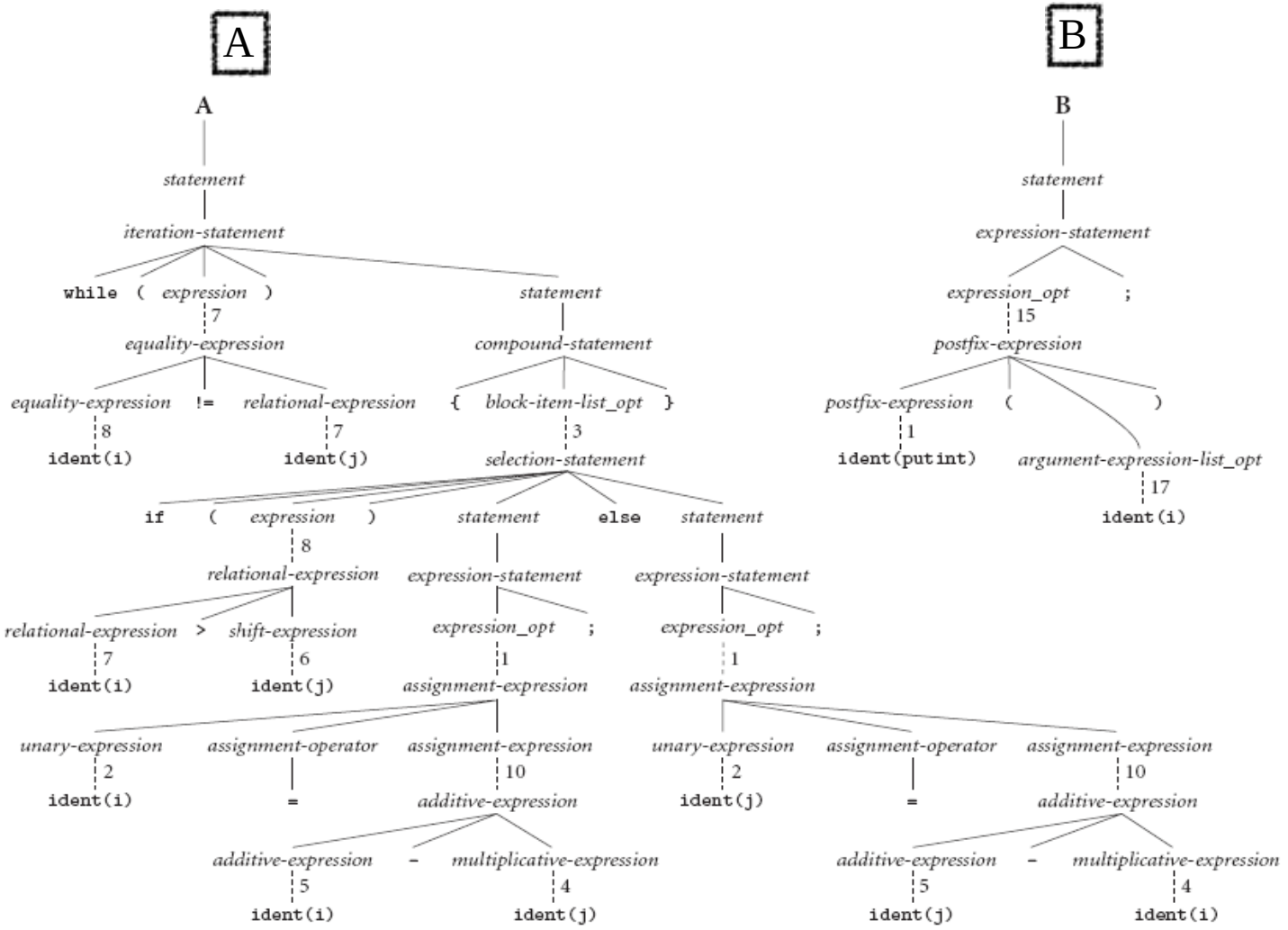


An Overview of Compilation (13/15)

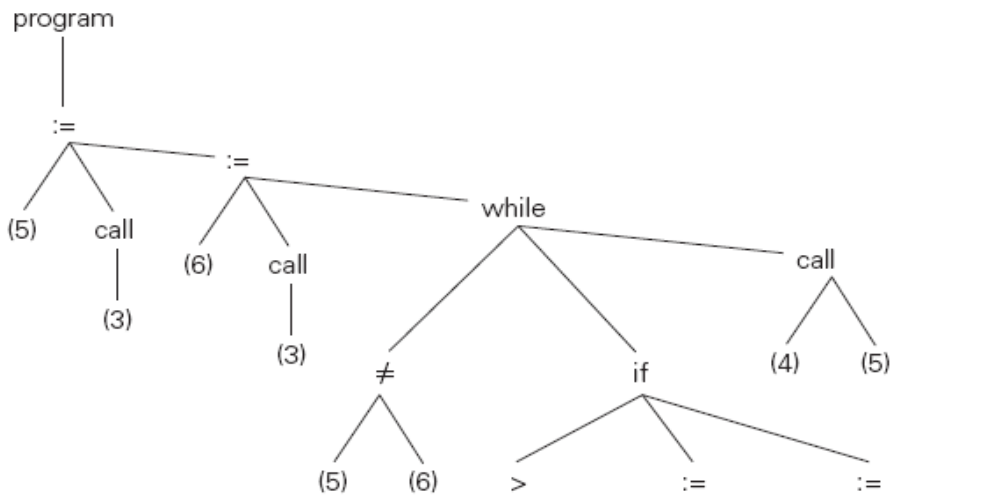
- Context-Free Grammar and Parsing (cont.)



- Context-Free Grammar and Parsing (cont.)



- Syntax Tree
  - » GCD Program Parse Tree



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Abstract Syntax Tree (1/2)

- Many non-terminals inside a parse tree are artifacts of the grammar
- Remember:

$$E ::= E + T \mid T$$

$$T ::= T * Id \mid Id$$

The parse tree for B * C can be written as

E(T(Id(B), Id(C)))

In constrast, an abstract syntax tree (AST) captures only those tree nodes that are necessary for representing the program

In the example:

T(Id(B), Id(C))

Consequently, many parsers really generate abstract syntax trees.

Abstract Syntax Tree (2/2)

- Another explanation for abstract syntax tree: It's a tree capturing only semantically relevant information for a program
  - i.e., omitting all formatting and comments
- Question 1: What is a concrete syntax tree?
- Question 2: When do I need a concrete syntax tree?

Scannerless Parsing

- Separating syntactic analysis into lexing and parsing helps performance. After all, regular expressions can be made very fast
- But it also limits language design choices. For example, it's very hard to compose different languages with separate lexers and parsers — think embedding SQL in JAVA
- Scannerless parsing integrates lexical analysis into the parser, making this problem more tractable.

Agenda

1. Instructor and Course Introduction
2. Introduction to Programming Languages
3. Programming Language Syntax
4. Conclusion

Programming Language Syntax - Sub-Topics

- Language Definition
- Syntax and Semantics
- Grammars
- The Chomsky Hierarchy
- Regular Expressions
- Regular Grammar Example
- Lexical Issues
- Context-Free Grammars
- Scanning
- Parsing
- LL Parsing
- LR Parsing

Language Definition

- Different users have different needs:
  - programmers: tutorials, reference manuals, programming guides (idioms)
  - implementors: precise operational semantics
  - verifiers: rigorous axiomatic or natural semantics
  - language designers and lawyers: all of the above
- Different levels of detail and precision
  - but none should be sloppy!

Syntax and Semantics

- Syntax refers to external representation:
  - Given some text, is it a well-formed program?
- Semantics denotes meaning:
  - Given a well-formed program, what does it mean?
  - Often depends on context
- The division is somewhat arbitrary
  - Note:
    - It is possible to fully describe the syntax and semantics of a programming language by syntactic means (e.g., Algol68 and W-grammars), but this is highly impractical
    - Typically use a grammar for the context-free aspects, and different method for the rest
  - Similar looking constructs in different languages often have subtly (or not-so-subtly) different meanings
  - Good syntax, unclear semantics: “Colorless green ideas sleep furiously”
  - Good semantics, poor syntax: “Me go swimming now, sorry bye”
  - In programming languages: syntax tells you what a well-formed program looks like. Semantic tells you relationship of output to input

Grammars (1/2)

- A grammar G is a tuple (Σ,N, S, δ)
  - N is the set of non-terminal symbols
  - S is the distinguished non-terminal: the root symbol
  - Σ is the set of terminal symbols (alphabet)
  - δ is the set of rewrite rules (productions) of the form:

    ABC. . . ::= XYZ . . .

    where A,B,C,D,X,Y, Z are terminals and non terminals
  - The language is the set of sentences containing only terminal symbols that can be generated by applying the rewriting rules starting from the root symbol (let's call such sentences strings)

Grammars (2/2)

- Consider the following grammar G:
  - N = {S;X; Y}
  - S = S
  - Σ = {a; b; c}
  - δ consists of the following rules:
    - S -> b
    - S -> XbY
    - X -> a
    - X -> aX
    - Y -> c
    - Y ->Y c
  - Some sample derivations:
    - S -> b
    - S -> XbY -> abY -> abc
    - S -> XbY -> aXbY -> aaXbY -> aaabY -> aaabc

The Chomsky Hierarchy

- Regular grammars (Type 3)
  - all productions can be written in the form: N ::= TN
  - one non-terminal on left side; at most one on right
- Context-free grammars (Type 2)
  - all productions can be written in the form: N ::= XYZ
  - one non-terminal on the left-hand side; mixture on right
- Context-sensitive grammars (Type 1)
  - number of symbols on the left is no greater than on the right
  - no production shrinks the size of the sentential form
- Type-0 grammars
  - no restrictions

Regular Expressions (1/3)

- An alternate way of describing a regular language is with regular expressions
  - We say that a regular expression R denotes the language [[R]]
  - Recall that a language is a set of strings
  - Basic regular expressions:
    - » Є denotes Ø
    - » a character x, where x Є Σ, denotes {x}
    - » (sequencing) a sequence of two regular expressions *RS* denotes
    - » {αβ | α Є [[R]], β Є [[S]]}
    - » (alternation) R|S denotes [[R]] U [[S]]
    - » (Kleene star) R* denotes the set of strings which are concatenations of zero or more strings from [[R]]
    - » parentheses are used for grouping
    - » Shorthands:
      - $R^? \equiv Є \mid R$
      - $R^+ \equiv RR^*$

Regular Expressions (2/3)

- A regular expression is one of the following:
  - A character
  - The empty string, denoted by $\sum$
  - Two regular expressions concatenated
  - Two regular expressions separated by | (i.e., or)
  - A regular expression followed by the Kleene star (concatenation of zero or more strings)

Regular Expressions (3/3)

- Numerical literals in Pascal may be generated by the following:

$$
\begin{aligned}
\mathit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{unsigned\_integer} &\longrightarrow \mathit{digit}\ \mathit{digit}* \\
\mathit{unsigned\_number} &\longrightarrow \mathit{unsigned\_integer}\ ((\texttt{.}\ \mathit{unsigned\_integer}) \mid \epsilon) \\
&\qquad (((\texttt{e} \mid \texttt{E})(\texttt{+} \mid \texttt{-} \mid \epsilon)\ \mathit{unsigned\_integer}) \mid \epsilon)
\end{aligned}
$$

Regular Grammar Example

- A grammar for floating point numbers:
  - Float ::= Digits | Digits . Digits
  - Digits ::= Digit | Digit Digits
  - Digit ::= 0|1|2|3|4|5|6|7|8|9
- A regular expression for floating point numbers:
  - $(0|1|2|3|4|5|6|7|8|9)^{+}(.(0|1|2|3|4|5|6|7|8|9)^{+})^{?}$
- Perl offer some shorthands:
  - [0 -9]+(\.[0 -9]+)?

  or
  - \d +(\.\ d+)?

Lexical Issues

Lexical: formation of words or tokens

- Tokens are the basic building blocks of programs:
  - keywords (begin, end, while).
  - identifiers (myVariable, yourType)
  - numbers (137, 6:022e23)
  - symbols (+, □)
  - string literals (“Hello world”)
- Described (mainly) by regular grammars
- Terminals are characters. Some choices:
  - character set: ASCII, Latin-1, ISO646, Unicode, etc.
  - is case significant?
- Is indentation significant?
  - Python, Occam, Haskell

Example: identifiers

Id ::= Letter IdRest

IdRest ::= Є | Letter IdRest | Digit IdRest

Missing from above grammar: limit of identifier length

Other issues: international characters, case-sensitivity, limit of identifier length

Context-Free Grammars (1/7)

- BNF: notation for context-free grammars
  - (BNF = Backus-Naur Form) Some conventional abbreviations:
    - alternation: Symb ::= Letter | Digit
    - repetition: Id ::= Letter {Symb}

      or we can use a Kleene star: Id ::= Letter Symb*

      for one or more repetitions: Int ::= $Digit^{+}$
    - option: Num ::= $Digit^{+}$[. Digit*]
- abbreviations do not add to expressive power of grammar
- need convention for meta-symbols – what if "|" is in the language?

Context-Free Grammars (2/7)

- The notation for context-free grammars (CFG) is sometimes called Backus-Naur Form (BNF)
- A CFG consists of
  - A set of *terminals* *T*
  - A set of *non-terminals* *N*
  - A *start symbol* *S* (a non-terminal)
  - A set of *productions*

Context-Free Grammars (3/7)

- Expression grammar with precedence and associativity

1. *expr* ⟶ *term* | *expr add_op term*
2. *term* ⟶ *factor* | *term mult_op factor*
3. *factor* ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* ⟶ `+` | `-`
5. *mult_op* ⟶ `*` | `/`

Context-Free Grammars (4/7)

- A parse tree describes the grammatical structure of a sentence
  - root of tree is root symbol of grammar
  - leaf nodes are terminal symbols
  - internal nodes are non-terminal symbols
  - an internal node and its descendants correspond to some production for that non terminal
  - top-down tree traversal represents the process of generating the given sentence from the grammar
  - construction of tree from sentence is parsing

Context-Free Grammars (5/7)

- Ambiguity:
  - If the parse tree for a sentence is not unique, the grammar is ambiguous:

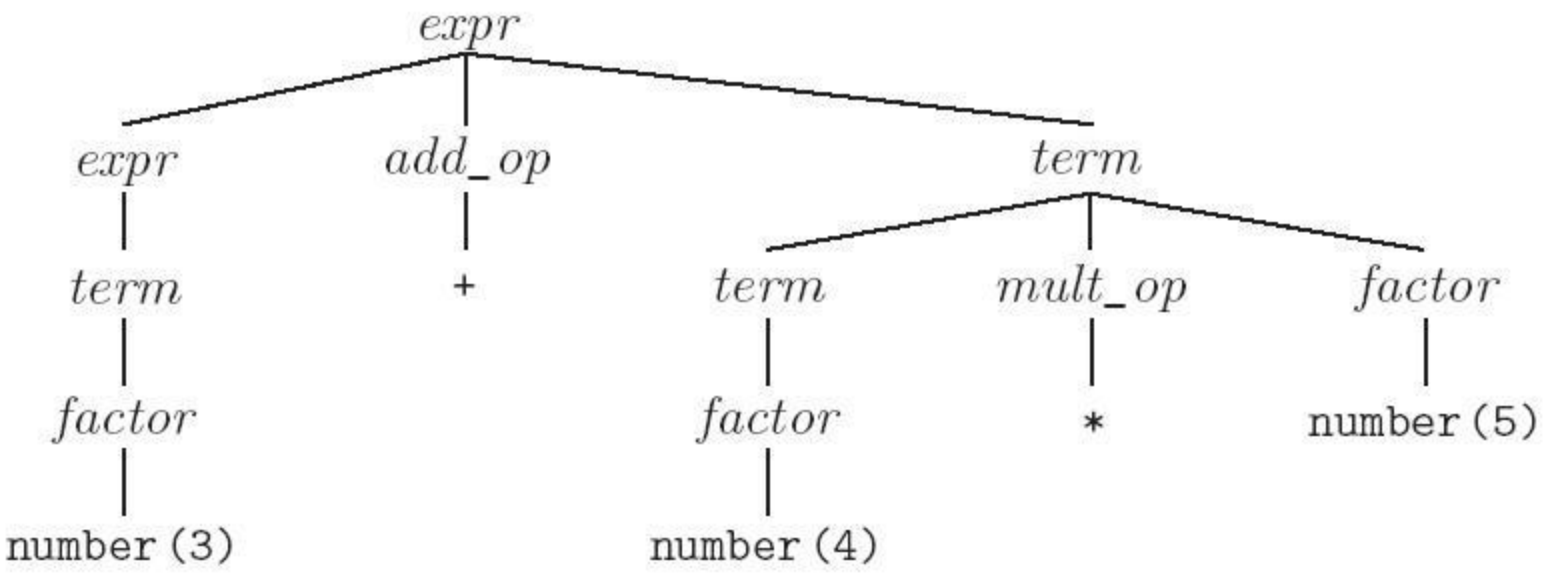
    E ::= E + E | E * E | Id
  - Two possible parse trees for "A + B * C":
    - ((A + B) * C)
    - (A + (B * C))
  - One solution: rearrange grammar:

    E ::= E + T | T

    T ::= T * Id | Id
  - Harder problems – disambiguate these (courtesy of Ada):
    - function call ::= name (expression list)
    - indexed component ::= name (index list)
    - type conversion ::= name (expression)

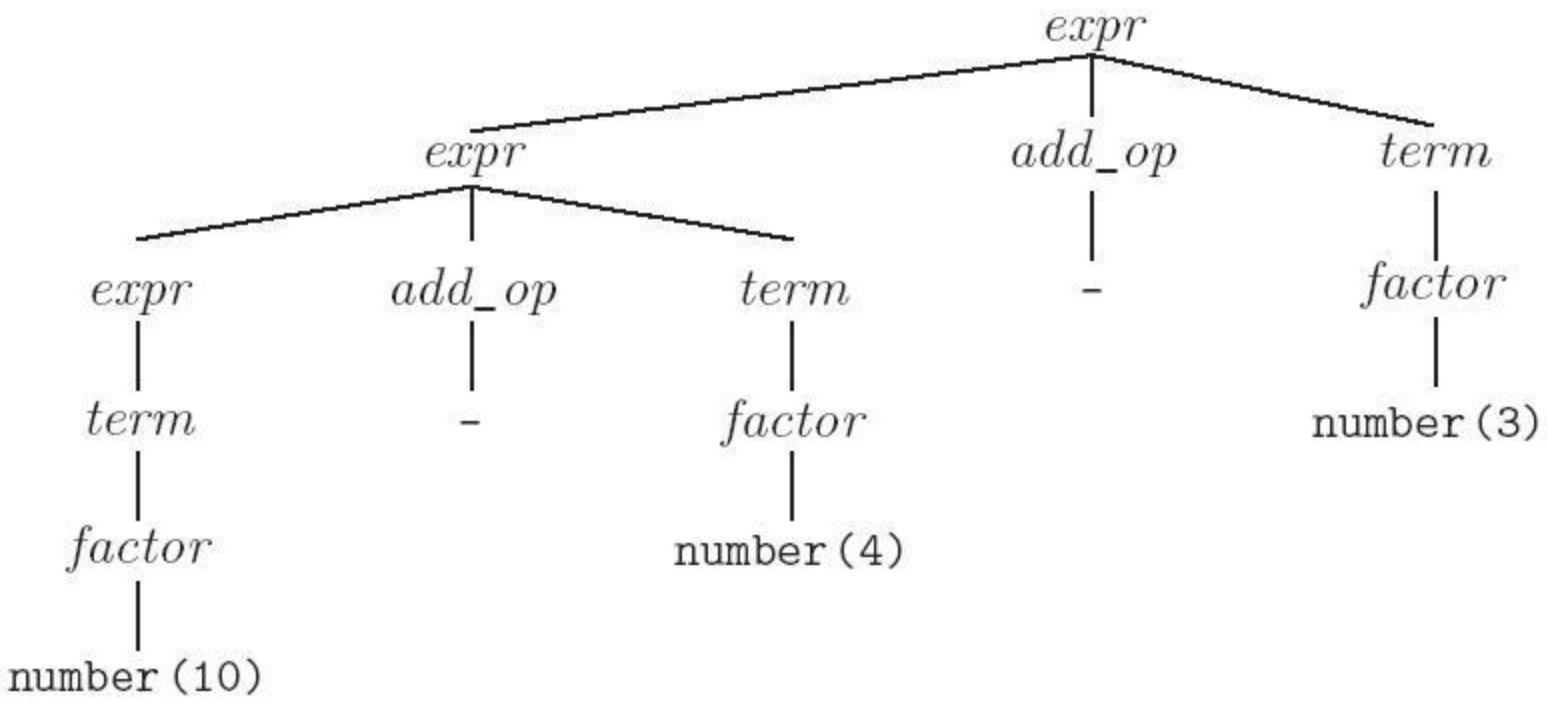
Context-Free Grammars (6/7)

- Parse tree for expression grammar (with precedence) for `3 + 4 * 5`



Context-Free Grammars (7/7)

- Parse tree for expression grammar (with left associativity) for `10 - 4 - 3`



Scanning (1/11)

- Recall scanner is responsible for
  - tokenizing source
  - removing comments
  - (often) dealing with *pragmas* (i.e., significant comments)
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages

Scanning (2/11)

- Suppose we are building an ad-hoc (hand-written) scanner for Pascal:
  - » We read the characters one at a time with look-ahead
- If it is one of the one-character tokens

  `{ ( ) [ ] < > , ; = + - etc }`

  we announce that token
- If it is a ., we look at the next character
  - » If that is a dot, we announce .
  - » Otherwise, we announce . and reuse the look-ahead

Scanning (3/11)

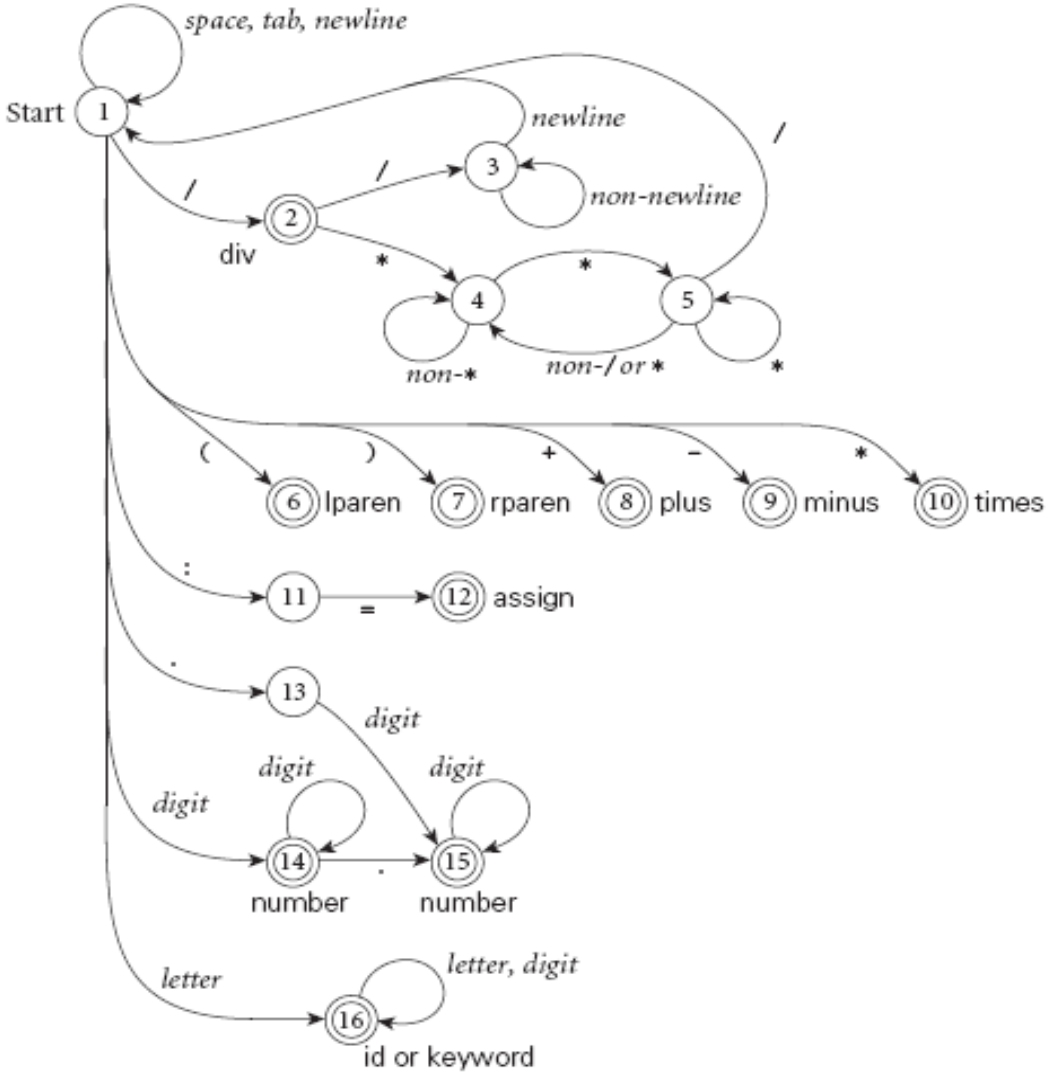
- If it is a `<`, we look at the next character
  - if that is a `=` we announce `<=`
  - otherwise, we announce `<` and reuse the look-ahead, etc
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore
  - then we check to see if it is a reserved word

Scanning (4/11)

- If it is a digit, we keep reading until we find a non-digit
  - if that is not a . we announce an integer
  - otherwise, we keep looking for a real number
  - if the character after the . is not a digit we announce an integer and reuse the . and the look-ahead

Scanning (5/11)

- Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton



Scanning (6/11)

- This is a deterministic finite automaton (DFA)
  - Lex, scangen, etc. build these things automatically from a set of regular expressions
  - Specifically, they construct a machine that accepts the language

```
identifier | int const
| real const | comment | symbol
| ...
```

Scanning (7/11)

- We run the machine over and over to get one token after another
  - » Nearly universal rule:
    - always take the longest possible token from the input
      thus foobar is foobar and never f or foo or foob
    - more to the point, `3.14159` is a real const and never `3`, `.`, and `14159`
- Regular expressions "generate" a regular language; DFAs "recognize" it

Scanning (8/11)

- Scanners tend to be built three ways
  - ad-hoc
  - semi-mechanical pure DFA
    (usually realized as nested case statements)
  - table-driven DFA
- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close

Scanning (9/11)

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - » though it's often easier to use perl, awk, sed
  - » for details (see textbook's Figure 2.11)
- Table-driven DFA is what lex and scangen produce
  - » lex (flex) in the form of C code
  - » scangen in the form of numeric tables and a separate driver (for details see textbook's Figure 2.12)

Scanning (10/11)

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
  - » the next character will generally need to be saved for the next token
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
  - » In Pascal, for example, when you have a 3 and you a see a dot
    - do you proceed (in hopes of getting 3.14)? or
    - do you stop (in fear of getting 3..5)?

Scanning (11/11)

- In messier cases, you may not be able to get by with any fixed amount of look-ahead.In Fortran, for example, we have

```
DO 5 I = 1,25   loop
DO 5 I = 1.25   assignment
```

- Here, we need to remember we were in a potentially final state, and save enough information that we can back up to it, if we get stuck later

Parsing (1/7)

- Terminology:
  - context-free grammar (CFG)
  - symbols
    - terminals (tokens)
    - non-terminals
  - production
  - derivations (left-most and right-most - canonical)
  - parse trees
  - sentential form

Parsing (2/7)

- By analogy to RE and DFAs, a context-free grammar (CFG) is a *generator* for a context-free language (CFL)
  - a parser is a language *recognizer*
- There is an infinite number of grammars for every context-free language
  - not all grammars are created equal, however

Parsing (3/7)

- It turns out that for any CFG we can create a parser that runs in O(n^3) time
- There are two well-known parsing algorithms that permit this
  - Early's algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- O(n^3) time is clearly unacceptable for a parser in a compiler - too slow

Parsing (4/7)

- Fortunately, there are large classes of grammars for which we can build parsers that run in linear time
  - The two most important classes are called **LL** and **LR**
- LL stands for 'Left-to-right, Leftmost derivation'.
- LR stands for 'Left-to-right, Rightmost derivation’

Parsing (5/7)

- LL parsers are also called 'top-down', or 'predictive' parsers & LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail of the differences between them

Parsing (6/7)

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the *prefix property* (no valid string is a prefix of another valid string) has an LR(0) grammar

Parsing (7/7)

- You commonly see LL or LR (or whatever) written with a number in parentheses after it
  - This number indicates how many tokens of look-ahead are required in order to parse
  - Almost all real compilers use one token of look-ahead
- The expression grammar (with precedence and associativity) you saw before is LR(1), but not LL(1)

LL Parsing (1/23)

- Here is an LL(1) grammar (Fig 2.15):

```
1.  program        → stmt list $$$
2.  stmt_list      → stmt stmt_list
3.                 | ε
4.  stmt     →     id := expr
5.                 | read id
6.                 | write expr
7.  expr     →     term term_tail
8.  term_tail → add op term term_tail
9.                 | ε
```

LL Parsing (2/23)

- LL(1) grammar (continued)

```
10. term    →     factor fact_tailt
11. fact_tail → mult_op fact fact_tail
12.                | ε
13.  factor   →    ( expr )
14.                | id
15.                | number
16.  add_op →     +
17.                | -
18.  mult_op → *
19.                | /
```

LL Parsing (3/23)

- Like the bottom-up grammar, this one captures associativity and precedence, but most people don't find it as pretty
  - for one thing, the operands of a given operator aren't in a RHS together!
  - however, the simplicity of the parsing algorithm makes up for this weakness
- How do we parse a string with this grammar?
  - by building the parse tree incrementally

LL Parsing (4/23)

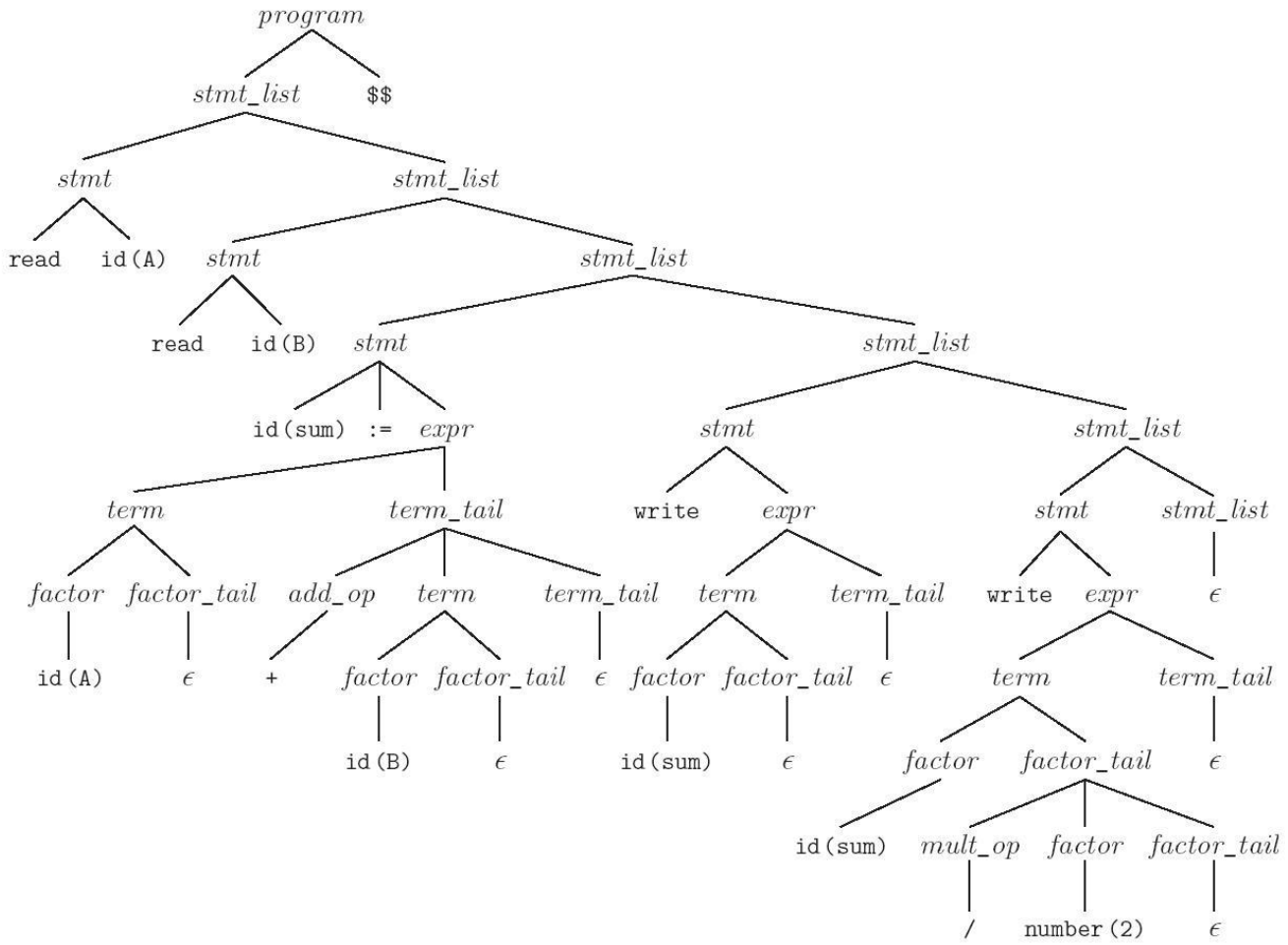
- Example (average program)

```
read A
read B
sum := A + B
write sum
write sum / 2
```

- We start at the top and predict needed productions on the basis of the current left-most non-terminal in the tree and the current input token

LL Parsing (5/23)

- Parse tree for the average program (Figure 2.17)



LL Parsing (6/23)

- Table-driven LL parsing: you have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token. The actions are
  (1) match a terminal
  (2) predict a production
  (3) announce a syntax error

LL Parsing (7/23)

- LL(1) parse table for parsing for calculator language

Top-of-stack nonterminal	Current input token											
	id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

LL Parsing (8/23)

- To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack
  - » for details see Figure 2.20
- The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program
  - » what you *predict* you will see

- Problems trying to make a grammar LL(1)
  - left recursion
    - example:

```
id_list → id | id_list , id
          equivalently
id_list → id id_list_tail
id_list_tail → , id id_list_tail
               | epsilon
```

    - we can get rid of all left recursion mechanically in any grammar

LL Parsing (10/23)

- Problems trying to make a grammar LL(1)
  - » common prefixes: another thing that LL parsers can't handle
    - solved by "left-factoring"
    - example:

```
stmt → id := expr | id ( arg_list )
          equivalently
stmt → id id_stmt_tail
id_stmt_tail → := expr
               | ( arg_list)
```

    - we can eliminate left-factor mechanically

LL Parsing (11/23)

- Note that eliminating left recursion and common prefixes does NOT make a grammar LL
  - there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine
  - the few that arise in practice, however, can generally be handled with kludges

LL Parsing (12/23)

- Problems trying to make a grammar LL(1)
  - the"dangling else" problem prevents grammars from being LL(1) (or in fact LL(k) for any k)
  - the following natural grammar fragment is ambiguous (Pascal)

```
stmt → if cond then_clause else_clause
     | other_stuff
then_clause → then stmt
else_clause → else stmt
            | epsilon
```

LL Parsing (13/23)

- Consider: S ::= if E then S

  S ::= if E then S else S

- The sentence

  if E1 then if E2 then S1 else S2

  is ambiguous (Which then does else S2 match?)

- Solutions:
  - Pascal rule: else matches most recent if
  - grammatical solution: different productions for balanced and unbalanced
  - if-statements
  - grammatical solution: introduce explicit end-marker
- The general ambiguity problem is unsolvable

LL Parsing (14/23)

- The less natural grammar fragment can be parsed bottom-up but not top-down

```
stmt → balanced_stmt | unbalanced_stmt
balanced_stmt → if cond then balanced_stmt
                    else balanced_stmt
              | other_stuff
unbalanced_stmt → if cond then stmt
              | if cond then balanced_stmt
                    else unbalanced_stmt
```

LL Parsing (15/23)

- The usual approach, whether top-down OR bottom-up, is to use the ambiguous grammar together with a *disambiguating rule* that says
  - else goes with the closest then or
  - more generally, the first of two possible productions is the one to predict (or reduce)

LL Parsing (16/23)

- Better yet, languages (since Pascal) generally employ explicit end-markers, which eliminate this problem
- In Modula-2, for example, one says:

```
if A = B then
        if C = D then E := F end
else
        G := H
end
```

- Ada says 'end if'; other languages say 'fi'

LL Parsing (17/23)

- One problem with end markers is that they tend to bunch up. In Pascal you say

```
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...;
```

- With end markers this becomes

```
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...;
end; end; end; end;
```

LL Parsing (18/23)

- The algorithm to build predict sets is tedious (for a "real" sized grammar), but relatively simple
- It consists of three stages:
  - (1) compute FIRST sets for symbols
  - (2) compute FOLLOW sets for non-terminals (this requires computing FIRST sets for some *strings*)
  - (3) compute predict sets or table for all productions

LL Parsing (19/23)

- It is conventional in general discussions of grammars to use
  - lower case letters near the beginning of the alphabet for terminals
  - lower case letters near the end of the alphabet for strings of terminals
  - upper case letters near the beginning of the alphabet for non-terminals
  - upper case letters near the end of the alphabet for arbitrary symbols
  - greek letters for arbitrary strings of symbols

LL Parsing (20/23)

- Algorithm First/Follow/Predict:
  - FIRST(α) == {a : α →* a β}
    ∪ (if α =>* ε THEN {ε} ELSE NULL)
  - FOLLOW(A) == {a : S →+ α A a β}
    ∪ (if S →* α A THEN {ε} ELSE NULL)
  - Predict (A → X_1 ... X_m) == (FIRST (X_1 ... X_m) - {ε}) ∪ (if X_1, ..., X_m →* ε then FOLLOW (A) ELSE NULL)
- Details following…

LL Parsing (21/23)

$program \longrightarrow stmt\_list$ \$\$	\$\$ $\in$ FOLLOW($stmt\_list$), $\epsilon \in$ FOLLOW(\$\$), and $\epsilon \in$ FOLLOW($program$)
$stmt\_list \longrightarrow stmt\ stmt\_list$	
$stmt\_list \longrightarrow \epsilon$	$\epsilon \in$ FIRST($stmt\_list$)
$stmt \longrightarrow$ `id :=` $expr$	`id` $\in$ FIRST($stmt$) and `:=` $\in$ FOLLOW(`id`)
$stmt \longrightarrow$ `read id`	`read` $\in$ FIRST($stmt$) and `id` $\in$ FOLLOW(`read`)
$stmt \longrightarrow$ `write` $expr$	`write` $\in$ FIRST($stmt$)
$expr \longrightarrow term\ term\_tail$	
$term\_tail \longrightarrow add\_op\ term\ term\_tail$	
$term\_tail \longrightarrow \epsilon$	$\epsilon \in$ FIRST($term\_tail$)
$term \longrightarrow factor\ factor\_tail$	
$factor\_tail \longrightarrow mult\_op\ factor\ factor\_tail$	
$factor\_tail \longrightarrow \epsilon$	$\epsilon \in$ FIRST($factor\_tail$)
$factor \longrightarrow$ `(` $expr$ `)`	`(` $\in$ FIRST($factor$) and `)` $\in$ FOLLOW($expr$)
$factor \longrightarrow$ `id`	`id` $\in$ FIRST($factor$)
$factor \longrightarrow$ `number`	`number` $\in$ FIRST($factor$)
$add\_op \longrightarrow$ `+`	`+` $\in$ FIRST(add_op)
$add\_op \longrightarrow$ `-`	`-` $\in$ FIRST(add_op)
$mult\_op \longrightarrow$ `*`	`*` $\in$ FIRST($mult\_op$)
$mult\_op \longrightarrow$ `/`	`/` $\in$ FIRST($mult\_op$)

Figure 2.21: **"Obvious" facts about the LL(1) calculator grammar.**

LL Parsing (22/23)

FIRST

program {id, read, write, $$}
stmt_list {id, read, write, ϵ}
stmt {id, read, write}
expr {(, id, number}
term_tail {+, -, ϵ}
term {(, id, number}
factor_tail {*, /, ϵ}
factor {(, id, number}
add_op {+, -}
mult_op {*, /}

Also note that FIRST(a) = {a} $\forall$ tokens a.

FOLLOW

id {+, -, *, /,), :=, id, read, write, $$}
number {+, -, *, /,), id, read, write, $$}
read {id}
write {(, id, number}
({(, id, number}
) {+, -, *, /,), id, read, write, $$}
:= {(, id, number}
+ {(, id, number}
- {(, id, number}
* {(, id, number}
/ {(, id, number}
$$ {ϵ}
program {ϵ}
stmt_list {$$}
stmt {id, read, write, $$}
expr {), id, read, write, $$}
term_tail {), id, read, write, $$}
term {+, -,), id, read, write, $$}
factor_tail {+, -,), id, read, write, $$}
factor {+, -, *, /,), id, read, write, $$}
add_op {(, id, number}
mult_op {(, id, number}

PREDICT

1 *program* $\longrightarrow$ *stmt_list* $$ {id, read, write, $$}
2 *stmt_list* $\longrightarrow$ *stmt stmt_list* {id, read, write}
3 *stmt_list* $\longrightarrow \epsilon$ {$$}
4 *stmt* $\longrightarrow$ id := *expr* {id}
5 *stmt* $\longrightarrow$ read id {read}
6 *stmt* $\longrightarrow$ write *expr* {write}
7 *expr* $\longrightarrow$ *term term_tail* {(, id, number}
8 *term_tail* $\longrightarrow$ *add_op term term_tail* {+, -}
9 *term_tail* $\longrightarrow \epsilon$ {), id, read, write, $$}
10 *term* $\longrightarrow$ *factor factor_tail* {(, id, number}
11 *factor_tail* $\longrightarrow$ *mult_op factor factor_tail* {*, /}
12 *factor_tail* $\longrightarrow \epsilon$ {+, -,), id, read, write, $$}
13 *factor* $\longrightarrow$ (*expr*) {(}
14 *factor* $\longrightarrow$ id {id}
15 *factor* $\longrightarrow$ number {number}
16 *add_op* $\longrightarrow$ + {+}
17 *add_op* $\longrightarrow$ - {-}
18 *mult_op* $\longrightarrow$ * {*}
19 *mult_op* $\longrightarrow$ / {/}

Figure 2.22: FIRST, FOLLOW, **and** PREDICT **sets for the calculator language.**

LL Parsing (23/23)

- If any token belongs to the predict set of more than one production with the same LHS, then the grammar is not LL(1)
- A conflict can arise because
  - the same token can begin more than one RHS
  - it can begin one RHS and can also appear *after* the LHS in some valid program, and one possible RHS is $\sum$

LR Parsing (1/11)

- LR parsers are almost always table-driven:
  - like a table-driven LL parser, an LR parser uses a big loop in which it repeatedly inspects a two-dimensional table to find out what action to take
  - unlike the LL parser, however, the LR driver has non-trivial state (like a DFA), and the table is indexed by current input token and current state
  - the stack contains a record of what has been seen SO FAR (NOT what is expected)

LR Parsing (2/11)

- A scanner is a DFA
  - it can be specified with a state diagram
- An LL or LR parser is a PDA
  - Early's & CYK algorithms do NOT use PDAs
  - a PDA can be specified with a state diagram and a stack
    - the state diagram looks just like a DFA state diagram, except the arcs are labeled with <input symbol, top-of-stack symbol> pairs, and in addition to moving to a new state the PDA has the option of pushing or popping a finite number of symbols onto/off the stack

LR Parsing (3/11)

- An LL(1) PDA has only one state!
  - well, actually two; it needs a second one to accept with, but that's all (it's pretty simple)
  - all the arcs are self loops; the only difference between them is the choice of whether to push or pop
  - the final state is reached by a transition that sees EOF on the input and the stack

LR Parsing (4/11)

- An SLR/LALR/LR PDA has multiple states
  - it is a "recognizer," not a "predictor"
  - it builds a parse tree from the bottom up
  - the states keep track of which productions we *might* be in the middle
- The parsing of the Characteristic Finite State Machine (CFSM) is based on
  - Shift
  - Reduce

LR Parsing (5/11)

- To illustrate LR parsing, consider the grammar (Figure 2.24, Page 73):

```
1. program          → stmt list $$$
2. stmt_list → stmt_list stmt
3.             | stmt
4. stmt        → id := expr
5.             | read id
6.             | write expr
7. expr   →    term
8.             | expr add op term
```

- LR grammar (continued):

```
9. term   →     factor
10.             | term mult_op factor
11.  factor →( expr )
12.             | id
13.             | number
14.  add op → +
15.             | -
16.  mult op → *
17.             | /
```

LR Parsing (7/11)

- This grammar is SLR(1), a particularly nice class of bottom-up grammar
  - it isn't exactly what we saw originally
  - we've eliminated the epsilon production to simplify the presentation
- For details on the table driven SLR(1) parsing please note the following slides

LR Parsing (8/11)

	State	Transitions
0.	*program* ⟶ . *stmt_list* $$	on *stmt_list* shift and goto 2
	stmt_list ⟶ . *stmt_list stmt*	
	stmt_list ⟶ . *stmt*	on *stmt* shift and reduce (pop 1 state, push
	stmt ⟶ . id := *expr*	on id shift and goto 3
	stmt ⟶ . read id	on read shift and goto 1
	stmt ⟶ . write *expr*	on write shift and goto 4
1.	*stmt* ⟶ read . id	on id shift and reduce (pop 2 states, push
2.	*program* ⟶ *stmt_list* . $$	on $$ shift and reduce (pop 2 states, push
	stmt_list ⟶ *stmt_list* . *stmt*	on *stmt* shift and reduce (pop 2 states, pus
	stmt ⟶ . id := *expr*	on id shift and goto 3
	stmt ⟶ . read id	on read shift and goto 1
	stmt ⟶ . write *expr*	on write shift and goto 4
3.	*stmt* ⟶ id . := *expr*	on := shift and goto 5
4.	*stmt* ⟶ write . *expr*	on *expr* shift and goto 6
	expr ⟶ . *term*	on *term* shift and goto 7
	expr ⟶ . *expr add_op term*	
	term ⟶ . *factor*	on *factor* shift and reduce (pop 1 state, pu
	term ⟶ . *term mult_op factor*	
	factor ⟶ . (*expr*)	on (shift and goto 8
	factor ⟶ . id	on id shift and reduce (pop 1 state, push
	factor ⟶ . number	on number shift and reduce (pop 1 state, p
5.	*stmt* ⟶ id := . *expr*	on *expr* shift and goto 9
	expr ⟶ . *term*	on *term* shift and goto 7
	expr ⟶ . *expr add_op term*	
	term ⟶ . *factor*	on *factor* shift and reduce (pop 1 state, pu
	term ⟶ . *term mult_op factor*	
	factor ⟶ . (*expr*)	on (shift and goto 8
	factor ⟶ . id	on id shift and reduce (pop 1 state, push
	factor ⟶ . number	on number shift and reduce (pop 1 state, p
6.	*stmt* ⟶ write *expr* .	on FOLLOW(*stmt*) = {id, read, write, $$
	stmt ⟶ *expr* . *add_op term*	(pop 2 states, push *stmt* on input)
		on *add_op* shift and goto 10
	add_op ⟶ . +	on + shift and reduce (pop 1 state, push *a*
	add_op ⟶ . -	on - shift and reduce (pop 1 state, push *a*

Figure 2.25: **CFSM for the calculator grammar (Figure 2.24).** Basi items in each state are separated by a horizontal rule. Trivial reduce-only sta eliminated by use of "shift and reduce" transitions *(continued)*.

	State	Transitions
7.	*expr* ⟶ *term* .	on FOLLOW(*expr*) = {id, read, write, $$,), +, -} reduce
	term ⟶ *term* . *mult_op factor*	(pop 1 state, push *expr* on input)
		on *mult_op* shift and goto 11
	mult_op ⟶ . *	on * shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ . /	on / shift and reduce (pop 1 state, push *mult_op* on input)
8.	*factor* ⟶ (. *expr*)	on *expr* shift and goto 12
	expr ⟶ . *term*	on *term* shift and goto 7
	expr ⟶ . *expr add_op term*	
	term ⟶ . *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ . *term mult_op factor*	
	factor ⟶ . (*expr*)	on (shift and goto 8
	factor ⟶ . id	on id shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ . number	on number shift and reduce (pop 1 state, push *factor* on input)
9.	*stmt* ⟶ id := *expr* .	on FOLLOW(*stmt*) = {id, read, write, $$} reduce
	expr ⟶ *expr* . *add_op term*	(pop 3 states, push *stmt* on input)
		on *add_op* shift and goto 10
	add_op ⟶ . +	on + shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ . -	on - shift and reduce (pop 1 state, push *add_op* on input)
10.	*expr* ⟶ *expr add_op* . *term*	on *term* shift and goto 13
	term ⟶ . *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ . *term mult_op factor*	
	factor ⟶ . (*expr*)	on (shift and goto 8
	factor ⟶ . id	on id shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ . number	on number shift and reduce (pop 1 state, push *factor* on input)
11.	*term* ⟶ *term mult_op* . *factor*	on *factor* shift and reduce (pop 3 states, push *term* on input)
	factor ⟶ . (*expr*)	on (shift and goto 8
	factor ⟶ . id	on id shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ . number	on number shift and reduce (pop 1 state, push *factor* on input)
12.	*factor* ⟶ (*expr* .)	on) shift and reduce (pop 3 states, push *factor* on input)
	expr ⟶ *expr* . *add_op term*	on *add_op* shift and goto 10
	add_op ⟶ . +	on + shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ . -	on - shift and reduce (pop 1 state, push *add_op* on input)
13.	*expr* ⟶ *expr add_op term* .	on FOLLOW(*expr*) = {id, read, write, $$,), +, -} reduce
	term ⟶ *term* . *mult_op factor*	(pop 3 states, push *expr* on input)
		on *mult_op* shift and goto 11
	mult_op ⟶ . *	on * shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ . /	on / shift and reduce (pop 1 state, push *mult_op* on input)

Figure 2.25: *(continued)*

LR Parsing (9/11)

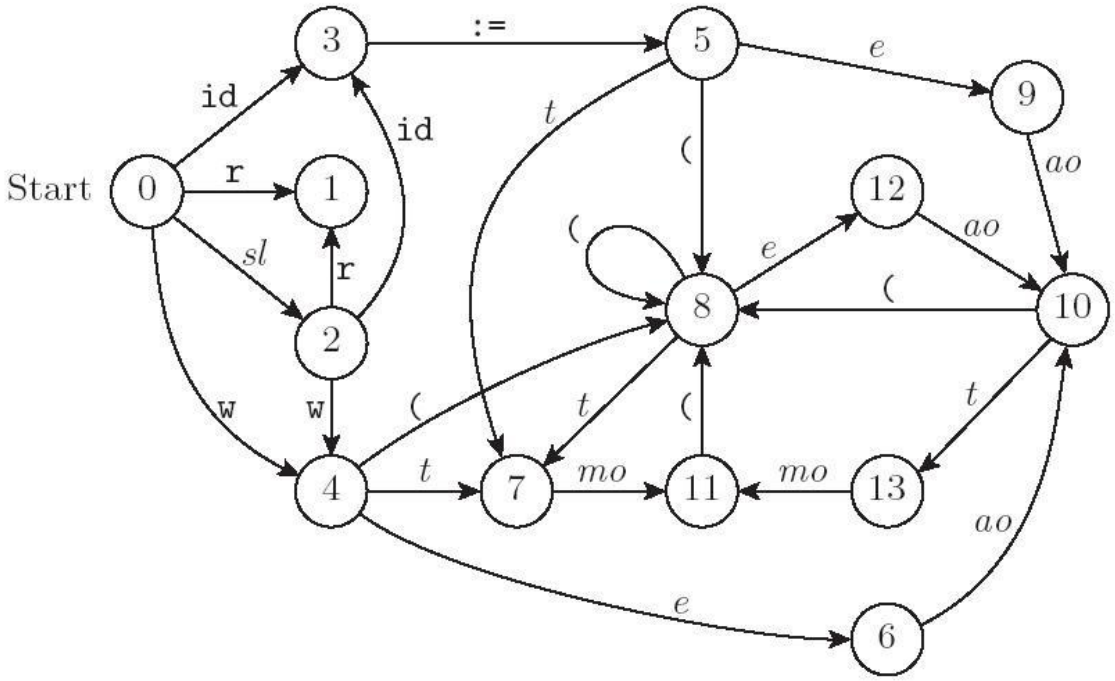


Figure 2.26: **Pictorial representation of the CFSM of Figure 2.25.** Symbol names have been abbreviated for clarity. Reduce actions are not shown.

LR Parsing (10/11)

Top-of-stack state	Current input symbol																	
	sl	*s*	*e*	*t*	*f*	*ao*	*mo*	id	lit	r	w	:=	(	)	+	-	*	/	$$
0	s2	b3	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	b5	–	–	–	–	–	–	–	–	–	–	–
2	–	b2	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	–	–	–	–	–	–	–	–	s5	–	–	–	–	–	–	–
4	–	–	s6	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
5	–	–	s9	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
6	–	–	–	–	–	s10	–	r6	–	r6	r6	–	–	–	b14	b15	–	–	r6
7	–	–	–	–	–	–	s11	r7	–	r7	r7	–	–	r7	r7	r7	b16	b17	r7
8	–	–	s12	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
9	–	–	–	–	–	s10	–	r4	–	r4	r4	–	–	–	b14	b15	–	–	r4
10	–	–	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	–	–	–	–	b11	b14	b15	–	–	–
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	r8

Figure 2.27: **SLR(1) parse table for the calculator language.** Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.24. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand side symbol and right-hand side length for each production.

LR Parsing (11/11)

- SLR parsing is based on
  - » Shift
  - » Reduce

  and also
  - » Shift & Reduce (for optimization)

Parse stack	Input stream	Comment
0	read A read B ...	
0 read *1*	A read B ...	shift read
0	*stmt* read B ...	shift id(A) & reduce by *stmt* ⟶ read id
0	*stmt_list* read B ...	shift *stmt* & reduce by *stmt_list* ⟶ *stmt*
0 stmt_list 2	read B sum ...	shift *stmt_list*
0 stmt_list 2 read *1*	B sum := ...	shift read
0 stmt_list 2	*stmt* sum := ...	shift id(B) & reduce by *stmt* ⟶ read id
0	*stmt_list* sum := ...	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 stmt_list 2	sum := A ...	shift *stmt_list*
0 stmt_list 2 id *3*	:= A + ...	shift id(sum)
0 stmt_list 2 id *3* := *5*	A + B ...	shift :=
0 stmt_list 2 id *3* := *5*	*factor* + B ...	shift id(A) & reduce by *factor* ⟶ id
0 stmt_list 2 id *3* := *5*	*term* + B ...	shift *factor* & reduce by *term* ⟶ *factor*
0 stmt_list 2 id *3* := *5 term 7*	+ B write ...	shift *term*
0 stmt_list 2 id *3* := *5*	*expr* + B write ...	reduce by *expr* ⟶ *term*
0 stmt_list 2 id *3* := *5 expr 9*	+ B write ...	shift *expr*
0 stmt_list 2 id *3* := *5 expr 9*	*add_op* B write ...	shift + & reduce by *add_op* ⟶ +
0 stmt_list 2 id *3* := *5 expr 9 add_op 10*	B write sum ...	shift *add_op*
0 stmt_list 2 id *3* := *5 expr 9 add_op 10*	*factor* write sum ...	shift id(B) & reduce by *factor* ⟶ id
0 stmt_list 2 id *3* := *5 expr 9 add_op 10*	*term* write sum ...	shift *factor* & reduce by *term* ⟶ *factor*
0 stmt_list 2 id *3* := *5 expr 9* *add_op 10 term 13*	write sum ...	shift *term*
0 stmt_list 2 id *3* := *5*	*expr* write sum ...	reduce by *expr* ⟶ *expr add_op term*
0 stmt_list 2 id *3* := *5 expr 9*	write sum ...	shift *expr*
0 stmt_list 2	*stmt* write sum ...	reduce by *stmt* ⟶ id := *expr*
0	*stmt_list* write sum ...	shift *stmt* & reduce by *stmt_list* ⟶ *stmt*
0 stmt_list 2	write sum ...	shift *stmt_list*
0 stmt_list 2 write *4*	sum write sum ...	shift write
0 stmt_list 2 write *4*	*factor* write sum ...	shift id(sum) & reduce by *factor* ⟶ id
0 stmt_list 2 write *4*	*term* write sum ...	shift *factor* & reduce by *term* ⟶ *factor*
0 stmt_list 2 write *4 term 7*	write sum ...	shift *term*
0 stmt_list 2 write *4*	*expr* write sum ...	reduce by *expr* ⟶ *term*
0 stmt_list 2 write *4 expr 6*	write sum ...	shift *expr*
0 stmt_list 2	*stmt* write sum ...	reduce by *stmt* ⟶ write *expr*
0	*stmt_list* write sum ...	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 stmt_list 2	write sum / ...	shift *stmt_list*
0 stmt_list 2 write *4*	sum / 2 ...	shift write
0 stmt_list 2 write *4*	*factor* / 2 ...	shift id(sum) & reduce by *factor* ⟶ id
0 stmt_list 2 write *4*	*term* / 2 ...	shift *factor* & reduce by *term* ⟶ *factor*
0 stmt_list 2 write *4 term 7*	/ 2 $$	shift *term*
0 stmt_list 2 write *4 term 7*	*mult_op* 2 $$	shift / & reduce by *mult_op* ⟶ /
0 stmt_list 2 write *4 term 7 mult_op 11*	2 $$	shift *mult_op*
0 stmt_list 2 write *4 term 7 mult_op 11*	*factor* $$	shift number(2) & reduce by *factor* ⟶ number
0 stmt_list 2 write *4*	*term* $$	shift *factor* & reduce by *term* ⟶ *term mult_op factor*
0 stmt_list 2 write *4 term 7*	$$	shift *term*
0 stmt_list 2 write *4*	*expr* $$	reduce by *expr* ⟶ *term*
0 stmt_list 2 write *4 expr 6*	$$	shift *expr*
0 stmt_list 2	*stmt* $$	reduce by *stmt* ⟶ write *expr*
0	*stmt_list* $$	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 stmt_list 2	$$	shift *stmt_list*
0	*program*	shift $$ & reduce by *program* ⟶ *stmt_list* $$
[done]		

Figure 2.29: **Trace of a table-driven SLR(1) parse of the sum-and-average program.** States in the parse stack are shown in boldface type. Symbols in the parse stack are for clarity only; they are not needed by the parsing algorithm. Parsing begins with the initial state of the CFSM (State 0) in the stack. It ends when we reduce by *program* ⟶ *stmt_list* $$, uncovering State 0 again and pushing *program* onto the input stream.

Agenda

1. Instructor and Course Introduction
2. Introduction to Programming Languages
3. Programming Language Syntax
4. Conclusion

Assignments & Readings

- Readings
  - Foreword/Preface, Chapters 1 and 2 (in particular, section 2.2.1)
- Assignment #1
  - See Assignment #1 posted under “handouts” on the course Web site
  - Due on June 12, 2014 by the beginning of class

Recommended Reference Books

- The books written by the creators of C++ and Java are the standard references:
  - Stroustrup. *The C++ programming Language, 3rd ed*. (Addison-Wesley)
  - Ken Arnold, James Gosling, and David Holmes. *The Java(TM) Programming Language, 4th ed*. (Addison-Wesley)
- For the remaining languages, there is a lot of information available on the web in the form of references and tutorials, so books may not be strictly necessary, but a few recommended textbooks are as follows:
  - John Barnes. *Programming in Ada95, 2nd ed*. (Addison Wesley)
  - Lawrence C. Paulson. *ML for the Working Programmer, 2nd ed*. Cambridge University Press
  - David Gelernter and Suresh Jagannathan: “Programming Linguistics”, MIT Press, 1990
  - Benjamin C. Pierce: “Types and Programming Languages”, MIT Press, 2002
  - Larry Wall, Tom Christiansen, and Jon Orwant: *Programming Perl, 3rd ed*. (O'Reilly)
  - Giannesini et al: “Prolog”, Addison-Wesley 1986.
  - Dewhurst & Stark, “Programming in C++”, Prentice Hall, 1989.
  - Ada 95 Reference Manual, http://www.adahome.com/rm95/
  - MIT Scheme Reference
    - http://www-swiss.ai.mit.edu/projects/scheme/documentation/scheme.html
  - Strom et al: “Hermes: A Language for Distributed Computing”, Prentice-Hall, 1991.
  - R. Kent Dybvig, “The SCHEME Programming Language”, Prentice Hall, 1987
  - Jan Skansholm, “ADA 95 From the Beginning”, Addison Wesley, 1997.

Next Session: Imperative Languages – Names, Scoping, and Bindings