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

Adapted from course textbook resources
Programming Language Pragmatics (3rd Edition)
Michael L. Scott, Copyright © 2009 Elsevier
Introduction to Programming Languages
- 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

Who am I?
<table>
<thead>
<tr>
<th>Method</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell</td>
<td>(212) 203-5004</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:jcf@cs.nyu.edu">jcf@cs.nyu.edu</a></td>
</tr>
<tr>
<td>AIM, Y! IM, ICQ</td>
<td>jcf2_2003</td>
</tr>
<tr>
<td>MSN IM</td>
<td><a href="mailto:jcf2_2003@yahoo.com">jcf2_2003@yahoo.com</a></td>
</tr>
<tr>
<td>LinkedIn</td>
<td><a href="http://www.linkedin.com/in/jcfranchitti">http://www.linkedin.com/in/jcfranchitti</a></td>
</tr>
<tr>
<td>Twitter</td>
<td><a href="http://twitter.com/jcfranchitti">http://twitter.com/jcfranchitti</a></td>
</tr>
<tr>
<td>Skype</td>
<td><a href="mailto:jcf2_2003@yahoo.com">jcf2_2003@yahoo.com</a></td>
</tr>
</tbody>
</table>
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
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
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
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)
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.
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?”

- Hoare, “An Axiomatic Basis For Computer Programming”,

- Hoare, “The Emperor’s Old Clothes”,

- Parnas, “On the Criteria to be Used in Decomposing Systems into Modules”,
  - http://www.acm.org/classics/may96/
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!
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
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).
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
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 $x^2$ instead of $x^{**2}$)
– use C pointers or Pascal "with" statement to factor address calculations
– 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++)
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)
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)
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, 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
- 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.
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.
Modern imperative languages (e.g., Ada, C++, Java) have similar characteristics:

- 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)
**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
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
  » 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:
Pure Interpretation

- Interpreter stays around for the execution of the program
- Interpreter is the locus of control during execution
- **Interpretation:**
  - Greater flexibility
  - Better diagnostics (error messages)

- **Compilation**
  - Better performance
Common case is compilation or simple pre-processing, followed by interpretation.

Most language implementations include a mixture of both compilation and interpretation.
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
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+
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)
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:

```
Fortran program → Compiler → Incomplete machine language

Incomplete machine language

Library routines
```

```
Linker → Machine language program
```
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)
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
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:
Implementation strategies:

» *Bootstrapping*
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.
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.
- 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.
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
## Tools

<table>
<thead>
<tr>
<th>Type</th>
<th>Unix examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editors</td>
<td>vi, emacs</td>
</tr>
<tr>
<td>Pretty printers</td>
<td>cb, indent</td>
</tr>
<tr>
<td>Pre-processors (esp. macros)</td>
<td>cpp, m4, watfor</td>
</tr>
<tr>
<td>Debuggers</td>
<td>adb, sdb, dbx, gdb</td>
</tr>
<tr>
<td>Style checkers</td>
<td>lint, purify</td>
</tr>
<tr>
<td>Module management</td>
<td>make</td>
</tr>
<tr>
<td>Version management</td>
<td>sccs, rcs</td>
</tr>
<tr>
<td>Assemblers</td>
<td>as</td>
</tr>
<tr>
<td>Link editors, loaders</td>
<td>ld, ld-so</td>
</tr>
<tr>
<td>Perusal tools</td>
<td>more, less, od, nm</td>
</tr>
<tr>
<td>Program cross-reference</td>
<td>ctags</td>
</tr>
</tbody>
</table>
Phases of Compilation

- Character stream
- Token stream
- Parse tree
- Abstract syntax tree or other intermediate form
- Modified intermediate form
- Target language (e.g., assembler)
- Modified target language
- Scanner (lexical analysis)
- Parser (syntax analysis)
- Semantic analysis and intermediate code generation
- Machine-independent code improvement (optional)
- Target code generation
- Machine-specific code improvement (optional)
**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)
 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)
- **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
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
- **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
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.
Lexical and Syntax Analysis

GCD Program (in C)

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

```c
int main () {
    int i = getint (), j = getint ();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint (i);
}
```
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
Context-Free Grammar and Parsing

Example (while loop in C)

\[
\text{iteration-statement} \rightarrow \text{while (expression) statement}
\]

statement, in turn, is often a list enclosed in braces:
\[
\text{statement} \rightarrow \text{compound-statement}
\]
\[
\text{compound-statement} \rightarrow \{ \text{block-item-list opt} \}
\]
where
\[
\text{block-item-list opt} \rightarrow \text{block-item-list}
\]
or
\[
\text{block-item-list opt} \rightarrow \epsilon
\]
and
\[
\text{block-item-list} \rightarrow \text{block-item}
\]
\[
\text{block-item-list} \rightarrow \text{block-item-list block-item}
\]
\[
\text{block-item} \rightarrow \text{declaration}
\]
\[
\text{block-item} \rightarrow \text{statement}
\]
- Context-Free Grammar and Parsing
  - GCD Program Parse Tree
Context-Free Grammar and Parsing (cont.)
- Context-Free Grammar and Parsing (cont.)

### An Overview of Compilation (14/15)
- 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)
```
Many non-terminals inside a parse tree are artifacts of the grammar

Remember:

\[
E ::= E + T | T \\
T ::= T \ast \text{Id} | \text{Id}
\]

The parse tree for \(B \ast C\) can be written as

\[
E(T(\text{Id}(B), \text{Id}(C)))
\]

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

In the example:

\[
T(\text{Id}(B), \text{Id}(C))
\]

Consequently, many parsers really generate abstract syntax trees.
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?
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
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
A grammar $G$ is a tuple $(\Sigma, N, S, \delta)$

- $N$ is the set of non-terminal symbols
- $S$ is the distinguished non-terminal: the root symbol
- $\Sigma$ is the set of terminal symbols (alphabet)
- $\delta$ is the set of rewrite rules (productions) of the form: $ABC. \ldots ::= XYZ \ldots$

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)
Consider the following grammar G:

- $N = \{S; X; Y\}$
- $S = S$
- $\Sigma = \{a; b; c\}$
- $\delta$ consists of the following rules:
  - $S \rightarrow b$
  - $S \rightarrow XbY$
  - $X \rightarrow a$
  - $X \rightarrow aX$
  - $Y \rightarrow c$
  - $Y \rightarrow Yc$

Some sample derivations:

- $S \rightarrow b$
- $S \rightarrow XbY \rightarrow abY \rightarrow abc$
- $S \rightarrow XbY \rightarrow aXbY \rightarrow aaXbY \rightarrow aaabY \rightarrow aaabc$
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
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:

- $\epsilon$ denotes $\emptyset$
- a character $x$, where $x \in \Sigma$, denotes $\{x\}$
- (sequencing) a sequence of two regular expressions $RS$ denotes
- $\{\alpha\beta \mid \alpha \in [[R]], \beta \in [[S]]\}$
- (alternation) $R|S$ denotes $[[R]] \cup [[S]]$
- (Kleene star) $R^*$ denotes the set of strings which are concatenations of zero or more strings from $[[R]]$
- parentheses are used for grouping

Shorthands:

- $R^? \equiv \epsilon \lor R$
- $R^+ \equiv RR^*$
A regular expression is one of the following:

- A character
- The empty string, denoted by \( \Sigma \)
- 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)
Numerical literals in Pascal may be generated by the following:

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_integer → digit digit *

unsigned_number → unsigned_integer ( ( . unsigned_integer ) | ε )
( ( ( e | E ) ( + | - | ε ) unsigned_integer ) | ε )
• 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: 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

\[
\text{Id ::= Letter IdRest} \\
\text{IdRest ::= € | Letter IdRest | Digit IdRest}
\]

Missing from above grammar: limit of identifier length

Other issues: international characters, case-sensitivity, limit of identifier length
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?
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
Expression grammar with precedence and associativity

1. \( \text{expr} \rightarrow \text{term} \mid \text{expr} \ \text{add}\_\text{op} \ \text{term} \)
2. \( \text{term} \rightarrow \text{factor} \mid \text{term} \ \text{mult}\_\text{op} \ \text{factor} \)
3. \( \text{factor} \rightarrow \text{id} \mid \text{number} \mid - \ \text{factor} \mid ( \ \text{expr} \ ) \)
4. \( \text{add}\_\text{op} \rightarrow + \mid - \)
5. \( \text{mult}\_\text{op} \rightarrow * \mid / \)
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
Ambiguity:

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

\[ E ::= E + E \mid E \ast E \mid Id \]

» Two possible parse trees for “A + B \ast C”:
  • ((A + B) \ast C)
  • (A + (B \ast C))

» One solution: rearrange grammar:

\[ E ::= E + T \mid T \]
\[ T ::= T \ast Id \mid Id \]

» Harder problems – disambiguate these (courtesy of Ada):
  • function call ::= name (expression list)
  • indexed component ::= name (index list)
  • type conversion ::= name (expression)
Parse tree for expression grammar (with precedence) for \(3 + 4 \ast 5\)
Parse tree for expression grammar (with left associativity) for $10 - 4 - 3$
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
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
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
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
Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton.
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
| ...
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
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.
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)
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 see a dot
  • do you proceed (in hopes of getting 3.14)?
    or
  • do you stop (in fear of getting 3..5)?
In messier cases, you may not be able to get by with any fixed amount of look-ahead. In Fortran, for example, we have

\[
\text{DO 5 I = 1,25 loop}
\]
\[
\text{DO 5 I = 1.25 assignment}
\]

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.
- Terminology:
  - context-free grammar (CFG)
  - symbols
    - terminals (tokens)
    - non-terminals
  - production
  - derivations (left-most and right-most - canonical)
  - parse trees
  - sentential form
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
- 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.
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’
- 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
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 \textit{prefix property} (no valid string is a prefix of another valid string) has an LR(0) grammar.
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)
Here is an LL(1) grammar (Fig 2.15):

1. $\text{program} \rightarrow \text{stmt list} \$$$
2. $\text{stmt_list} \rightarrow \text{stmt stmt_list}$
3. \quad \mid \varepsilon$
4. $\text{stmt} \rightarrow \text{id := expr}$
5. \quad \mid \text{read id}$
6. \quad \mid \text{write expr}$
7. $\text{expr} \rightarrow \text{term term_tail}$
8. $\text{term_tail} \rightarrow \text{add op term term_tail}$
9. \quad \mid \varepsilon$
- 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. | /
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
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
- Parse tree for the average program (Figure 2.17)
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(1) parse table for parsing for calculator language

<table>
<thead>
<tr>
<th>Top-of-stack nonterminal</th>
<th>id</th>
<th>number</th>
<th>read</th>
<th>Current input token</th>
<th>write</th>
<th>:=</th>
<th>( )</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>stmt_list</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>stmt</td>
<td>4</td>
<td>-</td>
<td>5</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>expr</td>
<td>7</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term_tail</td>
<td>9</td>
<td>-</td>
<td>9</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>term</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>factor_tail</td>
<td>12</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>factor</td>
<td>14</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>add_op</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>mult_op</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19</td>
</tr>
</tbody>
</table>
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
- Problems trying to make a grammar LL(1)
  - common prefixes: another thing that LL parsers can't handle
    - solved by "left-factoring"
    - example:
      \[\text{stmt} \rightarrow \text{id} \ := \ \text{expr} \mid \text{id} \ ( \text{arg\_list} )\]
      equivalently
      \[\text{stmt} \rightarrow \text{id} \ \text{id\_stmt\_tail}\]
      \[\text{id\_stmt\_tail} \rightarrow \ := \ \text{expr} \mid ( \text{arg\_list})\]
  - we can eliminate left-factor mechanically
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
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)

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if } \text{cond}\ \text{then}_\text{clause}\ \text{else}_\text{clause} \\
& \quad | \ \text{other}_\text{stuff} \\
\text{then}_\text{clause} & \rightarrow \ \text{then } \text{stmt} \\
\text{else}_\text{clause} & \rightarrow \ \text{else } \text{stmt} \\
& \quad | \ \epsilon
\end{align*}
\]
Consider:  
\[ S ::= \text{if } E \text{ then } S \]
\[ S ::= \text{if } E \text{ then } S \text{ else } S \]

The sentence
\[ \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \]
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
The less natural grammar fragment can be parsed bottom-up but not top-down

\[
\begin{align*}
\text{stmt} & \rightarrow \text{balanced_stmt} \mid \text{unbalanced_stmt} \\
\text{balanced_stmt} & \rightarrow \text{if cond then balanced_stmt} \\
& \quad \text{else balanced_stmt} \\
& \quad \mid \text{other_stuff} \\
\text{unbalanced_stmt} & \rightarrow \text{if cond then stmt} \\
& \quad \mid \text{if cond then balanced_stmt} \\
& \quad \mid \text{else unbalanced_stmt}
\end{align*}
\]
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)
Better yet, languages (since Pascal) generally employ explicit end-markers, which eliminate this problem.

In Modula-2, for example, one says:

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

Ada says 'end if'; other languages say 'fi'.
One problem with end markers is that they tend to bunch up. In Pascal you say

```pascal
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

```pascal
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...
end; end; end; end; end;
```
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
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
• Algorithm First/Follow/Predict:

- $\text{FIRST}(\alpha) = \{ a : \alpha \rightarrow^* a \beta \}
  \cup (\text{if } \alpha \Rightarrow^* \varepsilon \text{ THEN } \{ \varepsilon \} \text{ ELSE NULL})$

- $\text{FOLLOW}(A) = \{ a : S \rightarrow^+ \alpha A a \beta \}
  \cup (\text{if } S \rightarrow^* \alpha A \text{ THEN } \{ \varepsilon \} \text{ ELSE NULL})$

- $\text{Predict } (A \rightarrow X_1 \ldots X_m) = (\text{FIRST } (X_1 \ldots X_m) - \{ \varepsilon \}) \cup (\text{if } X_1, \ldots, X_m \rightarrow^* \varepsilon \text{ then } \text{FOLLOW } (A) \text{ ELSE NULL})$

⚠️ Details following…
program → stmt_list $$

stmt_list → stmt stmt_list
stmt_list → ε
stmt → id := expr
stmt → read id
stmt → write expr
expr → term term_tail
term_tail → add_op term term_tail
term_tail → ε
term → factor factor_tail
factor_tail → mult_op factor factor_tail
factor_tail → ε
factor → ( expr )
factor → id
factor → number
add_op → +
add_op → -
mult_op → *
mult_op → /

$$ ∈ FOLLOW(stmt_list),
ε ∈ FOLLOW($$), and ε ∈ FOLLOW(program)
ε ∈ FIRST(stmt_list)
id ∈ FIRST(stmt) and := ∈ FOLLOW(id)
read ∈ FIRST(stmt) and id ∈ FOLLOW(read)
write ∈ FIRST(stmt)
ε ∈ FIRST(term_tail)
ε ∈ FIRST(factor_tail)
( ∈ FIRST(factor) and ) ∈ FOLLOW(expr)
id ∈ FIRST(factor)
number ∈ FIRST(factor)
+ ∈ FIRST(add_op)
- ∈ FIRST(add_op)
* ∈ FIRST(mult_op)
/ ∈ FIRST(mult_op)

Figure 2.21: “Obvious” facts about the LL(1) calculator grammar.
LL Parsing (22/23)

FIRST

\[
\begin{align*}
\text{program} & \rightarrow \text{id, read, write, }\$$
\text{stmt_list} & \rightarrow \text{id, read, write, }c
\text{stmt} & \rightarrow \text{id, read, write}
\text{expr} & \rightarrow \{, \text{id, number}\}
\text{term_tail} & \rightarrow +, -, c
\text{term} & \rightarrow \{, \text{id, number}\}
\text{factor_tail} & \rightarrow *, /, c
\text{factor} & \rightarrow \{, \text{id, number}\}
\text{add_op} & \rightarrow +, -
\text{mul_op} & \rightarrow *, /
\end{align*}
\]

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

FOLLOW

\[
\begin{align*}
\text{id} & \rightarrow +, -, *, /, \text{ :=, id, read, write, }\$$
\text{number} & \rightarrow +, -, *, /, \text{id, read, write, }\$$
\text{read} & \rightarrow \text{id}
\text{write} & \rightarrow \{, \text{id, number}\}
\text{:=} & \rightarrow \{, \text{id, number}\}
\text{+} & \rightarrow \{, \text{id, number}\}
\text{-} & \rightarrow \{, \text{id, number}\}
\text{*} & \rightarrow \{, \text{id, number}\}
\text{/} & \rightarrow \{, \text{id, number}\}
\text{\$$} & \rightarrow \{c\}
\text{program} & \rightarrow \{\epsilon\}
\text{stmt_list} & \rightarrow \$$
\text{stmt} & \rightarrow \text{id, read, write, }\$$
\end{align*}
\]

PREDICT

\[
\begin{align*}
1 & \quad \text{program} \rightarrow \text{stmt_list }\$$ \{\text{id, read, write, }\$$
2 & \quad \text{stmt_list} \rightarrow \text{stmt stmt_list} \{\text{id, read, write}\}
3 & \quad \text{stmt_list} \rightarrow c \{\$$
4 & \quad \text{stmt} \rightarrow \text{id} := \text{expr} \{\text{id}\}
5 & \quad \text{stmt} \rightarrow \text{read id} \{\text{read}\}
6 & \quad \text{stmt} \rightarrow \text{write expr} \{\text{write}\}
7 & \quad \text{expr} \rightarrow \text{term term_tail} \{, \text{id, number}\}
8 & \quad \text{term_tail} \rightarrow \text{add_op term term_tail} \{+, -\}
9 & \quad \text{term_tail} \rightarrow c \{, \text{id, number}\}
10 & \quad \text{term} \rightarrow \text{factor factor_tail} \{, \text{id, number}\}
11 & \quad \text{factor_tail} \rightarrow \text{mul_op factor factor_tail} \{*, /\}
12 & \quad \text{factor_tail} \rightarrow c \{+, -, \text{id, number}\}
13 & \quad \text{factor} \rightarrow \{\text{expr} \} \{\epsilon\}
14 & \quad \text{factor} \rightarrow \text{id} \{\text{id}\}
15 & \quad \text{factor} \rightarrow \text{number} \{\text{number}\}
16 & \quad \text{add_op} \rightarrow + \{+\}
17 & \quad \text{add_op} \rightarrow - \{-\}
18 & \quad \text{mul_op} \rightarrow * \{\ast\}
19 & \quad \text{mul_op} \rightarrow / \{/\}
\end{align*}
\]

Figure 2.22: FIRST, FOLLOW, and PREDICT sets for the calculator language.
- 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 $\Sigma$
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).
- 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 \(<\text{input symbol}, \text{top-of-stack symbol}>\) 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
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
• 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
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. | /
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
Figure 2.25: CFSSM for the calculator grammar (Figure 2.24). Bars in each state are separated by a horizontal rule. Trivial reduce-only states are eliminated by use of ‘shift and reduce’ transitions (continued).
Figure 2.26: Pictorial representation of the CFSM of Figure 2.25. Symbol names have been abbreviated for clarity. Reduce actions are not shown.
<table>
<thead>
<tr>
<th>Top-of-stack state</th>
<th>sl</th>
<th>s</th>
<th>e</th>
<th>t</th>
<th>f</th>
<th>ao</th>
<th>mo</th>
<th>Current input symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
<td>b3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s3 s1 s4</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>b5</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>b2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s3 s1 s4</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s5</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>s6</td>
<td>s7</td>
<td>b9</td>
<td>-</td>
<td>-</td>
<td>b12 b13 s8</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>s9</td>
<td>s7</td>
<td>b9</td>
<td>-</td>
<td>-</td>
<td>b12 b13 s8</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s10 r6 r6 r6</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s11 r7 r7 r7</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>s12 s7 b9</td>
<td>-</td>
<td>-</td>
<td>b12 b13 s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s10 r4 r4 r4</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>s13 b9</td>
<td>-</td>
<td>-</td>
<td>b12 b13 s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>b10 b12 b13 s8</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s10 b11 b14 b15</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>s11 r8 r8 r8</td>
</tr>
</tbody>
</table>

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.
• SLR parsing is based on
  » Shift
  » Reduce
  and also
  » Shift & Reduce (for optimization)

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Input stream</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read A read B ...</td>
<td>shift read</td>
</tr>
<tr>
<td>0</td>
<td>A read B ...</td>
<td>shift id(A) &amp; reduce by stmt → id</td>
</tr>
<tr>
<td>0</td>
<td>stmt_list read B ...</td>
<td>shift stmt &amp; reduce by stmt_list → stmt</td>
</tr>
<tr>
<td>0</td>
<td>read B sum ...</td>
<td>shift read</td>
</tr>
<tr>
<td>0</td>
<td>B sum → id ...</td>
<td>shift id(B) &amp; reduce by stmt → id</td>
</tr>
<tr>
<td>0</td>
<td>stmt_list sum → stmt</td>
<td>shift stmt &amp; reduce by stmt_list → stmt</td>
</tr>
<tr>
<td>0</td>
<td>sum → A ...</td>
<td>shift id(sum)</td>
</tr>
<tr>
<td>0</td>
<td>A = B ...</td>
<td>shift =</td>
</tr>
<tr>
<td>0</td>
<td>factor + B ...</td>
<td>shift id(A) &amp; reduce by factor → id</td>
</tr>
<tr>
<td>0</td>
<td>term + B ...</td>
<td>shift factor &amp; reduce by term → factor</td>
</tr>
<tr>
<td>0</td>
<td>+ B write ...</td>
<td>shift expr</td>
</tr>
<tr>
<td>0</td>
<td>expr + B write ...</td>
<td>shift expr</td>
</tr>
<tr>
<td>0</td>
<td>add_op B write ...</td>
<td>add &amp; reduce by add_op → +</td>
</tr>
<tr>
<td>0</td>
<td>B write sum ...</td>
<td>shift addOp</td>
</tr>
<tr>
<td>0</td>
<td>expr write sum ...</td>
<td>reduce by expr → expr addOp term</td>
</tr>
<tr>
<td>0</td>
<td>write sum ...</td>
<td>shift expr</td>
</tr>
<tr>
<td>0</td>
<td>expr write sum ...</td>
<td>reduce by expr → expr addOp term</td>
</tr>
<tr>
<td>0</td>
<td>write sum ...</td>
<td>shift expr</td>
</tr>
<tr>
<td>0</td>
<td>stmt_list write sum ...</td>
<td>shift stmt &amp; reduce by stmt_list → stmt</td>
</tr>
<tr>
<td>0</td>
<td>write sum ...</td>
<td>shift stmt_list</td>
</tr>
<tr>
<td>0</td>
<td>stmt_list write sum ...</td>
<td>shift stmt &amp; reduce by stmt_list → stmt</td>
</tr>
<tr>
<td>0</td>
<td>write sum ...</td>
<td>shift stmt_list</td>
</tr>
<tr>
<td>0</td>
<td>factor write sum ...</td>
<td>shift factor &amp; reduce by factor → id</td>
</tr>
<tr>
<td>0</td>
<td>term write sum ...</td>
<td>shift factor &amp; reduce by term → factor</td>
</tr>
<tr>
<td>0</td>
<td>write sum ...</td>
<td>shift term</td>
</tr>
<tr>
<td>0</td>
<td>expr write sum ...</td>
<td>reduce by expr → expr term</td>
</tr>
<tr>
<td>0</td>
<td>write sum ...</td>
<td>shift expr</td>
</tr>
<tr>
<td>0</td>
<td>stmt_list write sum ...</td>
<td>shift stmt &amp; reduce by stmt_list → stmt</td>
</tr>
<tr>
<td>0</td>
<td>write sum / ...</td>
<td>shift stmt_list</td>
</tr>
<tr>
<td>0</td>
<td>stmt_list write sum ...</td>
<td>shift stmt &amp; reduce by stmt_list → stmt</td>
</tr>
<tr>
<td>0</td>
<td>write sum / ...</td>
<td>shift stmt_list</td>
</tr>
<tr>
<td>0</td>
<td>factor / 2 ...</td>
<td>shift id(2) &amp; reduce by factor → id</td>
</tr>
<tr>
<td>0</td>
<td>term / 2 ...</td>
<td>shift factor &amp; reduce by term → factor</td>
</tr>
<tr>
<td>0</td>
<td>/ 2 $S$</td>
<td>shift term</td>
</tr>
<tr>
<td>0</td>
<td>shift mailOp ...</td>
<td>shift &amp; reduce by mailOp → /</td>
</tr>
<tr>
<td>0</td>
<td>mailOp 2 $S$</td>
<td>shift number(2) &amp; reduce by factor → number</td>
</tr>
<tr>
<td>0</td>
<td>number $S$</td>
<td>shift factor &amp; reduce by term → term mailOp factor</td>
</tr>
<tr>
<td>0</td>
<td>$S$</td>
<td>shift term</td>
</tr>
<tr>
<td>0</td>
<td>reduce by expr → term</td>
<td>shift expr</td>
</tr>
<tr>
<td>0</td>
<td>$S$</td>
<td>shift expr</td>
</tr>
<tr>
<td>0</td>
<td>stmt $S$</td>
<td>reduce by stmt → stmt write expr</td>
</tr>
<tr>
<td>0</td>
<td>stmt_list $S$</td>
<td>shift stmt &amp; reduce by stmt_list → stmt_list $S$</td>
</tr>
<tr>
<td>0</td>
<td>$S$</td>
<td>shift stmt_list</td>
</tr>
<tr>
<td>0</td>
<td>program</td>
<td>shift $S$ &amp; reduce by program → stmt_list $S$</td>
</tr>
<tr>
<td>[done]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $S$; uncovering State 0 again and pushing program onto the input stream.
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:

- 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)
  - Benjamin C. Pierce: “Types and Programming Languages”, MIT Press, 2002
  - Larry Wall, Tom Christiansen, and Jon Orwant: *Programming Perl, 3rd ed.* (O'Reilly)
  - MIT Scheme Reference