Introduction

Chapter 1

Programming languages - ubiquitous













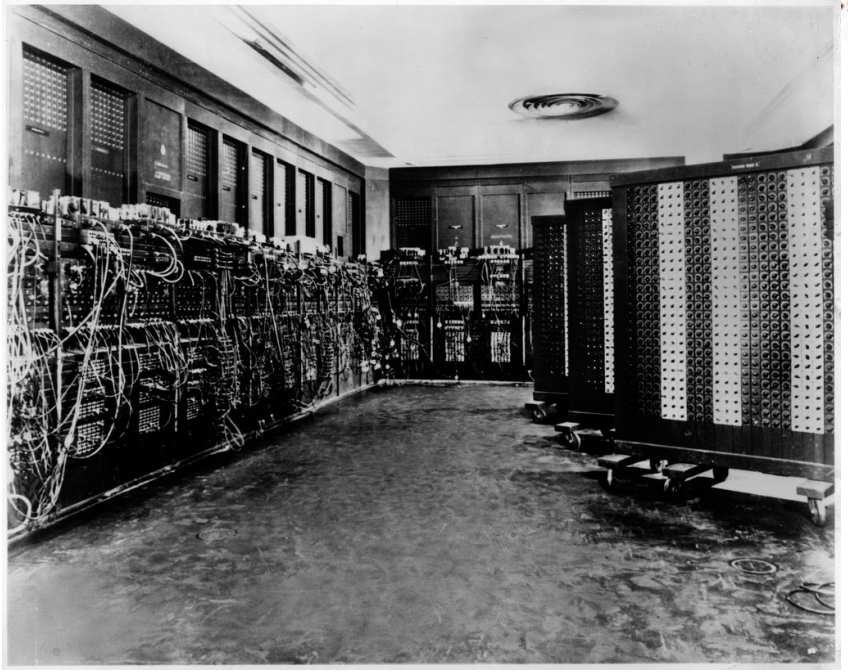






Computer evolution

- ENIAC
  - 18,000 sq feet
  - 25 tones = 25,000 Kg
  - 5,000 instr/s



- iPhone 6
  - 4.55 ounces = 0.13 Kg
  - 25,000,000,000 instr/s

  - 200,000 x smaller, 5,000,000 x faster
    = 1,000,000,000,000 x more efficient

Computer evolution - Quotes

- *"I think there is a world market for maybe five computers."*

  (Thomas Watson, president of IBM, 1943)

- *"Where a calculator like the ENIAC today is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh only 1½ tons."*

  (Andrew Hamilton, "Brains that Click", 1949)

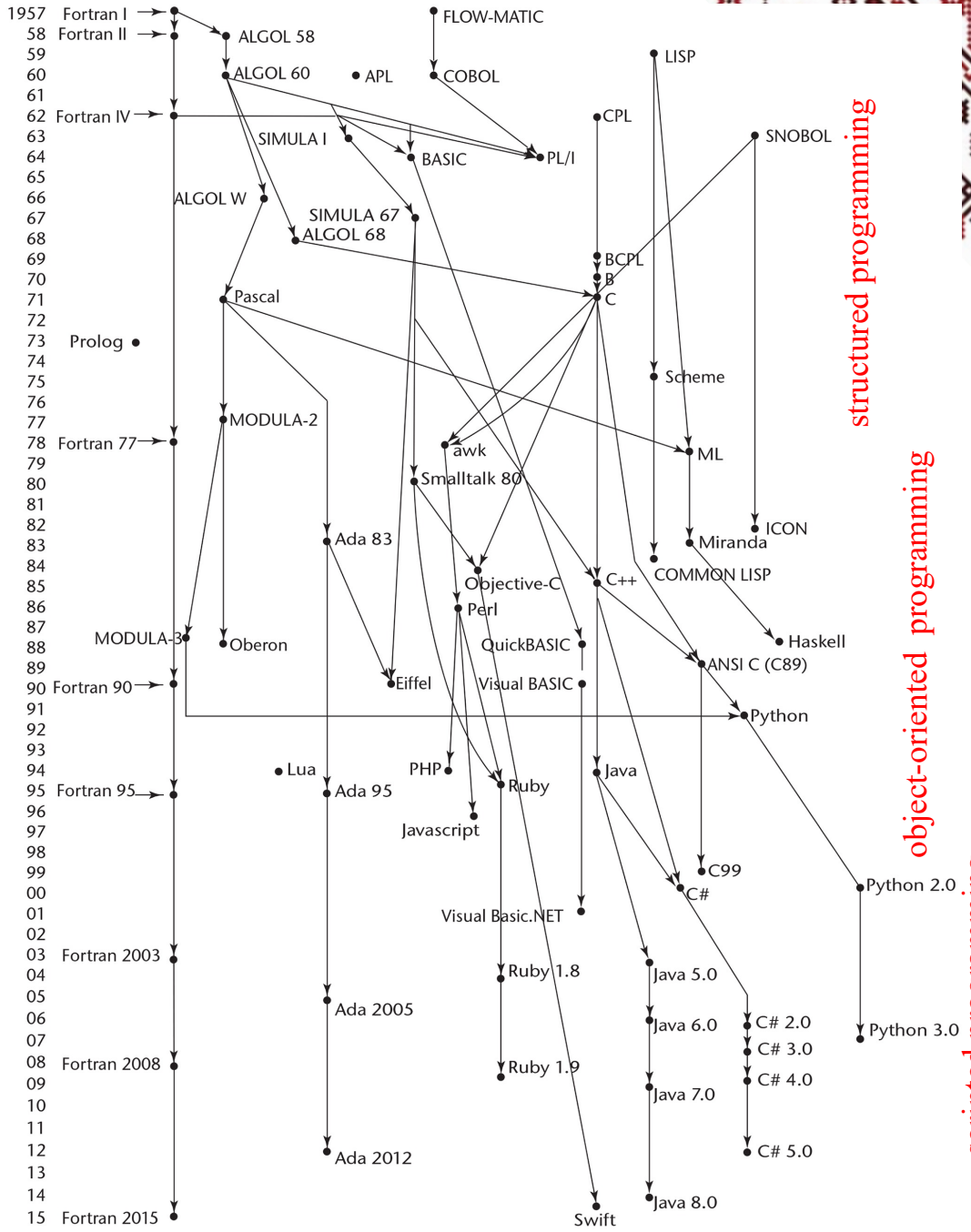
- *"The cost for 128 kilobytes of memory will fall below U$100 in the near future."*

  (Creative Computing magazine, December 1981)

Introduction

Why are there so many languages?

- Evolution
- Special purposes
- Personal preference
- Features
- Availability
  - Standardization
  - Open source
- Good compilers
- Socio-economic factors



Introduction

What are programming languages for?

- way of thinking - expressing algorithms
- abstraction of virtual machine - way of specifying what you want the hardware to do without getting down into the bits
- implementor's point of view *vs.* programmer's point of view

> *"Programming is the art of telling another human being what one wants the computer to do."*
>
> Donald Knuth

- conceptual clarity
- implementation efficiency

Introduction

What makes a language successful?

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

Programming languages spectrum

- imperative – how the computer should do it?
  - von Neumann - C, Fortran, Pascal, Basic
  - object-oriented - C++, Smalltalk, Java
  - scripting languages - Python, Perl, JavaScript, PHP
- declarative – what the computer is to do?
  - functional - Scheme, ML, Lisp, FP
  - logic - Prolog, VisiCalc, RPG

- imperative languages predominate
  - better performance
- declarative languages are higher level
  - farther from implementation details
  - safer; imperative languages started importing their features

Evolution

- Machine language

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

- Assembly

```
    pushl   %ebp            jle   D
    movl    %esp, %ebp      subl  %eax, %ebx
    pushl   %ebx         B: cmpl  %eax, %ebx
    subl    $4, %esp        jne   A
    andl    $-16, %esp   C: movl  %ebx, (%esp)
    call    getint          call  putint
    movl    %eax, %ebx      movl  -4(%ebp), %ebx
    call    getint          leave
    cmpl    %eax, %ebx      ret
    je      C            D: subl  %ebx, %eax
 A: cmpl    %eax, %ebx      jmp   B
```

- Fortran

```
FUNCTION GCD(A, B)
        IA = A
        IB = B
   1    IF (IB.NE.0) THEN
          ITEMP = IA
          IA = IB
          IB = MOD(ITEMP, IB)
          GOTO 1
        END IF
        GCD = IA
        RETURN
      END
```

Evolution

- C++

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```

```
int gcd2(int a, int b) {
    return (b==0) ? a : gcd2(b, a%b);
}
```

- Python

```
def gcd(x, y):
    while (y):
        x, y = y, x % y
    return x


def gcd2(a,b):
    return a if (b==0) else gcd2(b, a%b)
```

Evolution

- Scheme

```
(define gcd
  (lambda (a b)
    (cond ((zero? b) a)
          (else (gcd b (modulo a b))))))
```

- Prolog

```
gcd(X,Y,G) :- X=Y, G=X.
gcd(X,Y,G) :- X<Y, Y1 is Y-X, gcd(X,Y1,G).
gcd(X,Y,G) :- X>Y, gcd(Y,X,G).
```

Why study programming languages?

- Help you choose a language:
  - systems programming: C, C++, C#
  - numerical computations: Fortran, C, Matlab
  - web-based applications: PHP, Javascript, Ruby
  - embedded systems: Ada, C
  - symbolic data manipulation: Scheme, ML, Common Lisp
  - networked PC programs: Java, .NET
  - logical relationships: Prolog
- Make it easier to learn new languages:
  - many concepts are common to many languages: syntax, semantics, iteration, recursion, abstraction, etc.
- Make better use of the language you are using:
  - understand various features
  - understand implementation cost
  - find ways to do things that are not explicitly supported

Top Languages

TOP 10

Popular Programming Languages in 2020

1	Python
2	JavaScript
3	Java
4	C#
5	C
6	C++
7	GO
8	R
9	Swift
10	PHP

Our List of the Top 20 Programming Languages

1. JavaScript (React.js and Node.js)
2. Python
3. HTML
4. CSS
5. C++
6. TypeScript
7. Rust
8. Scheme
9. Java
10. Kotlin
11. C#
12. Perl
13. PHP
14. Scala
15. Swift
16. MATLAB
17. SQL
18. R Programming Language
19. Golang (Go)
20. Ruby

Top 10 Most Popular Programming Languages In 2020

Oct 2020	Programming Language	Ratings
1	C	16.95%
2	Java	12.56%
3	Python	11.28%
4	C++	6.94%
5	C#	4.16%
6	Visual Basic	3.97%
7	JavaScript	2.14%
8	PHP	2.09%
9	R	1.99%
10	SQL	1.57%

The Power of Abstraction

- Abstraction - ability to control complexity
  - high-level programming
  - names
  - functions / procedures / methods
  - objects
  - functional programming

  - *"Mathematics is the queen of the sciences."*

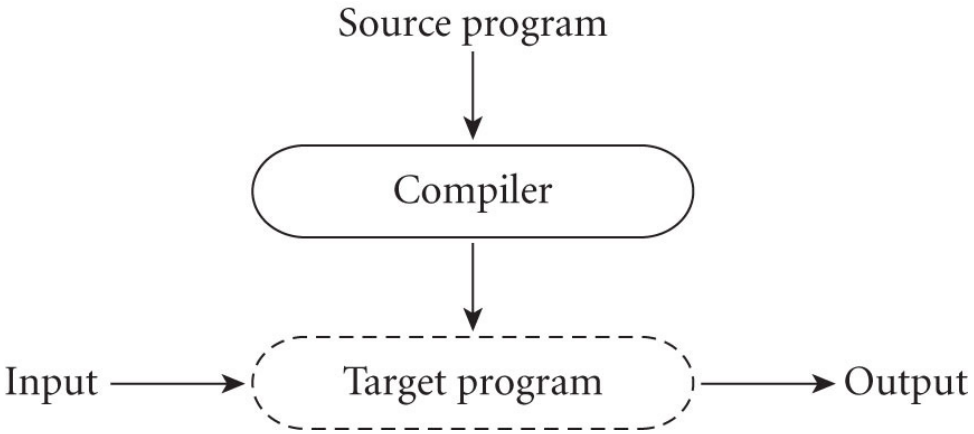
    Carl Friedrich Gauss

  - *"Mathematics is the language with which God has written the universe."*

    Galileo Galilei

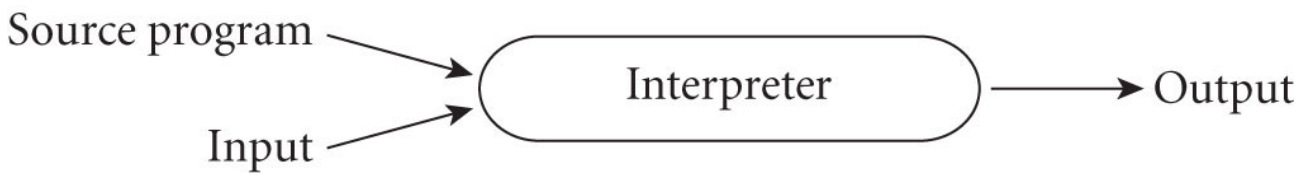
Compilation vs. Interpretation

- 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

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

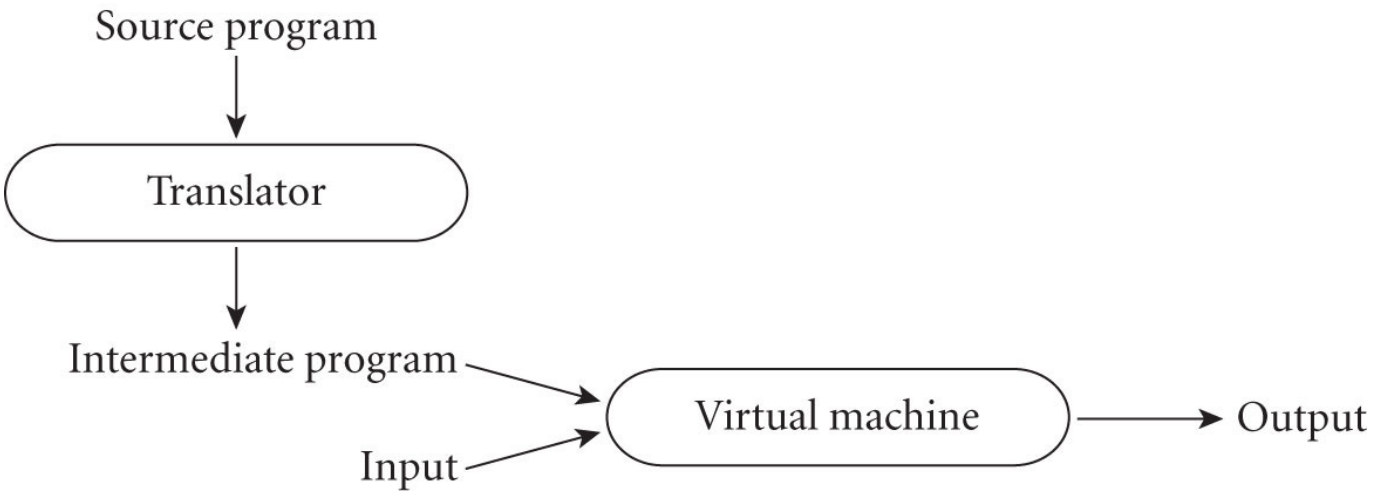


Compilation vs. Interpretation

- Compilation
  - Better performance
    - Early decisions can save time (*early vs. late binding*)
    - Example: a variable's address can be fixed at compile time
- Interpretation:
  - Greater flexibility
    - Example: Lisp, Prolog programs can write new pieces and execute them on the fly
  - Better diagnostics - error messages
    - Source-level debugger

Compilation vs. Interpretation

- Compilation, then interpretation
  - Distinction not very clear; compiled if:
    - Translator analyzes the program thoroughly
    - Intermediate program very different from source
  - Python – interpreted: dynamic semantic error checking
  - C, Fortran – compiled: static semantic error checking



Compilation vs. Interpretation

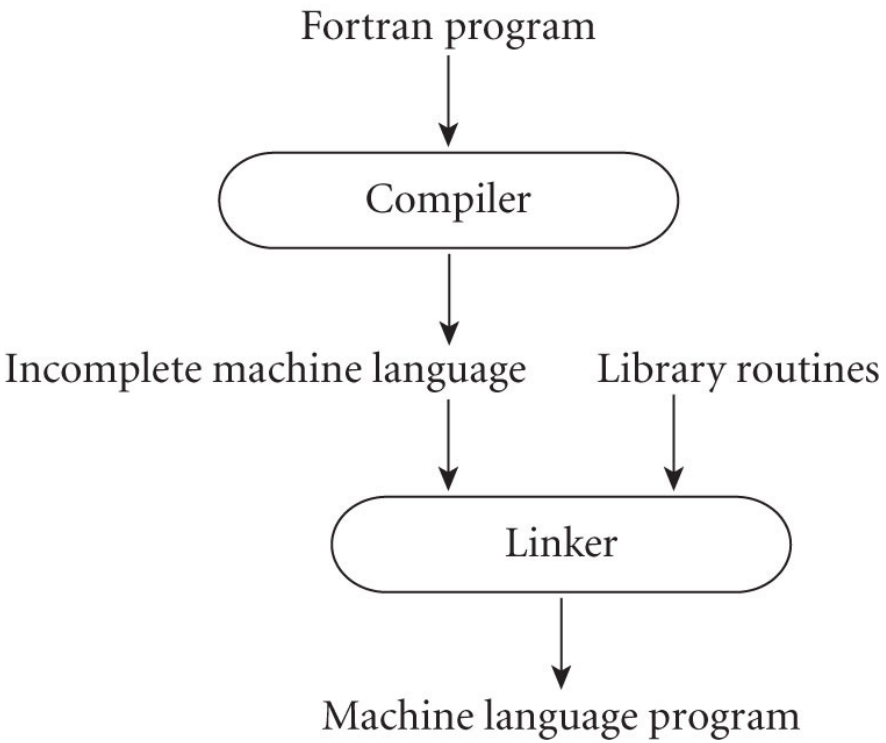
- Compilation
  - Compilation is *translation* from one language into another, with full analysis of the meaning of the input
  - Compilation entails semantic *understanding* of what is being processed; pre-processing does not
  - A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

Implementation strategies

- Preprocessor
  - Used by many interpreted languages
  - 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)

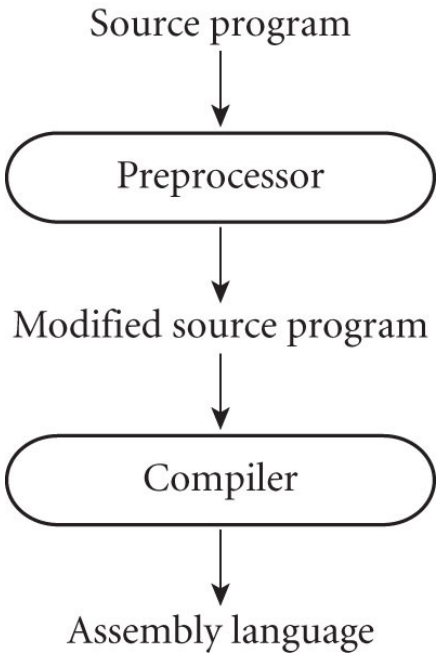
Implementation strategies

- 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:



Implementation strategies

- The C Preprocessor (conditional compilation)
  - Preprocessor deletes comments and expands macros
  - Preprocessor deletes portions of code, which allows several versions of a program be built from same source
    - Example: `#ifdef` directive

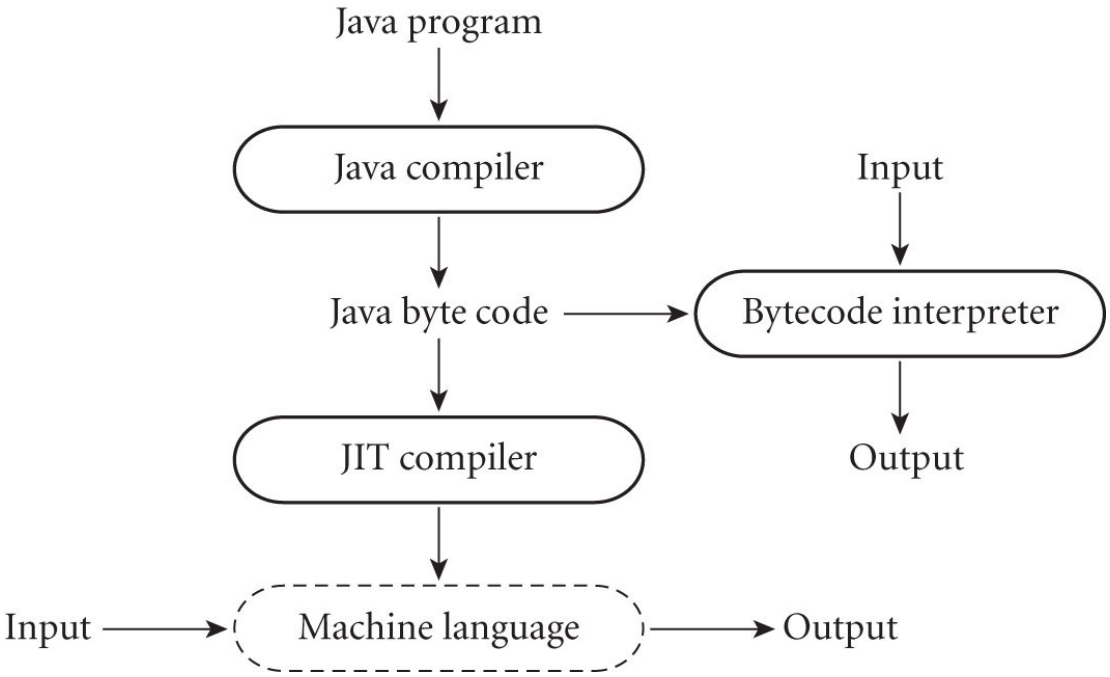


Implementation strategies

- Compilation of Interpreted Languages
  - Interpreted/compiled is a property of the implementation, not of the language
  - Python, Lisp, Prolog, Smalltalk
  - 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

- Just-in-Time Compilation
  - Delay compilation until the last possible moment
    - Java: machine-independent intermediate form – bytecode
      - bytecode is the standard format for distribution of Java programs
    - C# compiler produces Common Intermediate Language (CIL)



Implementation strategies

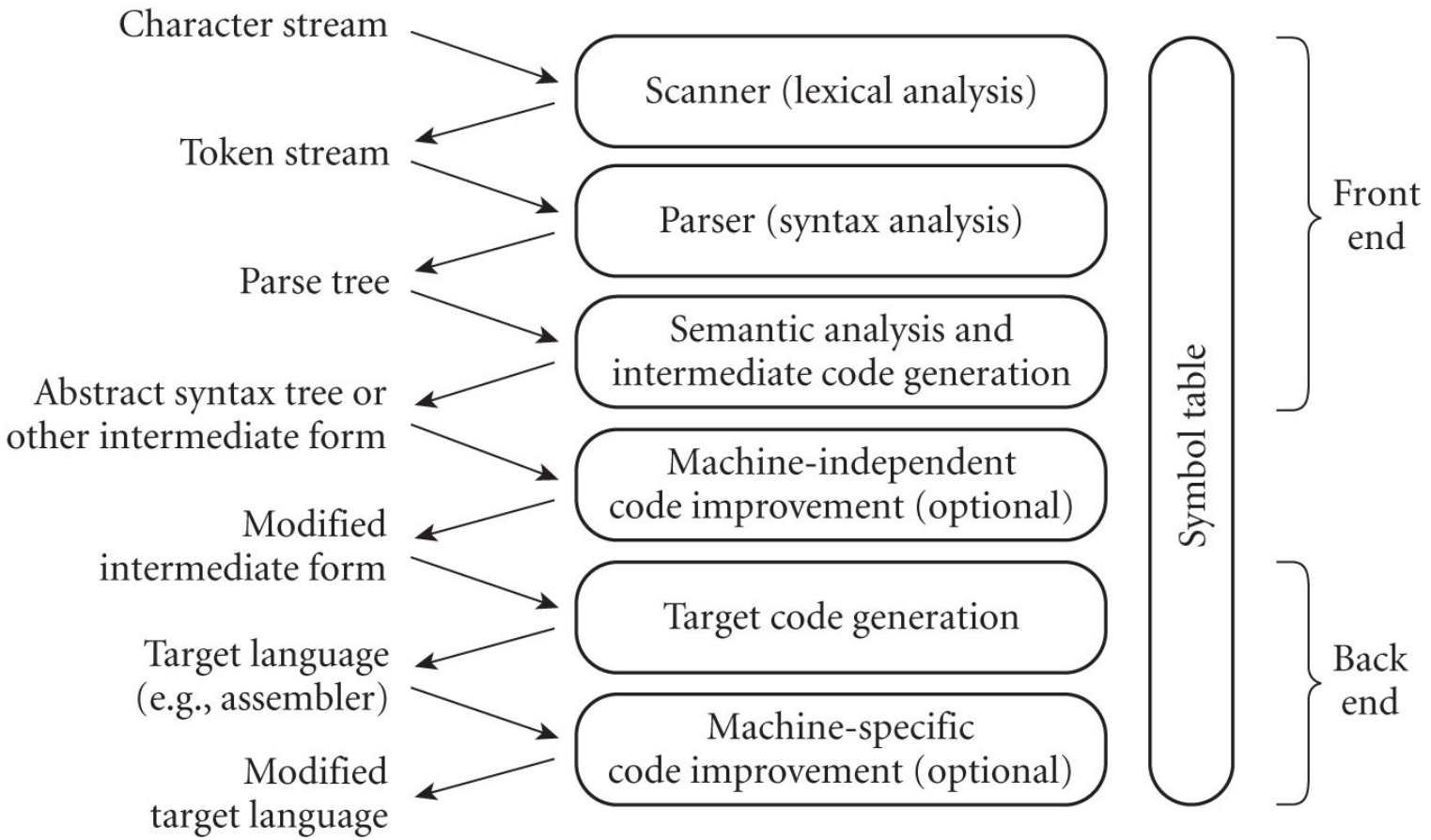
- Unconventional compilers
  - text formatters may compile high-level document description into commands for a printer
    - TEX, LATEX
  - query language processors translate into primitive operations on files
    - SQL

Programming Environment Tools

- Tools
  - Assemblers, debuggers, preprocessors, linkers
  - Editors – can have cross referencing
  - Version management – keep track of separately compiled modules
  - Profilers – performance analysis
  - IDEs – help with everything
    - knowledge of syntax
    - maintain partially compiled internal representation
    - Eclipse, NetBeans, Visual Studio, XCode

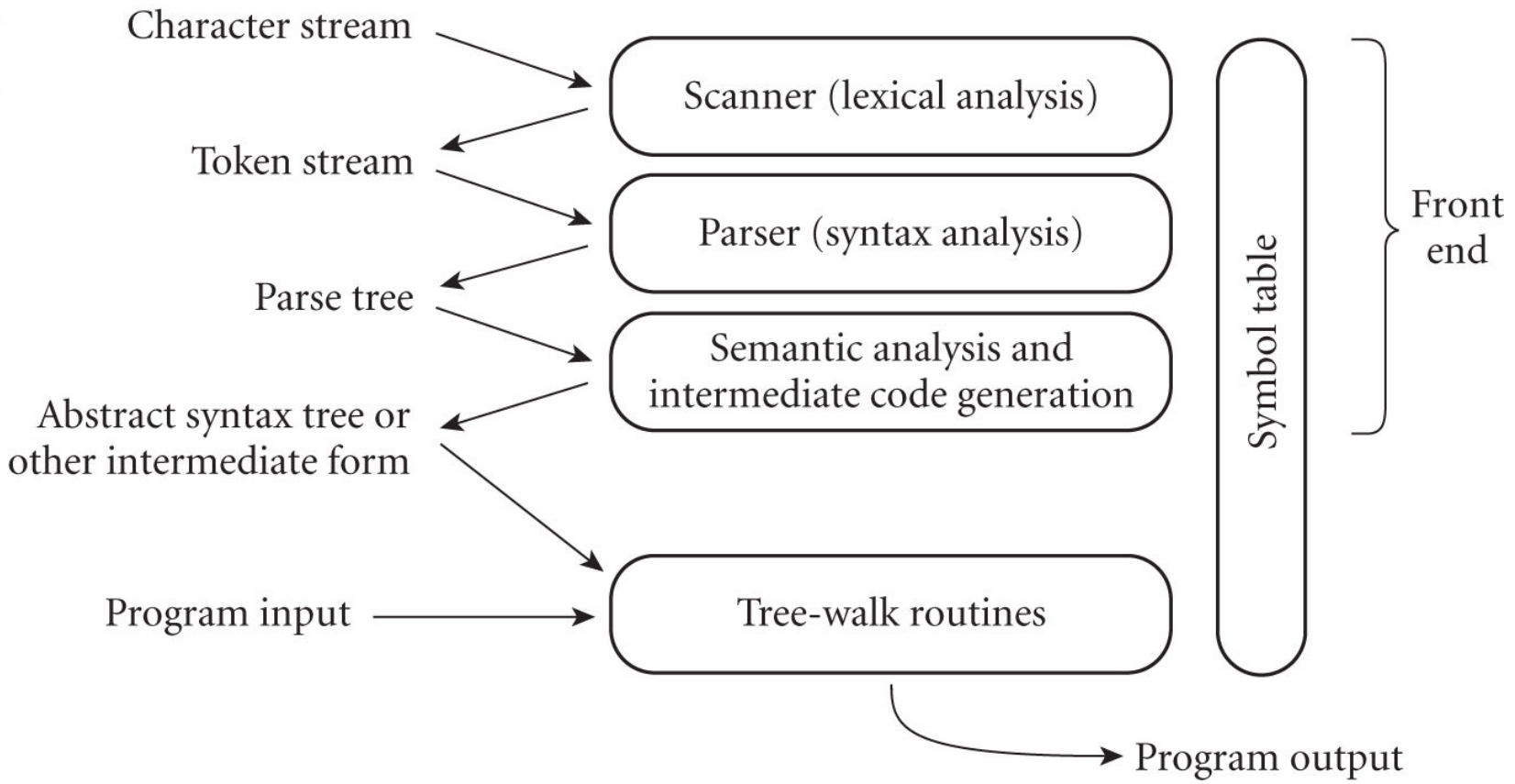
An Overview of Compilation

- Phases of Compilation



An Overview of Interpretation

- Phases of Interpretation



An Overview of Compilation

- Scanning (Lexical Analysis)
  - divide program into "tokens"
    - smallest meaningful units
    - this saves time, since character-by-character processing is slow
  - scanning is recognition of a regular language
    - via a DFA (Deterministic Finite Automaton)

An Overview of Compilation

- Scanning: Example
  - C Program (computes GCD):

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

  - Input – sequence of characters:
    - ‘i’, ‘n’, ‘t’, ‘ ’, ‘m’, ‘a’, ‘i’, ‘n’, ‘(’, ‘)’, …
  - Output – tokens:
    - `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

- Parsing (Syntax Analysis)
  - discovers the structure of the program
  - parsing is recognition of a context-free language
    - via a Push-Down Automaton (PDA)
  - organize tokens into a parse tree
    - higher-level constructs in terms of their constituents
    - as defined by a context-free grammar

An Overview of Compilation

- Parsing: Example – `while` loop in C
  - Context-free grammar (part of):

iteration-statement → `while (` *expression* `)` *statement*

statement → `{` *block-item-list-opt* `}`

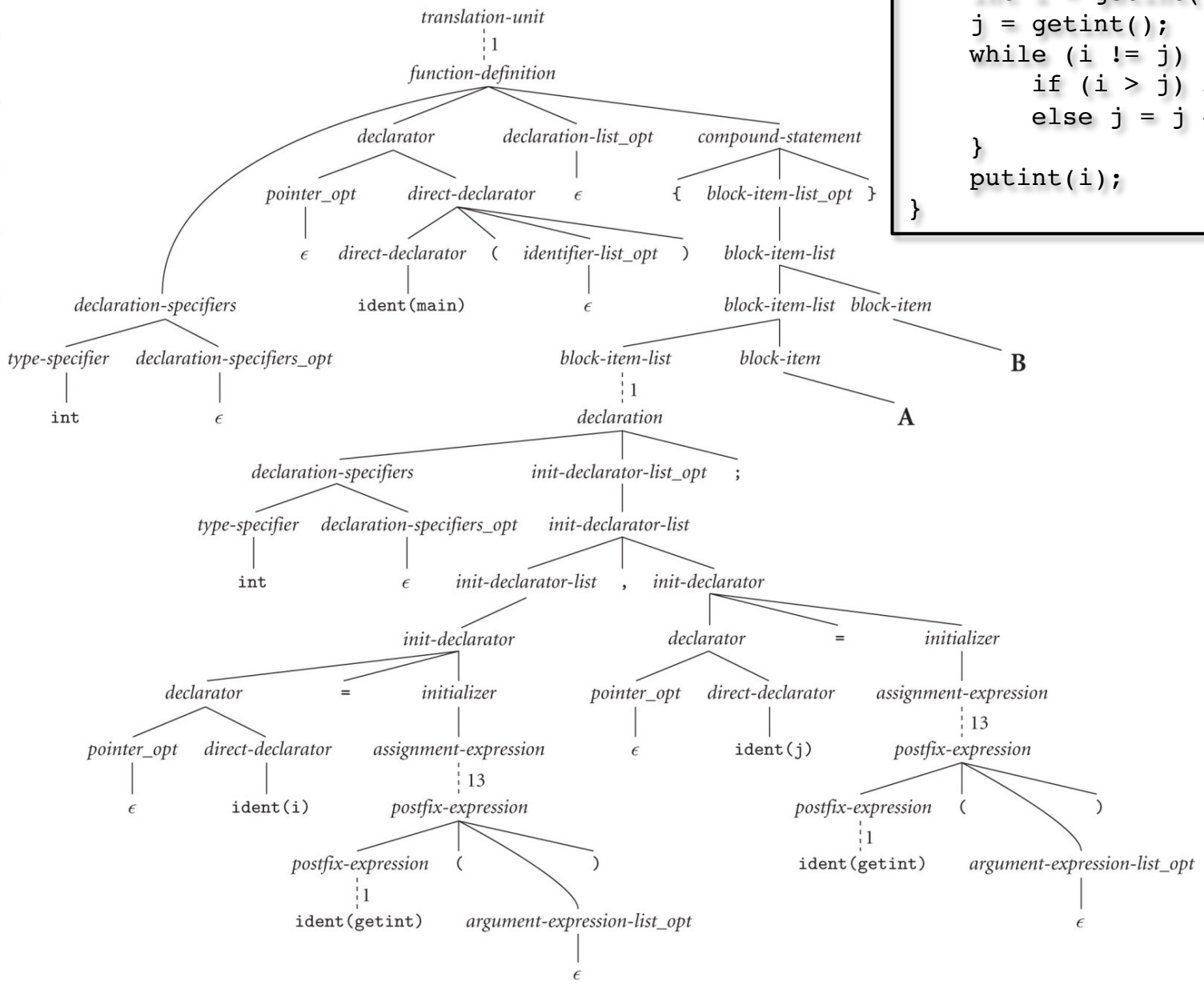
block-item-list-opt → *block-item-list* | ϵ

block-item-list → *block-item*
block-item-list → *block-item-list block-item*

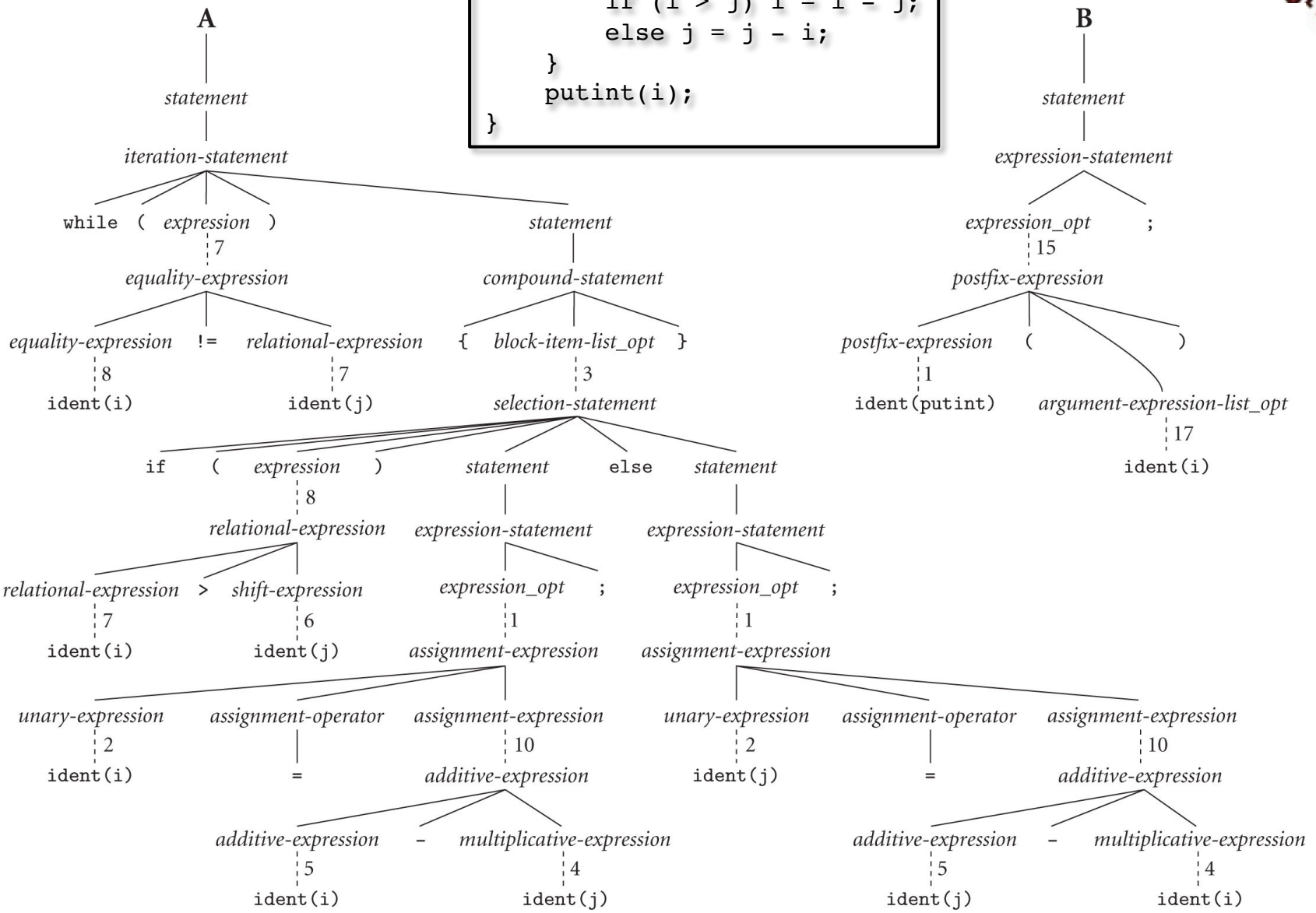
block-item → *declaration*
block-item → *statement*

  - Parse tree for GCD program
    - based on full context-free grammar
    - see next slides

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



```
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

- Semantic Analysis
  - the discovery of meaning in the program
  - detects multiple occurrences of the same identifier
  - tracks the *types* of identifiers and expressions
  - verify consistent usage and guide code generation
  - builds and maintains a *symbol table:*
    - maps each identifier to its information: type, scope, structure, etc.
    - used to check many things
    - Examples in C:
      - identifiers declared before used
      - identifiers used in the appropriate context
      - correct number and type of arguments for subroutines
      - return correct type
      - switch arms have distinct constant labels

An Overview of Compilation

- Semantic Analysis
  - compiler does *static* semantic analysis
  - *dynamic* semantics - for what must be checked at run time
  - Dynamic checks - trade off: safety vs. speed
    - C has very few dynamic checks
  - Examples in other languages:
    - array indexes within bounds
    - variables initialized before used
    - pointers are dereferenced only when referring to valid object
    - arithmetic operations do not overflow
  - Run time checks fail – abort or throw exception

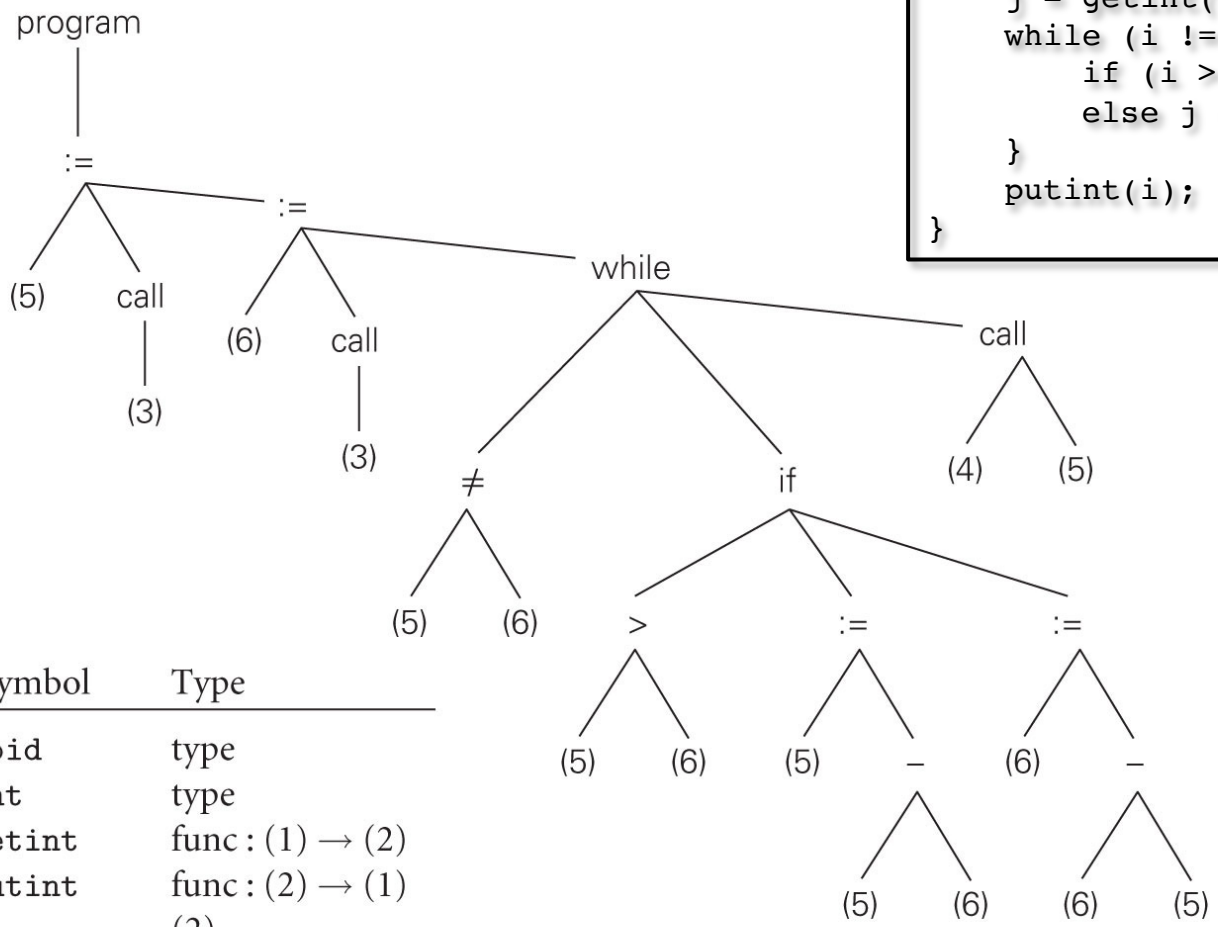
An Overview of Compilation

- Syntax Tree
  - Parse tree = *concrete syntax tree*
    - it shows how the tokens are derived from CFG
    - after that, much information in the parse tree is not relevant
  - Semantic analyzer: parse tree changed into *syntax tree*
    - syntax tree = *abstract syntax tree*
    - removes the “useless” internal nodes
    - annotates the remaining nodes with *attributes*

An Overview of Compilation

- Syntax Tree for GCD program

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



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)

An Overview of Compilation

- Code generation
  - Interpreters use annotated syntax tree to run the program
    - execution means tree traversal
  - Compilers pass the annotated syntax tree as intermediate form to the back end
- Target code generation
  - produces assembly language
  - Example for GCD program – next slide
    - naïve code
    - good code is difficult to produce
    - That's why you'll always find good jobs!

An Overview of Compilation

```
    pushl   %ebp                # \
    movl    %esp, %ebp          #  ) reserve space for local variables
    subl    $16, %esp           # /
    call    getint              # read
    movl    %eax, -8(%ebp)      # store i
    call    getint              # read
    movl    %eax, -12(%ebp)     # store j
A:  movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    je      D                   # jump if i == j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    jle     B                   # jump if i < j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    subl    %ebx, %edi          # i = i - j
    movl    %edi, -8(%ebp)      # store i
    jmp     C
B:  movl    -12(%ebp), %edi     # load j
    movl    -8(%ebp), %ebx      # load i
    subl    %ebx, %edi          # j = j - i
    movl    %edi, -12(%ebp)     # store j
C:  jmp     A
D:  movl    -8(%ebp), %ebx      # load i
    push    %ebx                # push i (pass to putint)
    call    putint              # write
    addl    $4, %esp            # pop i
    leave                       # deallocate space for local variables
    mov     $0, %eax            # exit status for program
    ret                         # return to operating system
```

```
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

- Optimization (code improvement)
  - takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - The code on the previous slide becomes:

```
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $4, %esp
    andl    $-16, %esp
    call    getint
    movl    %eax, %ebx
    call    getint
    cmpl    %eax, %ebx
    je      C
A:  cmpl    %eax, %ebx
    jle     D
    subl    %eax, %ebx
B:  cmpl    %eax, %ebx
    jne     A
C:  movl    %ebx, (%esp)
    call    putint
    movl    -4(%ebp), %ebx
    leave
    ret
D:  subl    %ebx, %eax
    jmp     B
```

Programming Language Syntax

- Scanning -

Chapter 2, Sections 2.1-2.2

Regular Expressions

- *Token*: a shortest string of characters with meaning
- Tokens – specified by regular expressions
- An *alphabet* Σ is any finite nonempty set
  - Examples:
  - English: {a, b, …, z},
  - binary: {0, 1}
  - {a, b, …, z, 0, 1, …, 9, ·, |, *, ε}
- The set of all finite strings over Σ is denoted Σ^*
- The *empty* string: $\varepsilon \in \Sigma^*$ (has zero characters)

Regular Expressions

- Regular expressions
- Regular expressions over an alphabet Σ are all strings obtained as follows:
  - ε is a regular expression
  - any character $a \in \Sigma$ is a regular expression
  - For reg. exp. α, β, the following are reg. exp.:
    - $\alpha \cdot \beta$ - *concatenation* ('$\cdot$' omitted: $\alpha\beta$)
    - $\alpha \mid \beta$ - *union* ('|' = or) (sometimes denoted $\alpha + \beta$)
    - α^* - *Kleene star* (0 or more repetitions)
      - $\alpha^+ = \alpha\, \alpha^*$ (1 or more repetitions)

Regular Expressions

- Example: Signed integers:

  sign_int → (+ | – | ε)(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)*

- Example: Numerical constants:

  number → *integer* | *real*

  integer → *digit digit**

  real → *integer exponent* | *decimal* (*exponent* | ε)

  decimal → *digit** (**.** *digit* | *digit* **.**) *digit**

  exponent → (**e** | **E**) (+ | – | ε) *integer*

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

- '→' means "can be"
- Precedence order: '*' > '.' > '|'

Regular Expressions

- Other applications:
  - `grep` family of tools in Unix
  - many editors
  - scripting languages:
    - Perl
    - Python
    - Ruby
    - `awk`
    - `sed`

Formatting issues

- Upper vs. lower case
  - distinct in some languages: C, Python, Perl
  - same in others: Fortran, Lisp, Ada
- Identifiers: letters, digits, underscore (most languages)
  - camel case: someIdentifierName
  - underscore: some_identifier_name
- Unicode
  - non-Latin characters have become important
- White spaces
  - usually ignored
  - separate statements: Python, Haskell, Go, Swift
  - indentation important: Python, Haskell

Context-Free Grammars

Context Free Grammar (CFG)

- CFG consists of:
  - A set of *terminals, T*
  - A set of *non-terminals, N*
  - A *start symbol, S* $\in N$
  - A set of *productions;* subset of $N \times (N \cup T)^*$

- Example: Balanced parentheses:

$S \rightarrow \varepsilon$
$S \rightarrow SS$
$S \rightarrow (S)$

Context-Free Grammars

- Example: CFG for arithmetic expressions:

  expr → `id` | `number` | `-` *expr* | `(` *expr* `)` | *expr op expr*

  op → `+` | `-` | `*` | `/` |

- *Derivation*: start with *S*, continue with productions
  - replace LHS nonterminal by the RHS
- Example: generate the string: `slope * x + intercept`

expr	⟹	*expr op expr*	(*S* = *expr*)
	⟹	*expr op* `id`	('⟹' = "derives")
	⟹	*expr* `+ id`	('⟹*' = 0 or more steps)
	⟹	*expr op expr* `+ id`	
	⟹	*expr op* `id + id`	
	⟹	*expr* `* id + id`	
	⟹	`id * id + id`	
		`(slope) (x) (intercept)`	

- Sentential form: any string along the way

Context-Free Grammars

- *Right-most derivation*: the rightmost nonterminal is replaced

$$
\begin{aligned}
\underline{expr} &\Rightarrow expr\ op\ \underline{expr} \\
&\Rightarrow expr\ \underline{op}\ \texttt{id} \\
&\Rightarrow \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Rightarrow expr\ op\ \underline{expr}\ \texttt{+}\ \texttt{id} \\
&\Rightarrow expr\ \underline{op}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Rightarrow \underline{expr}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Rightarrow \texttt{id}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id}
\end{aligned}
$$

- *Left-most derivation*: the leftmost nonterminal is replaced

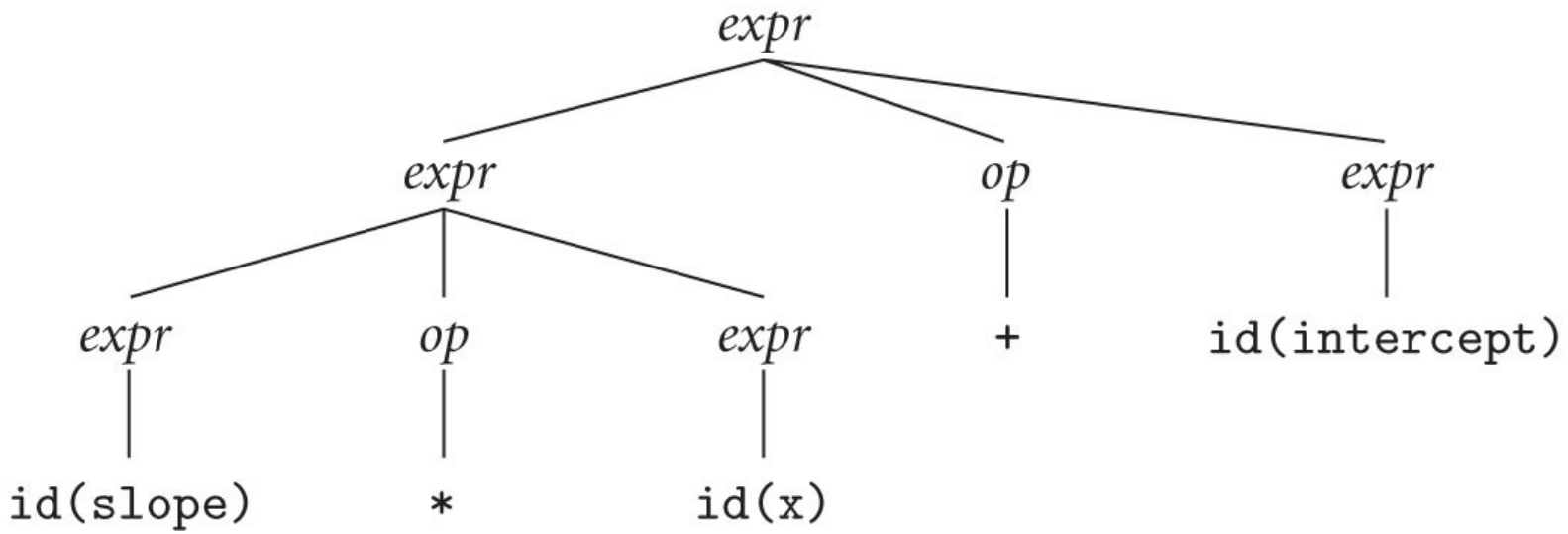
$$
\begin{aligned}
\underline{expr} &\Rightarrow \underline{expr}\ op\ expr \\
&\Rightarrow \underline{expr}\ op\ expr\ op\ expr \\
&\Rightarrow \texttt{id}\ \underline{op}\ expr\ op\ expr \\
&\Rightarrow \texttt{id}\ \texttt{*}\ \underline{expr}\ op\ expr \\
&\Rightarrow \texttt{id}\ \texttt{*}\ \texttt{id}\ \underline{op}\ expr \\
&\Rightarrow \texttt{id}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \underline{expr} \\
&\Rightarrow \texttt{id}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id}
\end{aligned}
$$

Context-Free Grammars

Parse Tree

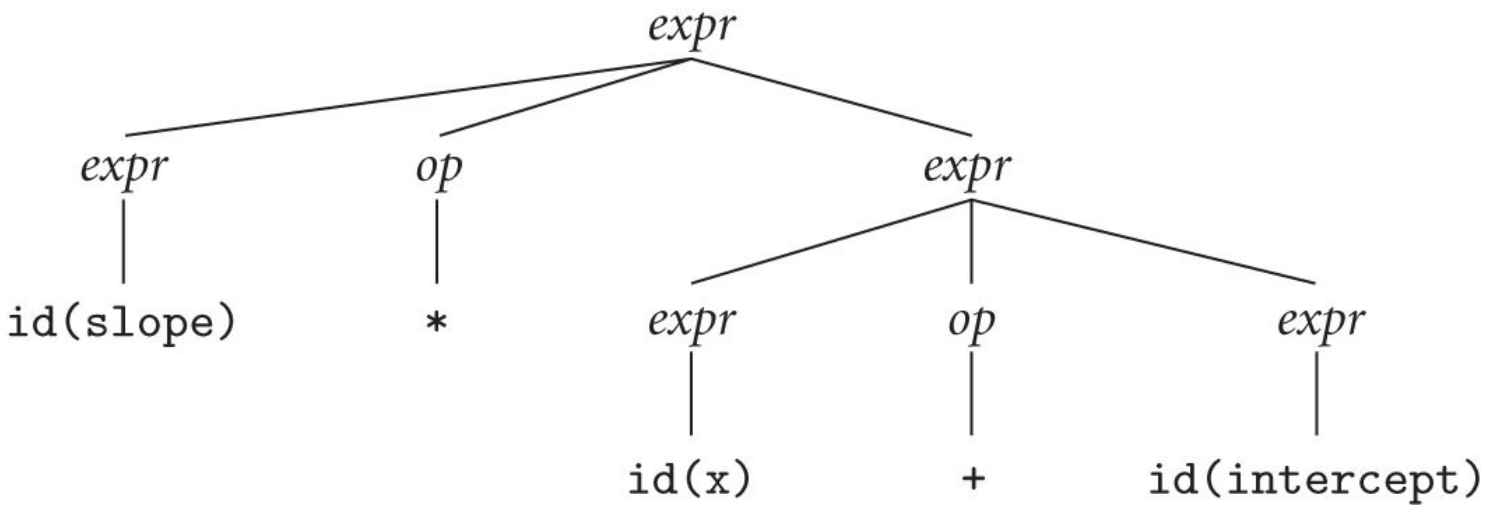
- Represents a derivation graphically
- Example: Parse tree for the string:

```
slope * x + intercept
```



Context-Free Grammars

- Different parse tree for: `slope * x + intercept`
- Tree allowed by the grammar but incorrect for the expression



- *Ambiguous* grammar: two different parse trees for one string
  - Ambiguity is a problem for parsers
  - We want unambiguous grammars

Context-Free Grammars

- Better version – unambiguous
- Captures *associativity* and *precedence*

expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

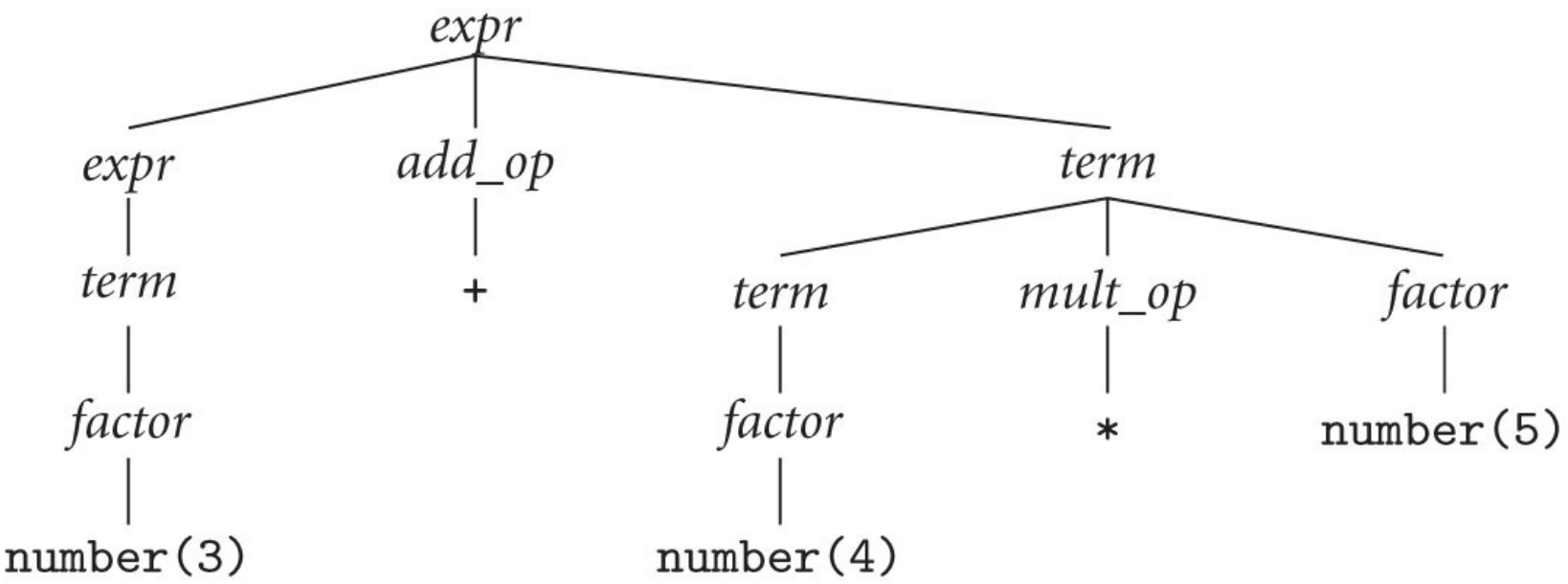
factor → `id` | `number` | `-` *factor* | `(` *expr* `)`

add_op → `+` | `-`

mult_op → `*` | `/`

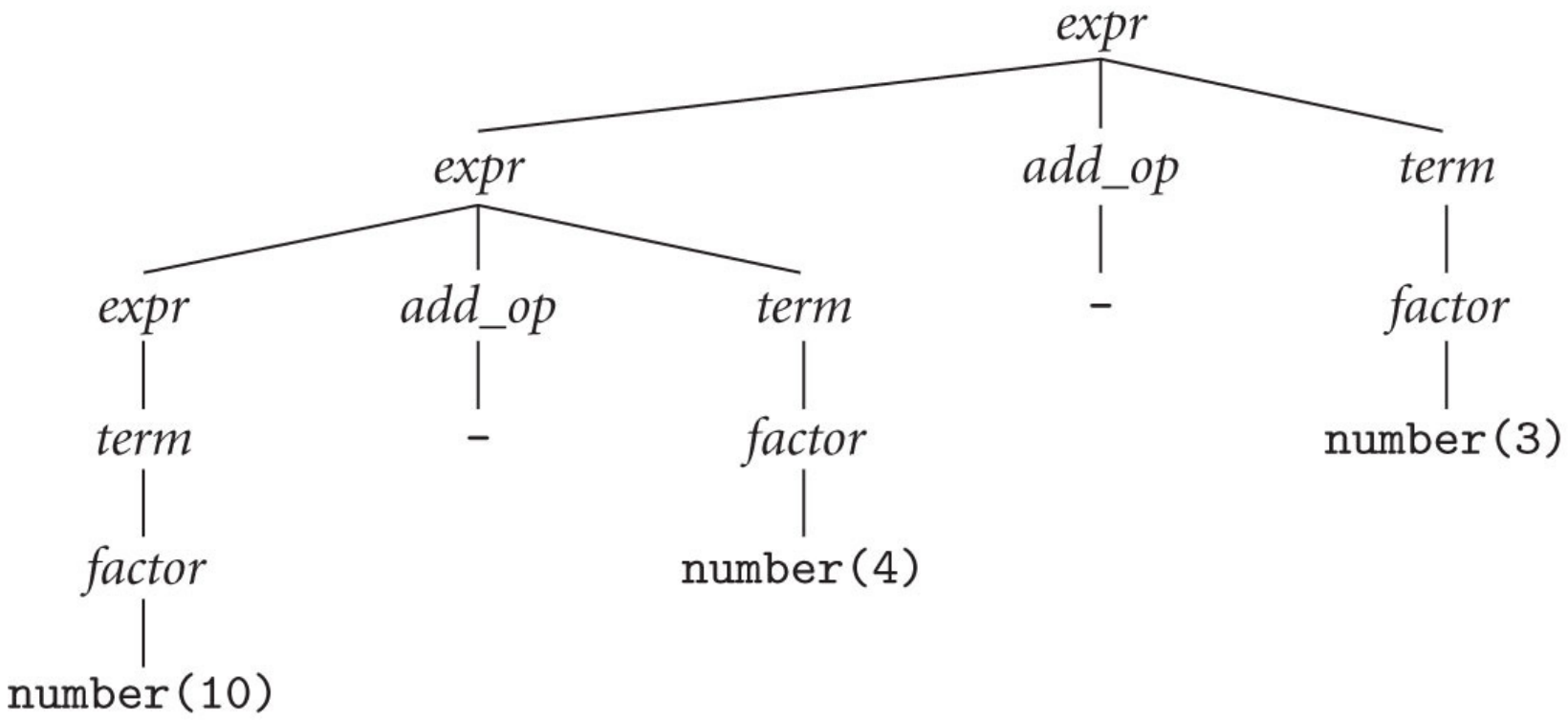
Context-Free Grammars

- Parse tree for: `3 + 4 * 5`
  - Precedence rules



Context-Free Grammars

- Parse tree for: `10 - 4 - 3`
  - Left-associativity rules



Scanning

Scanning = Lexical Analysis

- tokenizing source
- removing comments
- saving text of identifiers, numbers, strings
- saving source locations (file, line, column) for error messages

Scanning

- Example: simple calculator language

assign → `:=` (Algol style; C has '=')
plus → `+`
minus → `-`
times → `*`
div → `/`
lparen → `(`
rparen → `)`
id → *letter* (*letter* | *digit*)* (except for `read` and `write`)
number → *digit digit** | *digit** (`.` *digit* | *digit* `.`) *digit**
comment → `/*` (*non-*`*` | `*` *non-*`/`)* `*`+`/`
| `//` (*non-newline*)* *newline*

Scanning

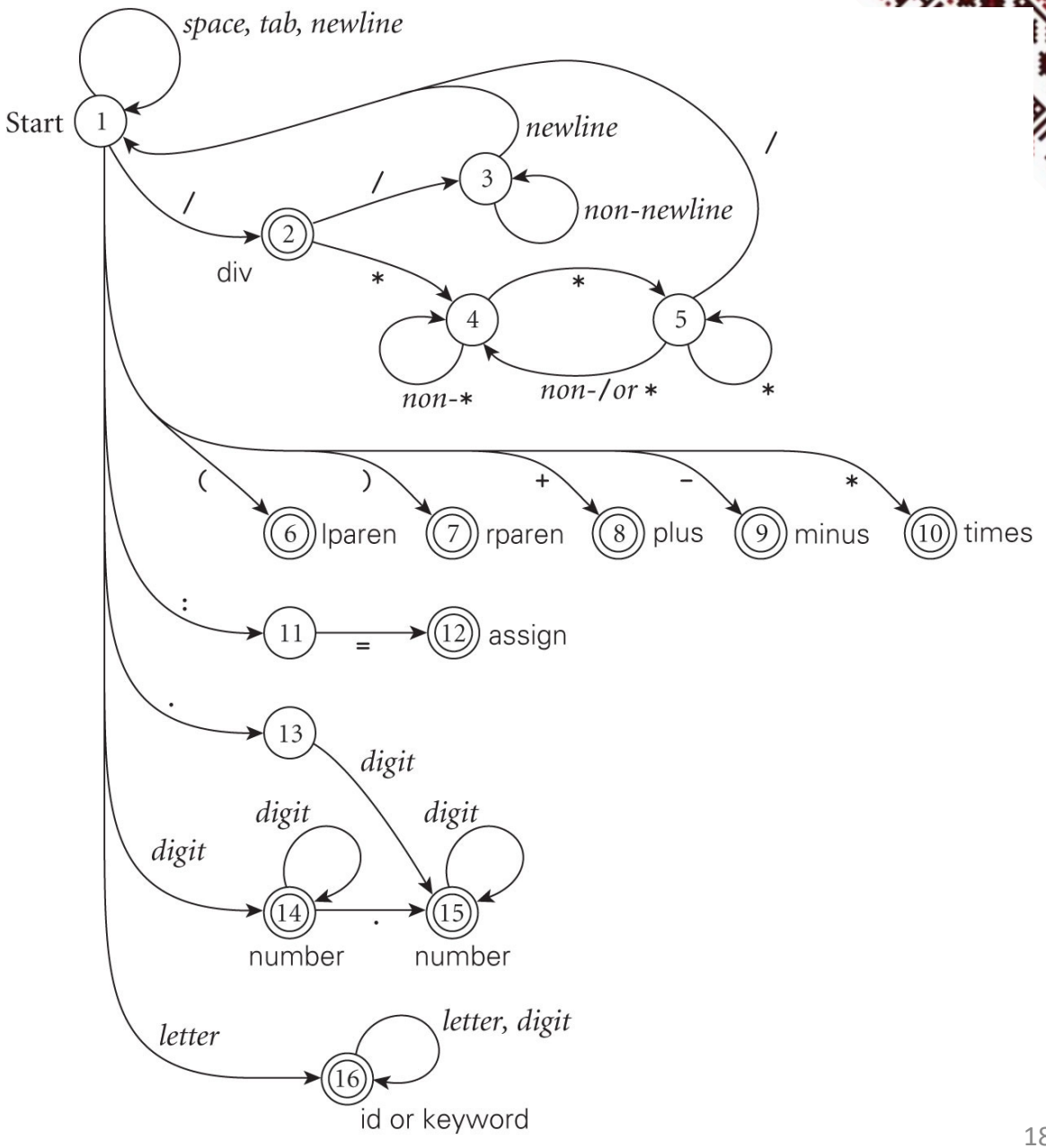
Ad-hoc scanner

- Longest possible token extracted
- White spaces are delimiters

```
skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '-', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return assign else announce an error
if cur_char = '/'
    peek at the next character
    if it is '*' or '/'
        read additional characters until "*/" or newline is seen, respectively
        jump back to top of code
    else return div
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error
```

Scanning

- Structured scanner
- DFA – Deterministic Finite Automaton
- Separate final state for each token category

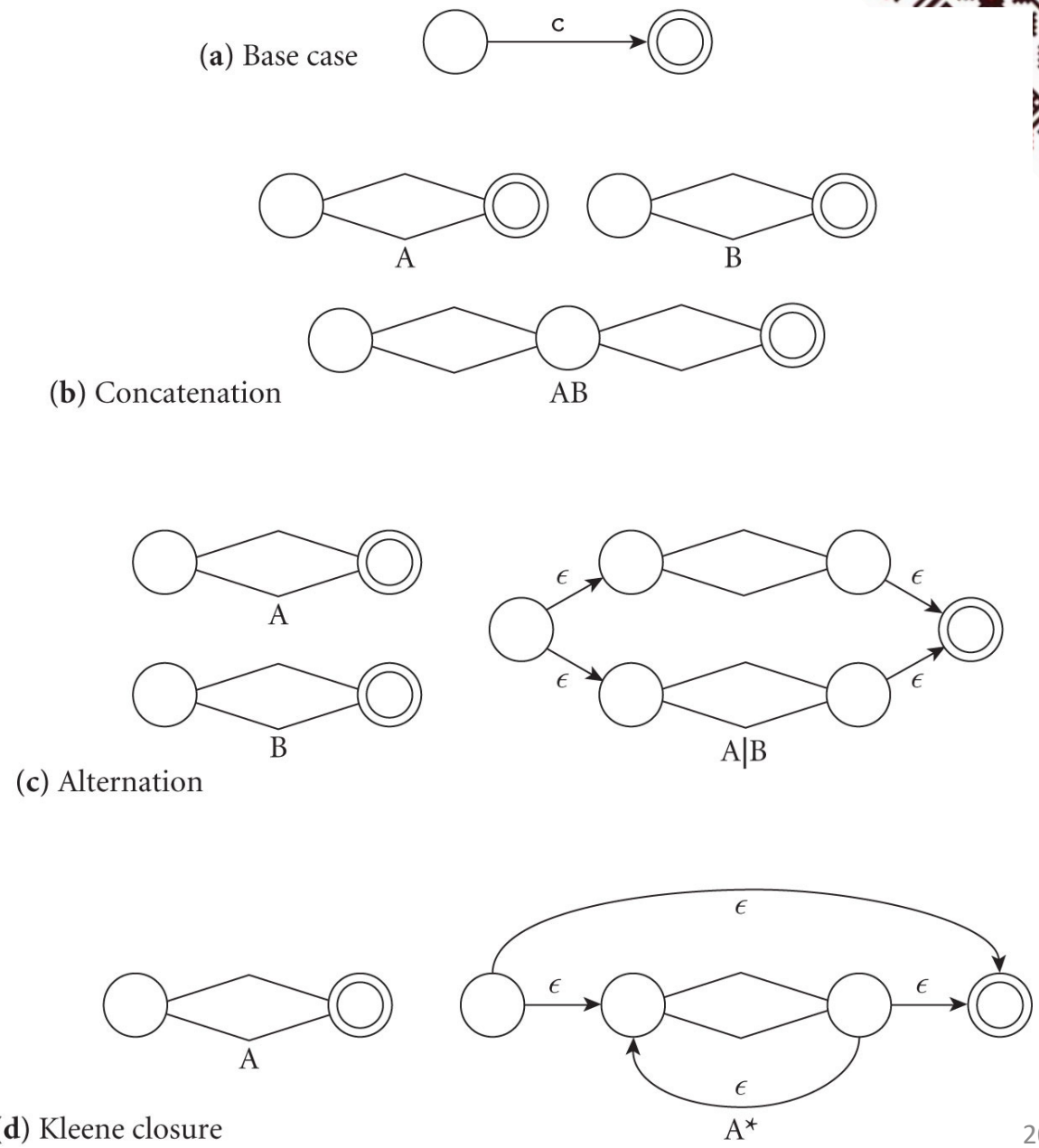


Scanning

- DFA
  - Built automatically from regular expressions
  - Tools: `lex`, `flex`, `scagen`
  - Difficult to build directly
    - build first an NFA – Nondeterministic FA
    - convert to DFA
    - minimize DFA (smallest number of states)

Scanning

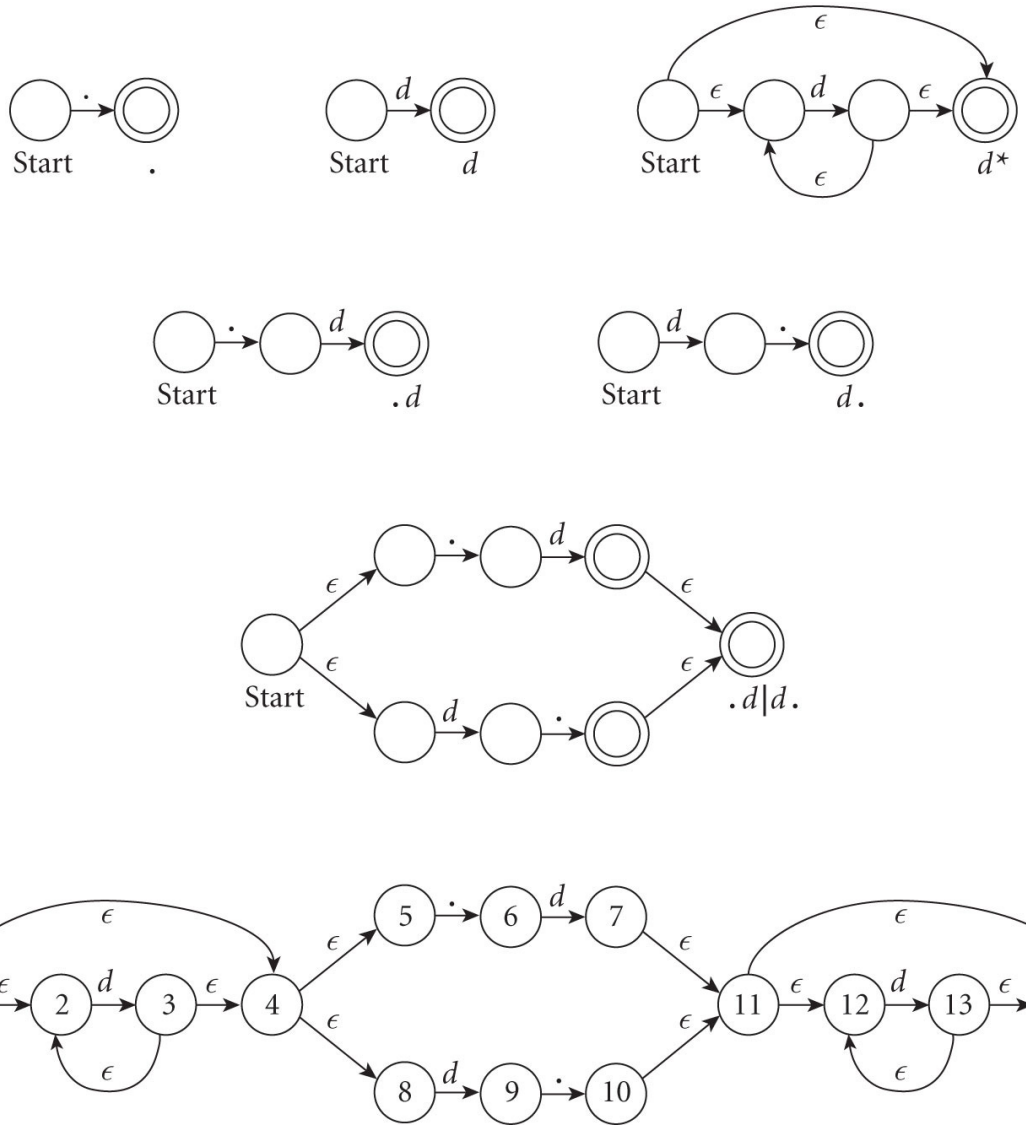
- Reg.exp. to NFA
- Follows the structural definition of regular expressions



Scanning

- Reg.exp. to NFA
- Example:

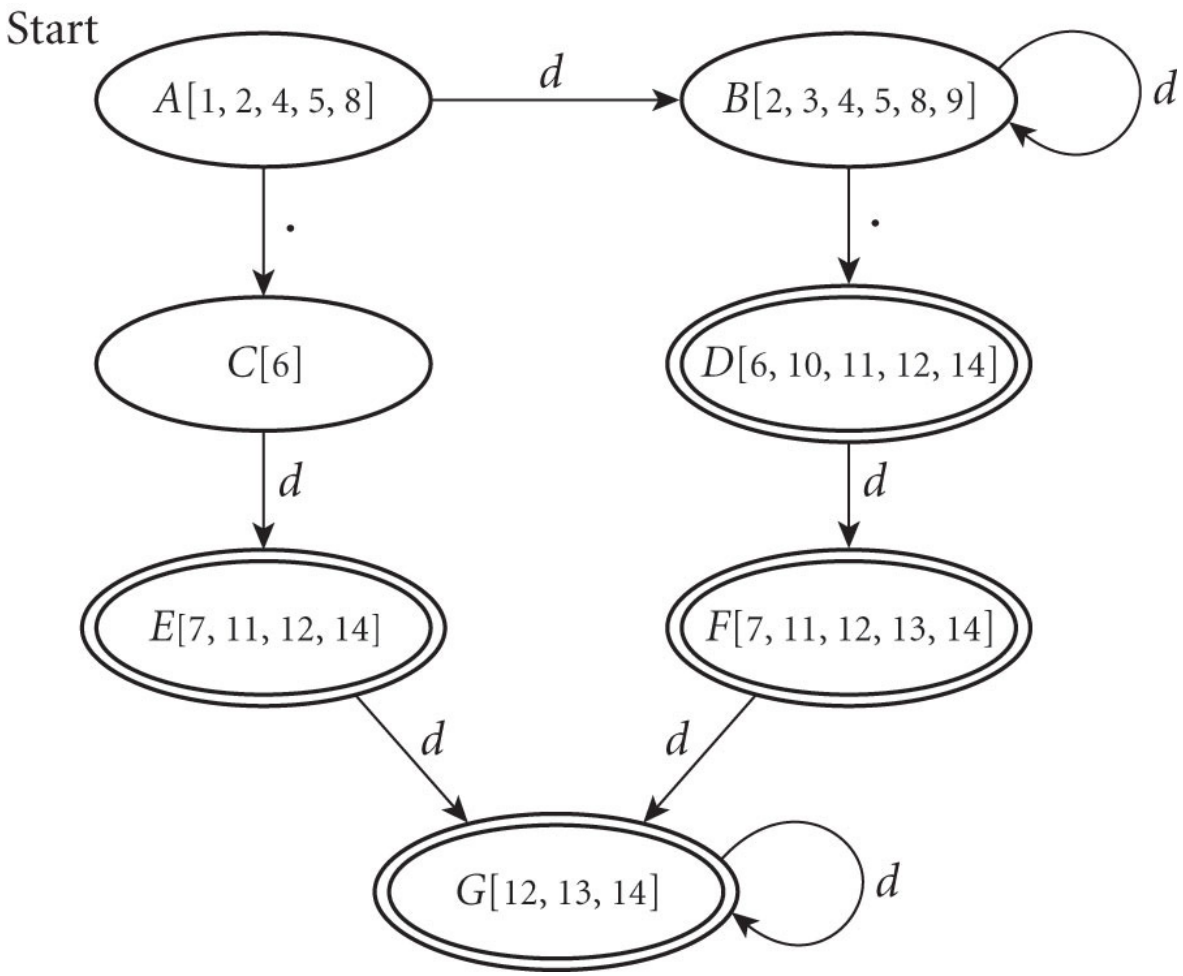
d^* (. d | d .) d^*



Scanning

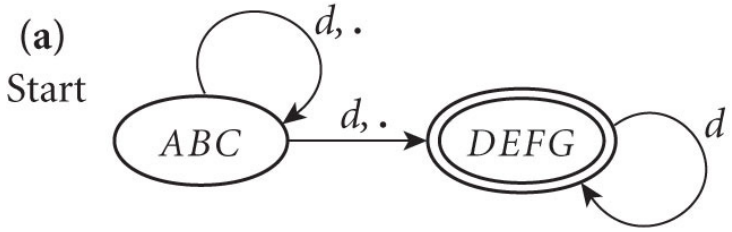
- NFA to DFA
- Example:

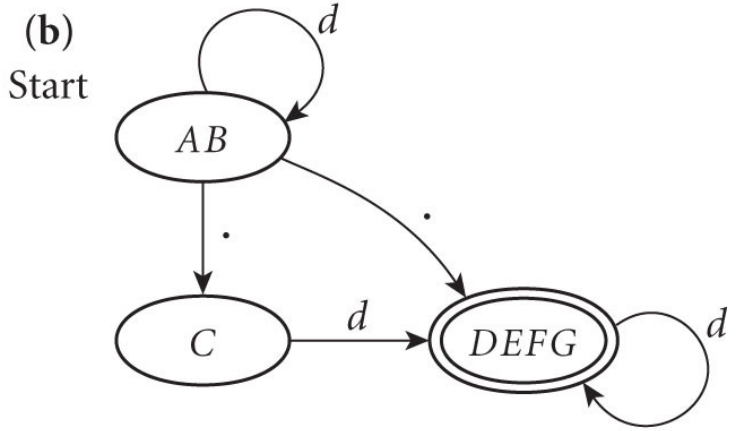
d^* (. d | d .) d^*

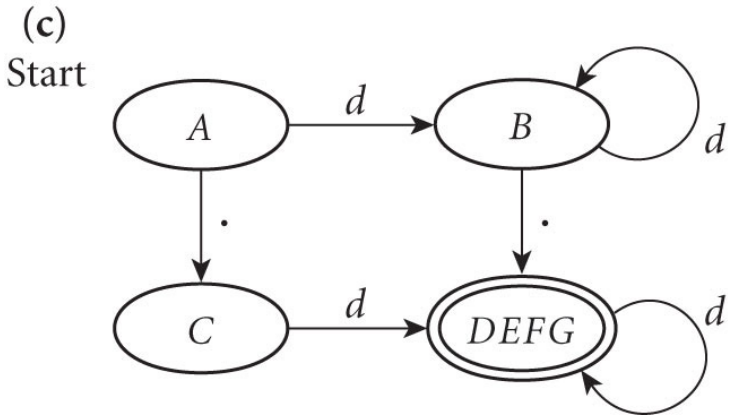


Scanning

- DFA minimization
- Example: d^* (. d | d .) d^*







Scanning

- Scanners are built three ways:
  - ad-hoc:
    - fastest, most compact code
  - semi-mechanical pure DFA
    - nested `case` statements
  - table-driven DFA
    - automatically-generated scanners
- "Longest-possible token" rule
  - return only when next character cannot continue current token
  - the next character needs to be saved for the next token
- Keywords
  - DFA would need many states to identify
  - Better treat keywords as exceptions to the identifier rule

Scanning

- Nested `case` statement DFA

```
state := 1
loop
   read cur_char
   case state of
      1: case cur_char of
            ' ', '\t', '\n':     …
            'a' … 'z':           …
            '0' … '9':           …
            '>' :                …
             …
      2: case cur_char of
            …
      ⋮
      n: case cur_char of
            …
```

Scanning

- Look-ahead
- May need to peek at more than one character
- *look-ahead* – characters necessary to decide
- Example: Pascal
  - have 3 so far and see '.'
  - 3.14 or 3..5 may follow
- Example: Fortran
  - arbitrarily long look-ahead
  - `DO 5 I = 1,25`
    - execute statements up to 5 for I from 1 to 25
  - `DO 5 I = 1.25`
    - assign 1.25 to the variable `DO5I`
  - NASA's Mariner 1 may have been lost due to '`.`' i.o '`,`'
  - Fortran 77 has better syntax: `DO 5,I = 1,25`

Scanning

- Table-driven scanning

```
state = 0 . . number_of_states
token = 0 . . number_of_tokens
scan_tab : array [char, state] of record
      action : (move, recognize, error)
      new_state : state
token_tab : array [state] of token          -- what to recognize
keyword_tab : set of record
      k_image : string
      k_token : token
-- these three tables are created by a scanner generator tool
```

(continued on next slide)

Scanning

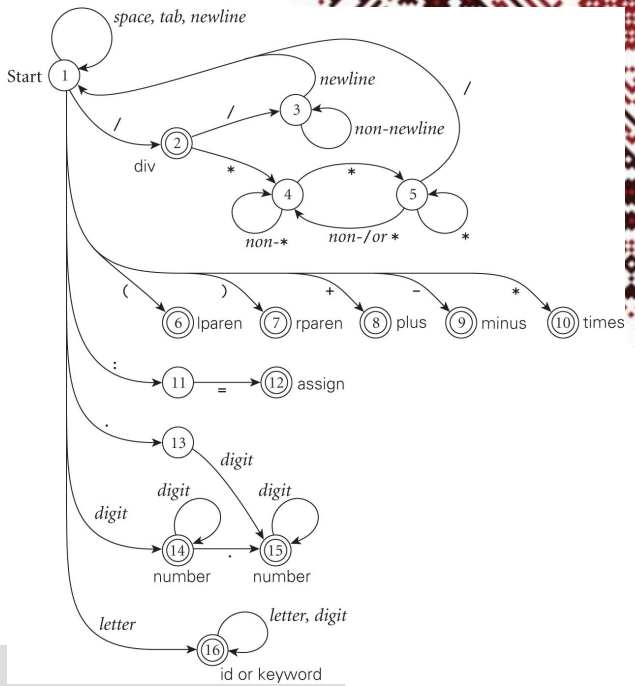
Table-driven scanning (cont'd)

```
state = 0 . . number_of_states
token = 0 . . number_of_tokens
scan_tab : array [char, state] of record
    action : (move, recognize, error)
    new_state : state
token_tab : array [state] of token
keyword_tab : set of record
    k_image : string
    k_token : token
```

```
tok : token
cur_char : char
remembered_chars : list of char
repeat
    cur_state : state := start_state
    image : string := null
    remembered_state : state := 0        -- none
    loop
        read cur_char
        case scan_tab[cur_char, cur_state].action
            move:
                if token_tab[cur_state] ≠ 0
                    -- this could be a final state
                    remembered_state := cur_state
                    remembered_chars := ϵ
                add cur_char to remembered_chars
                cur_state := scan_tab[cur_char, cur_state].new_state
            recognize:
                tok := token_tab[cur_state]
                unread cur_char          -- push back into input stream
                exit inner loop
            error:
                if remembered_state ≠ 0
                    tok := token_tab[remembered_state]
                    unread remembered_chars
                    remove remembered_chars from image
                    exit inner loop
                -- else print error message and recover; probably start over
        append cur_char to image
    -- end inner loop
until tok ∉ {white_space, comment}
look image up in keyword_tab and replace tok with appropriate keyword if found
return ⟨tok, image⟩
```

Scanning

- Scanner table used by previous code
  - state 17: white spaces; state 18: comments
  - `scan_tab`: entire table but last column
  - `token_tab`: last column
  - `keyword_tab` = {**`read`**, **`write`**}



State	Current input character														
	space, tab	*newline*	/	*	(	)	+	–	:	=	.	*digit*	*letter*	*other*	
1	17	17	2	10	6	7	8	9	11	–	13	14	16	–	
2	–	–	3	4	–	–	–	–	–	–	–	–	–	–	div
3	3	18	3	3	3	3	3	3	3	3	3	3	3	3	
4	4	4	4	5	4	4	4	4	4	4	4	4	4	4	
5	4	4	18	5	4	4	4	4	4	4	4	4	4	4	
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	lparen
7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	rparen
8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	plus
9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	minus
10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	times
11	–	–	–	–	–	–	–	–	–	12	–	–	–	–	
12	–	–	–	–	–	–	–	–	–	–	–	–	–	–	assign
13	–	–	–	–	–	–	–	–	–	–	–	15	–	–	
14	–	–	–	–	–	–	–	–	–	–	15	14	–	–	number
15	–	–	–	–	–	–	–	–	–	–	–	15	–	–	number
16	–	–	–	–	–	–	–	–	–	–	–	16	16	–	identifier
17	17	17	–	–	–	–	–	–	–	–	–	–	–	–	white_space
18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	comment

Scanning

- Lexical errors
- Very few – most strings correspond to some token
- Should recover to enable the compiler to detect more errors
  - throw away the current, invalid, token
  - skip forward to the next possible beginning of a new token
  - restart the scanning algorithm
  - count on the error-recovery mechanism of the parser to cope with a syntactically invalid sequence of tokens

Programming Language Syntax

- LL parsing -

Chapter 2, Section 2.3

Parsing

- Parser
  - in charge of the entire compilation process
    - *Syntax-directed translation*
  - calls the scanner to obtain tokens
  - assembles the tokens into a syntax tree
  - passes the tree to the later phases of the compiler
    - semantic analysis
    - code generation
    - code improvement
  - a parser is a language *recognizer*
  - context-free grammar is a language *generator*

Parsing

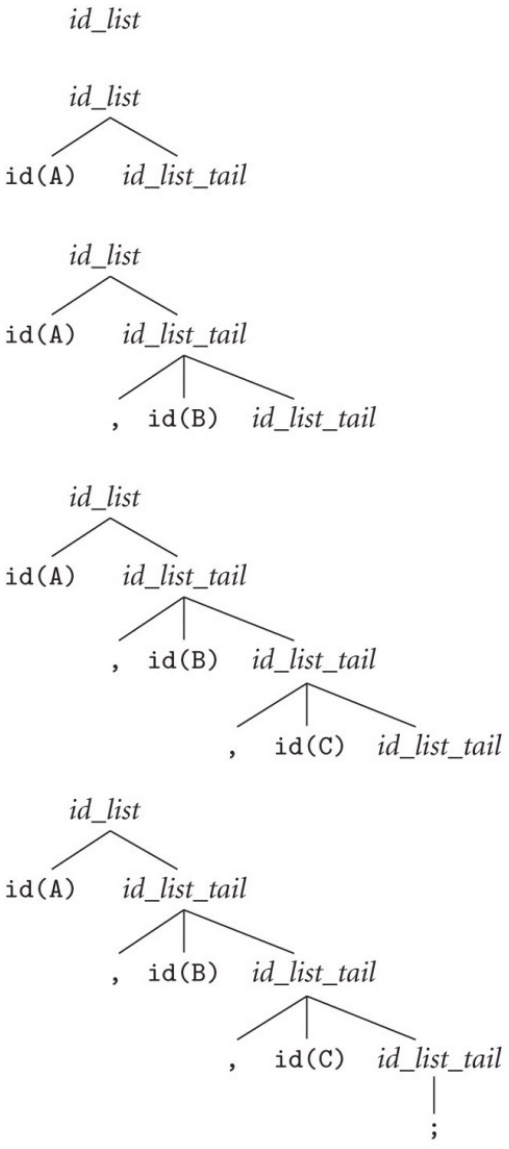
- Context-free language recognition
  - Earley, Cocke-Younger-Kasami alg's
  - $O(n^3)$ time
    - too slow
  - There are classes of grammars with $O(n)$ parsers:
    - **LL**: '**L**eft-to-right, **L**eftmost derivation'.
    - **LR**: '**L**eft-to-right, **R**ightmost derivation′

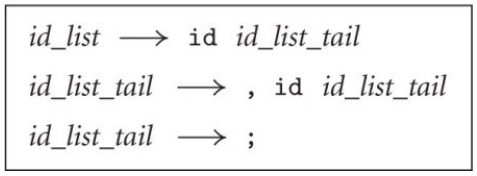
Class	Direction of scanning	Derivation discovered	Parse tree construction	Algorithm used
LL	left-to-right	left-most	top-down	predictive
LR	left-to-right	right-most	bottom-up	shift-reduce

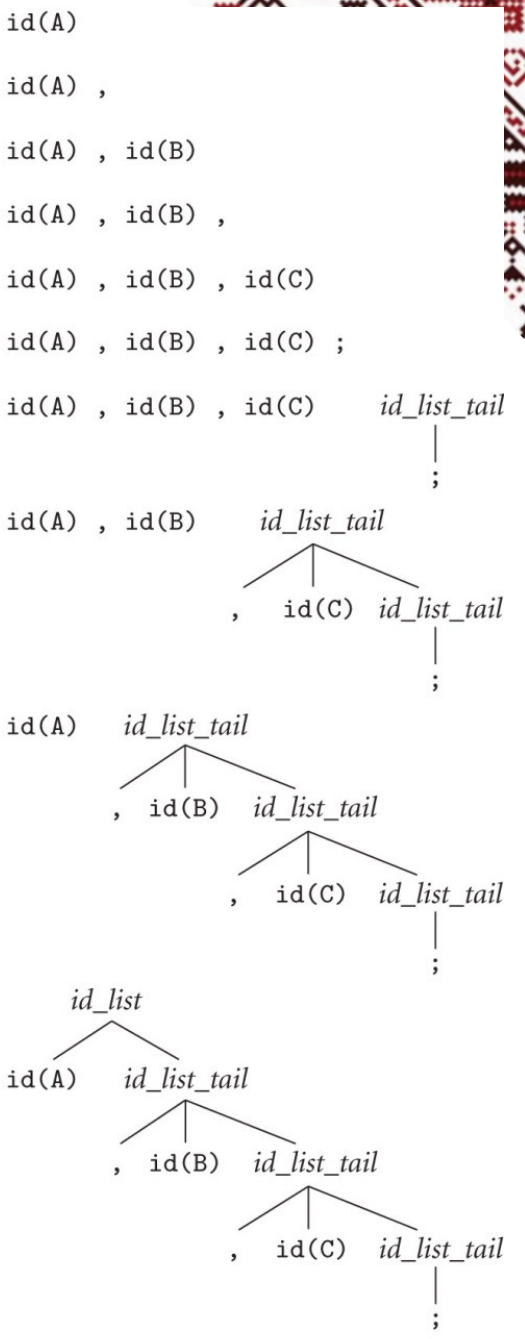
Parsing

- Top-down vs. Bottom-up
- Top-down
  - *predict* based on next token
- Bottom-up
  - *reduce* right-hand side
  - Example:

```
A, B, C;
```



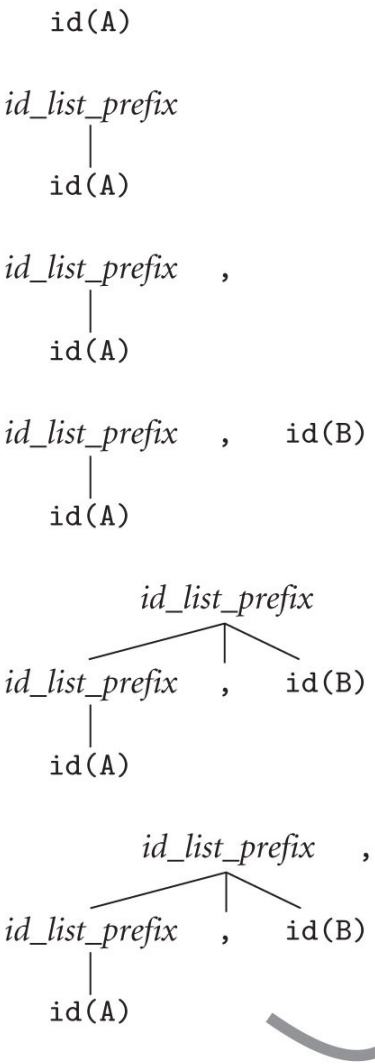


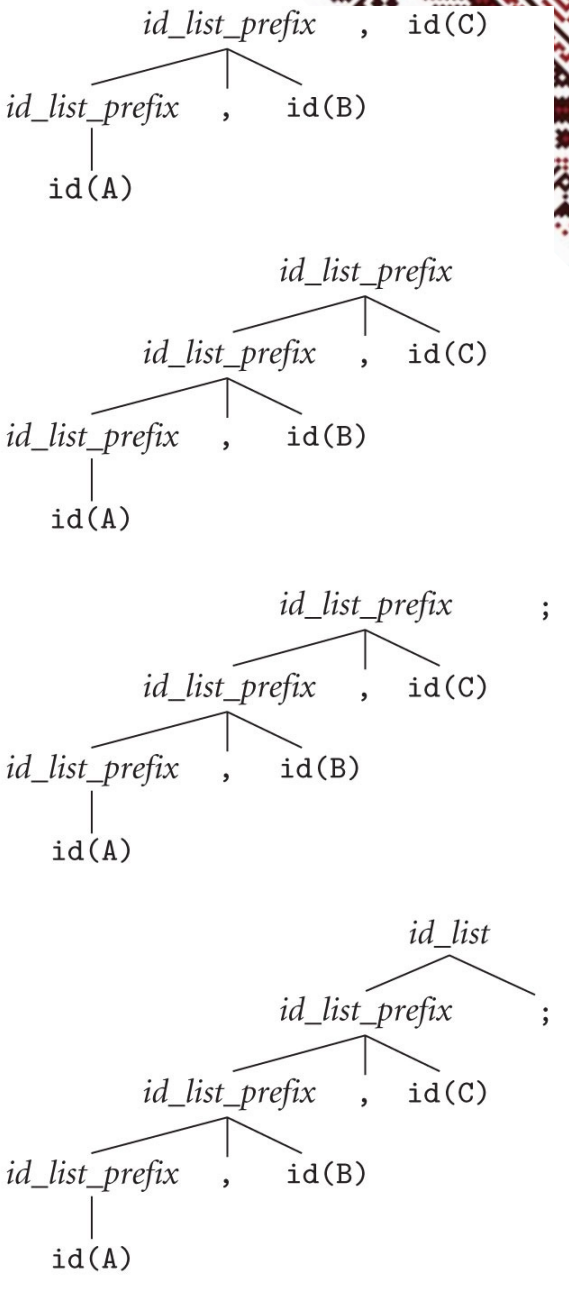


Parsing

- Bottom-up
  - better grammar
  - cannot be parsed top-down
  - Example:

    `A, B, C;`





$$\begin{aligned} \textit{id\_list} &\longrightarrow \textit{id\_list\_prefix}\ ; \\ \textit{id\_list\_prefix} &\longrightarrow \textit{id\_list\_prefix}\ ,\ \texttt{id} \\ &\longrightarrow \texttt{id} \end{aligned}$$

Parsing

- LL(k), LR(k)
  - k = no. tokens of look-ahead required to parse
  - almost all real compilers use LL(1), LR(1)
  - LR(0) - *prefix property*:
    - no valid string is a prefix of another valid string

LL Parsing

- LL(1) grammar for calculator language
  - less intuitive: operands not on the same right-hand side
  - parsing is easier ($$ added to mark the end of the program)

program → *stmt_list* `$$`
stmt_list → *stmt* *stmt_list* | ε
stmt → `id :=` *expr* | `read id` | `write` *expr*
expr → *term* *term_tail*
term_tail → *add_op* *term* *term_tail* | ε
term → *factor* *fact_tail*
fact_tail → *mult_op* *fact* *fact_tail* | ε
factor → `(` *expr* `)` | `id` | `number`
add_op → `+` | `-`
mult_op → `*` | `/`

- compare with LR grammar:

expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → `id` | `number` | `-` *factor* | (*expr*)
add_op → `+` | `-`
mult_op → `*` | `/`

- Top-down parsers
  - by hand – *recursive descent*
  - table-driven

LL Parsing

- Recursive descent parser
  - one subroutine for each nonterminal
- Example:

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

  - Continued on the next slide

```
procedure match(expected)
    if input_token = expected then consume_input_token()
    else parse_error

-- this is the start routine:
procedure program()
    case input_token of
        id, read, write, $$ :
            stmt_list()
            match($$)
        otherwise parse_error

procedure stmt_list()
    case input_token of
        id, read, write : stmt(); stmt_list()
        $$ : skip        -- epsilon production
        otherwise parse_error
```

LL Parsing

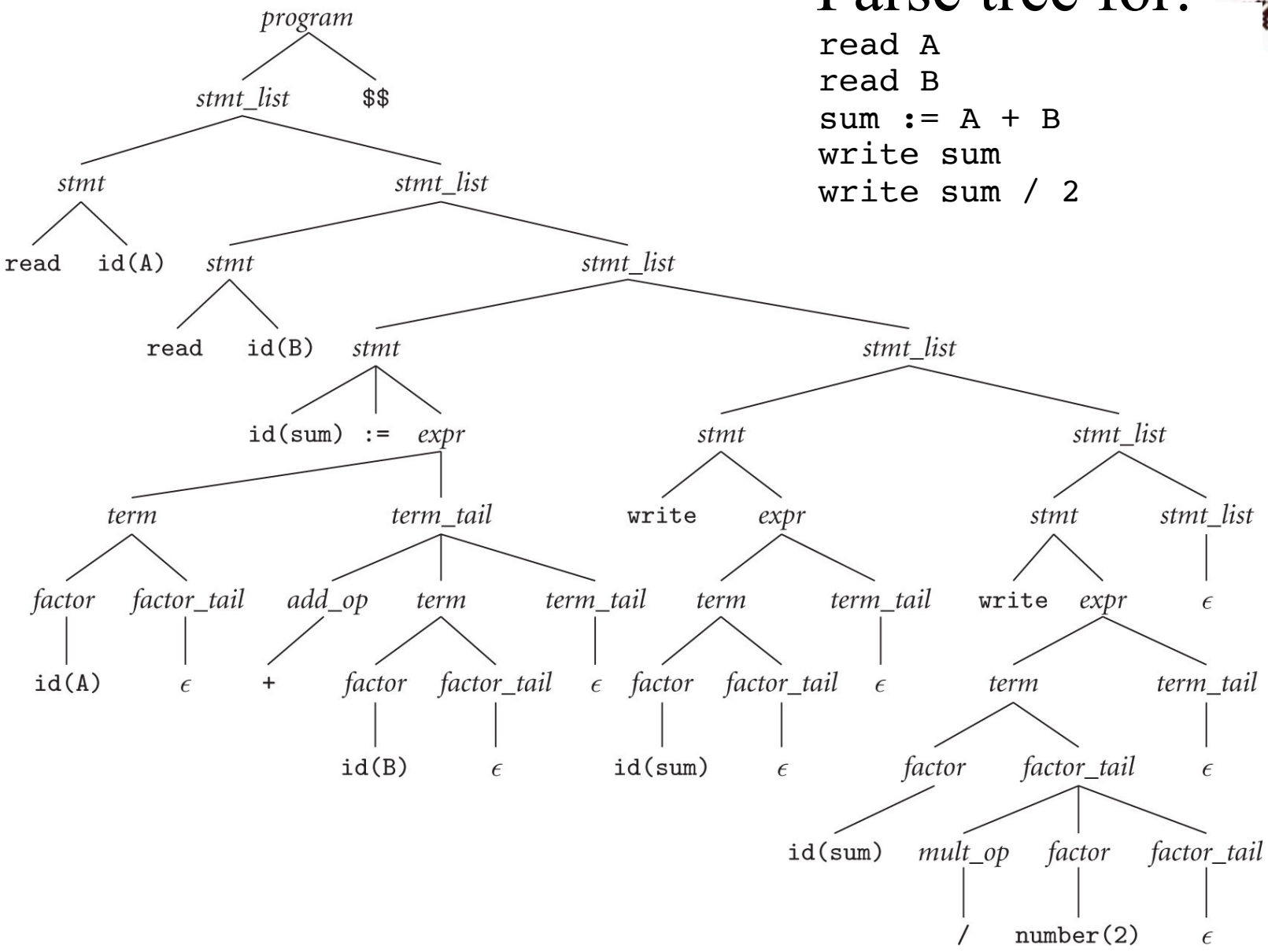
```
procedure stmt()
    case input_token of
        id : match(id); match(:=); expr()
        read : match(read); match(id)
        write : match(write); expr()
        otherwise parse_error

procedure expr()
    case input_token of
        id, number, ( : term(); term_tail()
        otherwise parse_error

procedure term_tail()
    case input_token of
        +, - : add_op(); term(); term_tail()
        ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise parse_error

procedure term()
    case input_token of
        id, number, ( : factor(); factor_tail()
        otherwise parse_error
```

```
procedure factor_tail()
    case input_token of
        *, / : mult_op(); factor(); factor_tail()
        +, -, ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise parse_error

procedure factor()
    case input_token of
        id : match(id)
        number : match(number)
        ( : match(); expr(); match())
        otherwise parse_error

procedure add_op()
    case input_token of
        + : match(+)
        - : match(-)
        otherwise parse_error

procedure mult_op()
    case input_token of
        * : match(*)
        / : match(/)
        otherwise parse_error
```

LL Parsing

- Parse tree for:

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

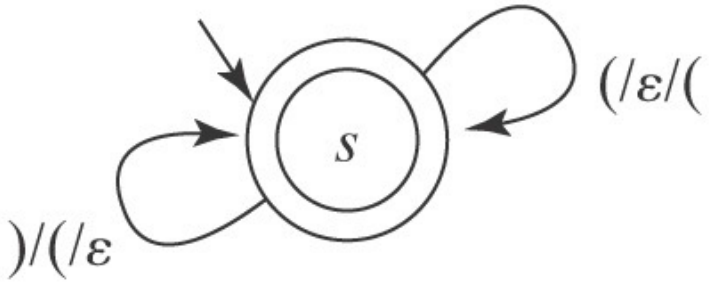


LL Parsing

- Table-driven LL parsing:
  - repeatedly look up action in 2D table based on:
    - current leftmost non-terminal and
    - current input token
  - actions:
    - (1) match a terminal
    - (2) predict a production
    - (3) announce a syntax error

LL Parsing

- Table-driven LL parsing:
  - Push-down automaton (PDA)
    - Finite automaton with a stack
    - Example: balanced parentheses: input / pop / push



  - Parsing stack: containing the expected symbols
    - initially contains the starting symbol
    - predicting a production: push the right-hand side in reverse order

LL Parsing

- Table-driven LL parsing:

```
terminal = 1 . . number_of_terminals
non_terminal = number_of_terminals + 1 . . number_of_symbols
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions

parse_tab : array [non_terminal, terminal] of record
        action : (predict, error)
        prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool

parse_stack : stack of symbol

parse_stack.push(start_symbol)
loop
        expected_sym : symbol := parse_stack.pop()
        if expected_sym ∈ terminal
                match(expected_sym)                          -- as in Figure 2.17
                if expected_sym = $$ then return             -- success!
        else
                if parse_tab[expected_sym, input_token].action = error
                        parse_error
                else
                        prediction : production := parse_tab[expected_sym, input_token].prod
                        foreach sym : symbol in reverse prod_tab[prediction]
                                parse_stack.push(sym)
```

LL Parsing

- LL(1): `parse_tab` for parsing for calculator language
- productions: 1..19
- '–' means error
- `prod_tab` (not shown) gives RHS

1	*program* → *stmt_list* `$$`
2,3	*stmt_list* → *stmt* *stmt_list* | ε
4,5,6	*stmt* → `id :=` *expr* | `read id` | `write` *expr*
7	*expr* → *term* *term_tail*
8,9	*term_tail* → *add_op* *term* *term_tail* | ε
10	*term* → *factor* *fact_tail*
11,12	*fact_tail* → *mult_op* *fact* *fact_tail* | ε
13,14,15	*factor* → `(` *expr* `)` | `id` | `number`
16,17	*add_op* → `+` | `-`
18,19	*mult_op* → `*` | `/`

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

- Example:

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

Parse stack	Input stream	Comment
program	`read A read B ...`	initial stack contents
stmt_list `$$`	`read A read B ...`	predict *program* ⟶ *stmt_list* `$$`
stmt stmt_list `$$`	`read A read B ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read A read B ...`	predict *stmt* ⟶ `read id`
`id` *stmt_list* `$$`	`A read B ...`	match `read`
stmt_list `$$`	`read B sum := ...`	match `id`
stmt stmt_list `$$`	`read B sum := ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read B sum := ...`	predict *stmt* ⟶ `read id`
`id` *stmt_list* `$$`	`B sum := ...`	match `read`
stmt_list `$$`	`sum := A + B ...`	match `id`
stmt stmt_list `$$`	`sum := A + B ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`id :=` *expr stmt_list* `$$`	`sum := A + B ...`	predict *stmt* ⟶ `id :=` *expr*
`:=` *expr stmt_list* `$$`	`:= A + B ...`	match `id`
expr stmt_list `$$`	`A + B ...`	match `:=`
term term_tail stmt_list `$$`	`A + B ...`	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`A + B ...`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`A + B ...`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`+ B write sum ...`	match `id`
term_tail stmt_list `$$`	`+ B write sum ...`	predict *factor_tail* ⟶ ϵ
add_op term term_tail stmt_list `$$`	`+ B write sum ...`	predict *term_tail* ⟶ *add_op term term_tail*
`+` *term term_tail stmt_list* `$$`	`+ B write sum ...`	predict *add_op* ⟶ `+`
term term_tail stmt_list `$$`	`B write sum ...`	match `+`
factor factor_tail term_tail stmt_list `$$`	`B write sum ...`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`B write sum ...`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`write sum ...`	match `id`

LL Parsing

- Example:

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

Parse stack	Input stream	Comment
term_tail stmt_list $$	write sum write ...	predict *factor_tail* $\longrightarrow \epsilon$
stmt_list $$	write sum write ...	predict *term_tail* $\longrightarrow \epsilon$
stmt stmt_list $$	write sum write ...	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
write *expr stmt_list* $$	write sum write ...	predict *stmt* $\longrightarrow$ write *expr*
expr stmt_list $$	sum write sum / 2	match write
term term_tail stmt_list $$	sum write sum / 2	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list $$	sum write sum / 2	predict *term* $\longrightarrow$ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	sum write sum / 2	predict *factor* $\longrightarrow$ id
factor_tail term_tail stmt_list $$	write sum / 2	match id
term_tail stmt_list $$	write sum / 2	predict *factor_tail* $\longrightarrow \epsilon$
stmt_list $$	write sum / 2	predict *term_tail* $\longrightarrow \epsilon$
stmt stmt_list $$	write sum / 2	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
write *expr stmt_list* $$	write sum / 2	predict *stmt* $\longrightarrow$ write *expr*
expr stmt_list $$	sum / 2	match write
term term_tail stmt_list $$	sum / 2	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list $$	sum / 2	predict *term* $\longrightarrow$ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	sum / 2	predict *factor* $\longrightarrow$ id
factor_tail term_tail stmt_list $$	/ 2	match id
mult_op factor factor_tail term_tail stmt_list $$	/ 2	predict *factor_tail* $\longrightarrow$ *mult_op factor factor_tail*
/ *factor factor_tail term_tail stmt_list* $$	/ 2	predict *mult_op* $\longrightarrow$ /
factor factor_tail term_tail stmt_list $$	2	match /
number *factor_tail term_tail stmt_list* $$	2	predict *factor* $\longrightarrow$ number
factor_tail term_tail stmt_list $$		match number
term_tail stmt_list $$		predict *factor_tail* $\longrightarrow \epsilon$
stmt_list $$		predict *term_tail* $\longrightarrow \epsilon$
$$		predict *stmt_list* $\longrightarrow \epsilon$

LL Parsing

- How to build the table:
  - `FIRST`(α) – tokens that can start an α
  - `FOLLOW`(A) – tokens that can come after an A

`EPS`(α) $\equiv$ if $\alpha \Longrightarrow^* \varepsilon$ then `true` else `false`

`FIRST`(α) $\equiv \{c \mid \alpha \Longrightarrow^* c\beta\}$

`FOLLOW`(A) $\equiv \{c \mid S \Longrightarrow^+ \alpha A c \beta\}$

`PREDICT`($A \rightarrow \alpha$) $\equiv$ `FIRST`(α) $\cup$ if `EPS`(α) then `FOLLOW`(A) else $\emptyset$

  - If a token belongs to the predict set of more than one production with the same left-hand side, then the grammar is not LL(1)
  - Compute: pass over the grammar until nothing changes
  - Algorithm and examples on the next slides

LL Parsing

- Constructing `EPS, FIRST, FOLLOW, PREDICT`

program $\longrightarrow$ *stmt_list* `$$`	`$$` $\in$ FOLLOW(*stmt_list*)
stmt_list $\longrightarrow$ *stmt* *stmt_list*	
stmt_list $\longrightarrow \epsilon$	EPS(*stmt_list*) = true
stmt $\longrightarrow$ `id :=` *expr*	`id` $\in$ FIRST(*stmt*)
stmt $\longrightarrow$ `read id`	`read` $\in$ FIRST(*stmt*)
stmt $\longrightarrow$ `write` *expr*	`write` $\in$ FIRST(*stmt*)
expr $\longrightarrow$ *term* *term_tail*	
term_tail $\longrightarrow$ *add_op* *term* *term_tail*	
term_tail $\longrightarrow \epsilon$	EPS(*term_tail*) = true
term $\longrightarrow$ *factor* *factor_tail*	
factor_tail $\longrightarrow$ *mult_op* *factor* *factor_tail*	
factor_tail $\longrightarrow \epsilon$	EPS(*factor_tail*) = true
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*)

LL Parsing

- Algorithm for constructing `EPS, FIRST, FOLLOW, PREDICT`

(Continued on the next slide)

```
-- EPS values and FIRST sets for all symbols:
    for all terminals c, EPS(c) := false; FIRST(c) := {c}
    for all nonterminals X, EPS(X) := if X ⟶ ϵ then true else false; FIRST(X) := ∅
    repeat
        ⟨outer⟩ for all productions X ⟶ Y1 Y2 ... Yk,
            ⟨inner⟩ for i in 1..k
                add FIRST(Yi) to FIRST(X)
                if not EPS(Yi) (yet) then continue outer loop
            EPS(X) := true
    until no further progress

-- Subroutines for strings, similar to inner loop above:

    function string_EPS(X1 X2 ... Xn)
        for i in 1..n
            if not EPS(Xi) then return false
        return true
```

LL Parsing

- Algorithm for constructing `EPS, FIRST, FOLLOW, PREDICT`

```
function string_FIRST(X_1 X_2 ... X_n)
    return_value := ∅
    for i in 1 .. n
        add FIRST(X_i) to return_value
        if not EPS(X_i) then return

-- FOLLOW sets for all symbols:
    for all symbols X, FOLLOW(X) := ∅
    repeat
        for all productions A ⟶ α B β,
            add string_FIRST(β) to FOLLOW(B)
        for all productions A ⟶ α B
                or A ⟶ α B β, where string_EPS(β) = true,
            add FOLLOW(A) to FOLLOW(B)
    until no further progress

-- PREDICT sets for all productions:
    for all productions A ⟶ α
        PREDICT(A ⟶ α) := string_FIRST(α) ∪ (if string_EPS(α) then FOLLOW(A) else ∅)
```

LL Parsing

EPS(A) is true iff
A ∈ {*stmt_list*, *term_tail*, *factor_tail*}

- Example: the sets `EPS, FIRST, FOLLOW, PREDICT`

FIRST

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

FOLLOW

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

LL Parsing

- Problems trying to make a grammar LL(1)
  - *left recursion*: $A \Longrightarrow^{+} A\alpha$
    - example – cannot be parsed top-down

$$\begin{aligned} &\textit{id\_list} \rightarrow \textit{id\_list\_prefix}\ \texttt{;} \\ &\textit{id\_list\_prefix} \rightarrow \textit{id\_list\_prefix}\ \texttt{,}\ \texttt{id} \\ &\textit{id\_list\_prefix} \rightarrow \texttt{id} \end{aligned}$$

    - solved by *left-recursion elimination*

$$\begin{aligned} &\textit{id\_list} \rightarrow \texttt{id}\ \textit{id\_list\_tail} \\ &\textit{id\_list\_tail} \rightarrow \texttt{,}\ \texttt{id}\ \textit{id\_list\_tail} \\ &\textit{id\_list\_tail} \rightarrow \texttt{;} \end{aligned}$$

  - General left-recursion elimination:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

replaced by:

$$\begin{aligned} &A \rightarrow \beta_1 B \mid \beta_2 B \mid \ldots \mid \beta_m B \\ &B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_n B \mid \varepsilon \end{aligned}$$

LL Parsing

- Problems trying to make a grammar LL(1)
  - *common prefixes*
    - example

$$\textit{stmt} \rightarrow \texttt{id} \ \texttt{:=} \ \textit{expr}$$
$$\textit{stmt} \rightarrow \texttt{id} \ \texttt{(} \ \textit{argument\_list} \ \texttt{)}$$

    - solved by *left-factoring*

$$\textit{stmt} \rightarrow \texttt{id} \ \textit{stmt\_list\_tail}$$
$$\textit{stmt\_list\_tail} \rightarrow \texttt{:=} \ \textit{expr}$$
$$\textit{stmt\_list\_tail} \rightarrow \texttt{(} \ \textit{argument\_list} \ \texttt{)}$$

- Note: Eliminating left recursion and common prefixes does NOT make a grammar LL; there are infinitely many non-LL languages, and the automatic transformations work on them just fine

LL Parsing

- Problems trying to make a grammar LL(1)
  - the *dangling else* problem
  - prevents grammars from being LL(*k*) for any *k*
  - Example: ambiguous (Pascal)

stmt → `if` *cond then_clause else_clause* | *other_stmt*
then_clause → `then` *stmt*
else_clause → `else` *stmt* | ε

`if` C_1 `then if` C_2 `then` S_1 `else` S_2

LL Parsing

- Dangling else problem
  - Solution: unambiguous grammar
  - can be parsed bottom-up but not top-down
    - there is no top-down grammar

stmt → *balanced_stmt* | *unbalanced_stmt*

balanced_stmt → `if` *cond* `then` *balanced_stmt* `else` *balanced_stmt*
| *other_stmt*

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

LL Parsing

- Dangling else problem
  - Another solution - *end-markers*

stmt → `IF` *cond then_clause else_clause* `END` | *other_stmt*
then_clause → `THEN` *stmt_list*
else_clause → `ELSE` *stmt_list* | ε

- Modula-2, for example, one says:

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

- Ada: `end if`
- other languages: `fi`

LL Parsing

- Problem with end markers: they tend to bunch up

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

  - To avoid this: `elsif`

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

Programming Language Syntax

- LR parsing -

Chapter 2 , Section 2.3

LR Parsing

- LR parsers
  - maintain a forest of subtrees of the parse tree
  - join trees together when recognizing a RHS
  - keeps the roots of subtrees in a stack
  - *shift*: tokens from scanner into the stack
  - *reduce*: when recognizing a RHS, pop it, push LHS
  - discovers a right-most derivation in reverse

Stack contents (roots of partial trees)	Remaining input
ε	`A, B, C;`
`id (A)`	`, B, C;`
`id (A),`	`B, C;`
`id (A), id (B)`	`, C;`
`id (A), id (B),`	`C;`
`id (A), id (B), id (C)`	`;`
`id (A), id (B), id (C) ;`	
`id (A), id (B),` *id_list_tail*	
`id (A),` *id_list_tail*	
`id (A)` *id_list_tail*	
id_list	

LR Parsing

- Example: LR(1) grammar for calculator language

1. *program* → *stmt_list* `$$`
2. *stmt_list* → *stmt_list stmt*
3. *stmt_list* → *stmt*
4. *stmt* → `id :=` *expr*
5. *stmt* → `read id`
6. *stmt* → `write` *expr*
7. *expr* → *term*
8. *expr* → *expr add_op term*
9. *term* → *factor*
10. *term* → *term mult_op factor*
11. *factor* → `(` *expr* `)`
12. *factor* → `id`
13. *factor* → `number`
14. *add_op* → `+`
15. *add_op* → `-`
16. *mult_op* → `*`
17. *mult_op* → `/`

- Compare with previous LL(1)
  - left recursive prod. is better
  - keeps operands together

program → *stmt list* `$$`

stmt_list → *stmt stmt_list* | ε

stmt → `id :=` *expr* | `read` *id* | `write` *expr*

expr → *term term_tail*

term_tail → *add op term term_tail* | ε

term → *factor fact_tail*

fact_tail → *mult_op fact fact_tail* | ε

factor → `(` *expr* `)` | `id` | `number`

add_op → `+` | `-`

mult_op → `*` | `/`

LR Parsing

- LR parser
  - recognizes right-hand sides of productions
    - keep track of productions we might be in the middle of
    - and where: represent the location in an RHS by a '•'
  - Example:

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

LR Parsing

- start with:

  program → • *stmt_list* `$$` – this is called an **LR-item**

- '•' in front of *stmt_list* means we may be about to see the yield of *stmt_list,* that is, we could also be at the beginning of a production with *stmt_list* on LHS:

  stmt_list → • *stmt_list stmt*
  stmt_list → • *stmt*

- similarly, we need to include also:

  stmt → • `id :=` *expr*
  stmt → • `read id`
  stmt → • `write` *expr*

- Only terminals follow, so we stop

LR Parsing

- the state we have obtained is:

program → • *stmt_list* `$$`	(the basis)	(state 0)
stmt_list → • *stmt_list stmt*	(closure …	
stmt_list → • *stmt*	…	
stmt → • `id :=` *expr*	…	
stmt → • `read id`	…	
stmt → • `write` *expr*	…	)

- next token: `read` - the next state is:

stmt → `read` • `id` (empty closure) (state 1)

- next token: `A` - the next state is:

stmt → `read id` • (state 1')

- '•' at the end means we can reduce
  - what is the new state?

LR Parsing

- replace `read id` with *stmt*

  stmt_list → • *stmt* becomes

  stmt_list → *stmt* • (state 0')

- we reduce again: replace *stmt* with *stmt_list*
- this means shifting a *stmt_list* in state 0:

  program → *stmt_list* • `$$` (basis … (state 2)

  stmt_list → *stmt_list* • *stmt* ...)

  stmt → • `id :=` *expr* (closure …

  stmt → • `read id` …

  stmt → • `write` *expr* …)

- Complete states on next slides

LR Parsing

	State	Transitions
0.	*program* ⟶ • *stmt_list* `$$`	on *stmt_list* shift and goto 2
	stmt_list ⟶ • *stmt_list stmt*	
	stmt_list ⟶ • *stmt*	on *stmt* shift and reduce (pop 1 state, push *stmt_list* on input)
	stmt ⟶ • `id` := *expr*	on `id` shift and goto 3
	stmt ⟶ • `read id`	on `read` shift and goto 1
	stmt ⟶ • `write` *expr*	on `write` shift and goto 4
1.	*stmt* ⟶ `read` • `id`	on `id` shift and reduce (pop 2 states, push *stmt* on input)
2.	*program* ⟶ *stmt_list* • `$$`	on `$$` shift and reduce (pop 2 states, push *program* on input)
	stmt_list ⟶ *stmt_list* • *stmt*	on *stmt* shift and reduce (pop 2 states, push *stmt_list* on input)
	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

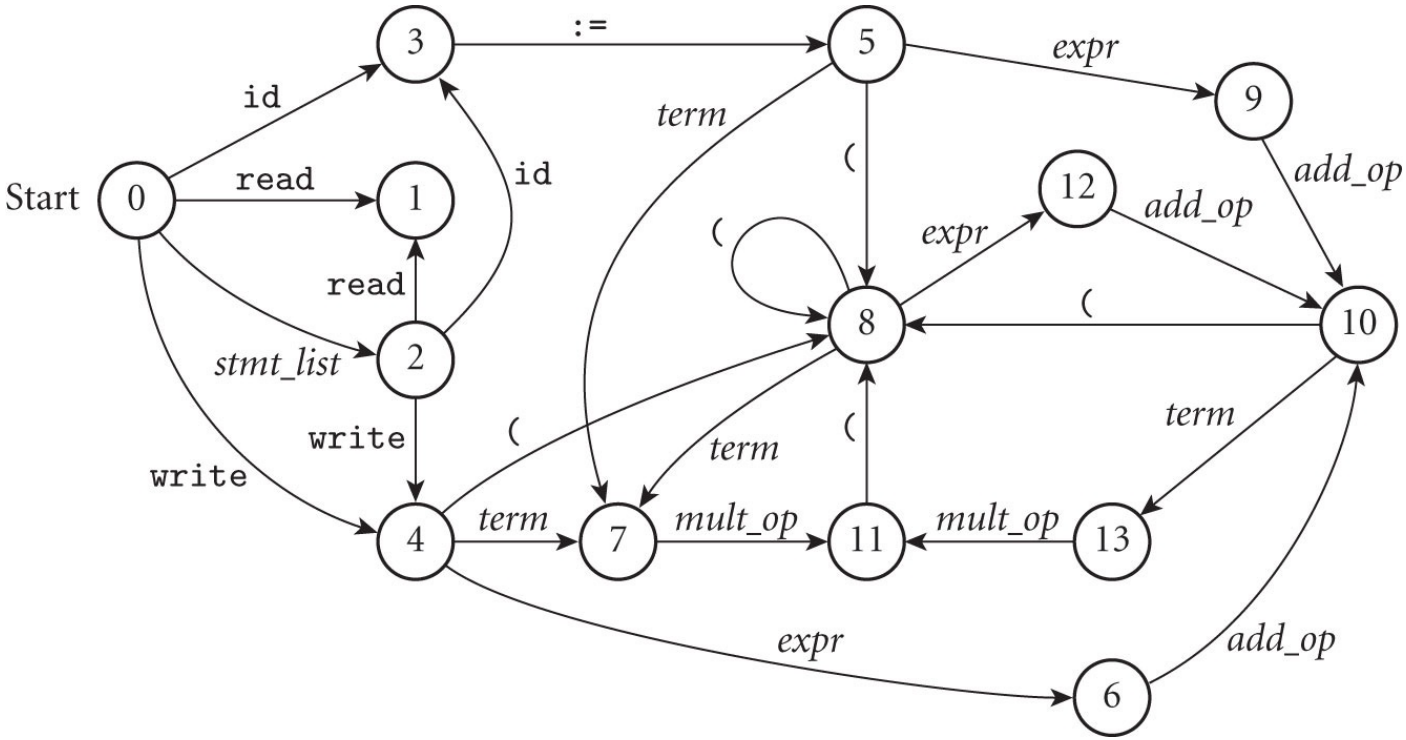
	State	Transitions
4.	*stmt* ⟶ `write` • *expr*	on *expr* shift and goto 6
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift 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)
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, 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)
6.	*stmt* ⟶ `write` *expr* •	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce (pop 2 states, push *stmt* 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)

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

	State	Transitions
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 (pop 3 states, push *expr* on input)
	term ⟶ *term* • *mult_op factor*	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)

LR Parsing

- LL(1) parser: decides using nonterminal + token
- LR(1) parser: decides using state + token
  - CFSM: Characteristic Finite State Machine
  - Almost always table-driven



LR Parsing

- Parse table `parse_tab`
  - shift (s) followed by state
  - reduce (r), shift + reduce (b) followed by production

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

LR Parsing

- Algorithm
- uses the `parse_tab` (previous slide) and `prod_tab` (not shown)
- example after algorithm for:

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

```
state = 1 . . number_of_states
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions
action_rec = record
        action : (shift, reduce, shift_reduce, error)
        new_state : state
        prod : production

parse_tab : array [symbol, state] of action_rec
prod_tab : array [production] of record
        lhs : symbol
        rhs_len : integer
-- these two tables are created by a parser generator tool

parse_stack : stack of record
        sym : symbol
        st : state
```

LR Parsing

```
parse_stack.push(⟨null, start_state⟩)
cur_sym : symbol := scan()                          -- get new token from scanner
loop
    cur_state : state := parse_stack.top().st       -- peek at state at top of stack
    if cur_state = start_state and cur_sym = start_symbol
        return                                      -- success!
    ar : action_rec := parse_tab[cur_state, cur_sym]
    case ar.action
        shift:
            parse_stack.push(⟨cur_sym, ar.new_state⟩)
            cur_sym := scan()                       -- get new token from scanner
        reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len)
        shift_reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len−1)
        error:
            parse_error
```

LR Parsing

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*

LR Parsing

Parse stack	Input stream	Comment
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]		

LR Parsing

- *Shift/reduce conflict*
  - two items in a state:
    - one with '•' in front of terminal (shift)
    - one with '•' at the end (reduce)
  - SLR (simple LR)
    - conflict can be resolved using `FIRST` and `FOLLOW`
  - Example: state 6
    - *stmt* → `write` *expr* •
    - *expr* → *expr* • *add_op term*
    - `FIRST`(*add_op*) ∩ `FOLLOW`(*stmt*) = ∅

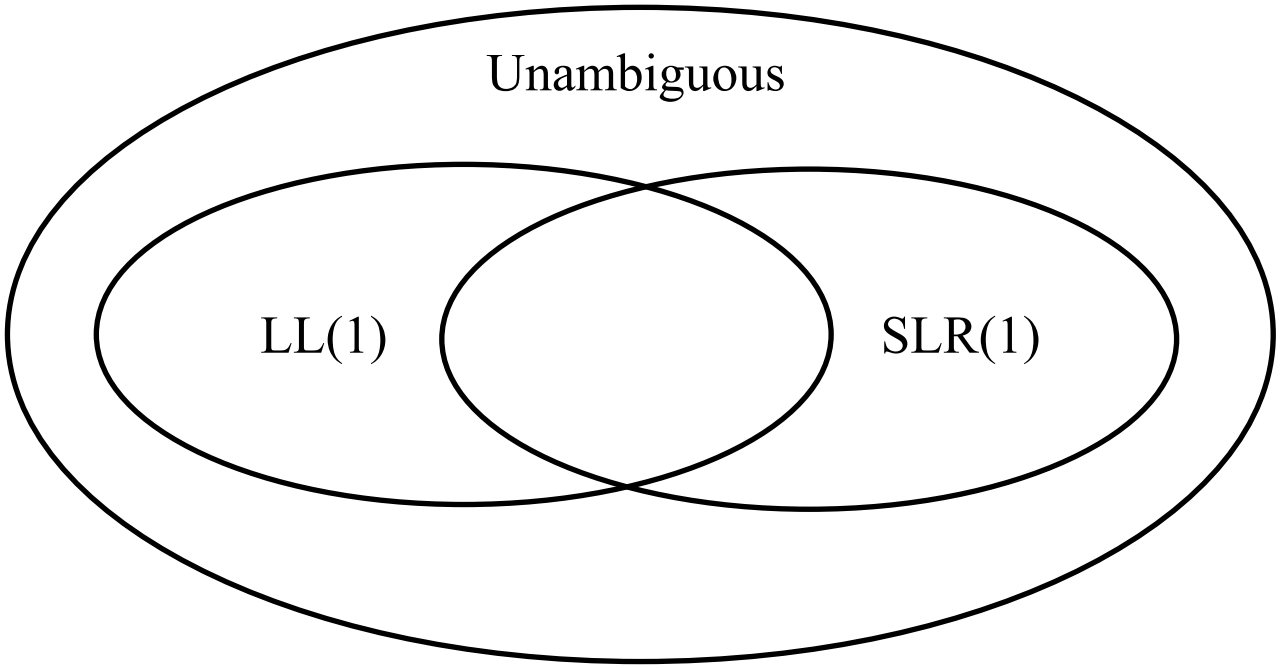
LL(1) vs SLR(1)

- LL(1)
  - For any productions $A \rightarrow u \mid v$:
    - $\texttt{FIRST}(u) \cap \texttt{FIRST}(v) = \emptyset$
    - at most one of u and v can derive the empty string ε
    - if $v \Rightarrow^* \varepsilon$, then $\texttt{FIRST}(u) \cap \texttt{FOLLOW}(A) = \emptyset$
- SLR(1)
  - No shift/reduce conflict: cannot have in the same state:

    $A \rightarrow u \bullet xv$, $B \rightarrow w \bullet$, with $x \in \texttt{FOLLOW}(B)$
  - No reduce/reduce conflict: cannot have in the same state:

    $A \rightarrow u \bullet$, $B \rightarrow v \bullet$, with $\texttt{FOLLOW}(A) \cap \texttt{FOLLOW}(B) \neq \emptyset$

Unambiguous vs LL(1) vs SLR(1)



Names, Scopes and Bindings

Chapter 3

Name, Scope, and Binding

- *Ease of programming* – main driving force behind the design of modern languages
- Core issues in language design:
  - names – abstraction
  - control flow
  - types, composite types
  - subroutines – control abstraction
  - classes – data abstraction
- High level programming – more abstract
  - Farther from hardware
- *Abstraction* – complexity becomes manageable
  - This is true in general

Name, Scope, and Binding

- *Name*: a character string representing something else
  - Abstraction
  - Easy for humans to understand
  - Much better than addresses
- *Binding*: association of two things
  - Example: between a name and the thing it names
- *Scope* of a binding: the part of the program (textually) in which the binding is active
- *Binding Time*: the point at which a binding is created

Binding

- Static vs. Dynamic
  - *Static*: bound before run time
  - *Dynamic*: bound at run time
- Trade-off:
  - *Early* binding times: greater *efficiency*
  - *Late* binding times: greater *flexibility*
- Compiled vs. Interpreted languages
  - *Compiled* languages tend to have *early* binding times
  - *Interpreted* languages tend to have *late* binding times

Language	Binding Time	Advantage
Compiled	Early (static)	Efficiency
Interpreted	Late (dynamic)	Flexibility

Lifetime and Storage Management

- *Lifetime* of name-to-binding:
  - from creation to destruction
  - Object's lifetime ≥ binding's lifetime
    - Example: C++ variable passed by reference (`&`)
  - Object's lifetime < binding's lifetime – *dangling reference*
    - Example: C++ object
      - created with `new`
      - passed by reference to subroutine with `&`
      - deallocated with `delete`

- *Scope of a binding*:
  - the textual region of the program in which the binding is *active*

Lifetime and Storage Management

- *Storage Allocation* mechanisms:

- Static
  - absolute address, retained throughout the program
- Stack
  - last-in, first-out order; for subroutines calls and returns
- Heap
  - allocated and deallocated at arbitrary times

Lifetime and Storage Management

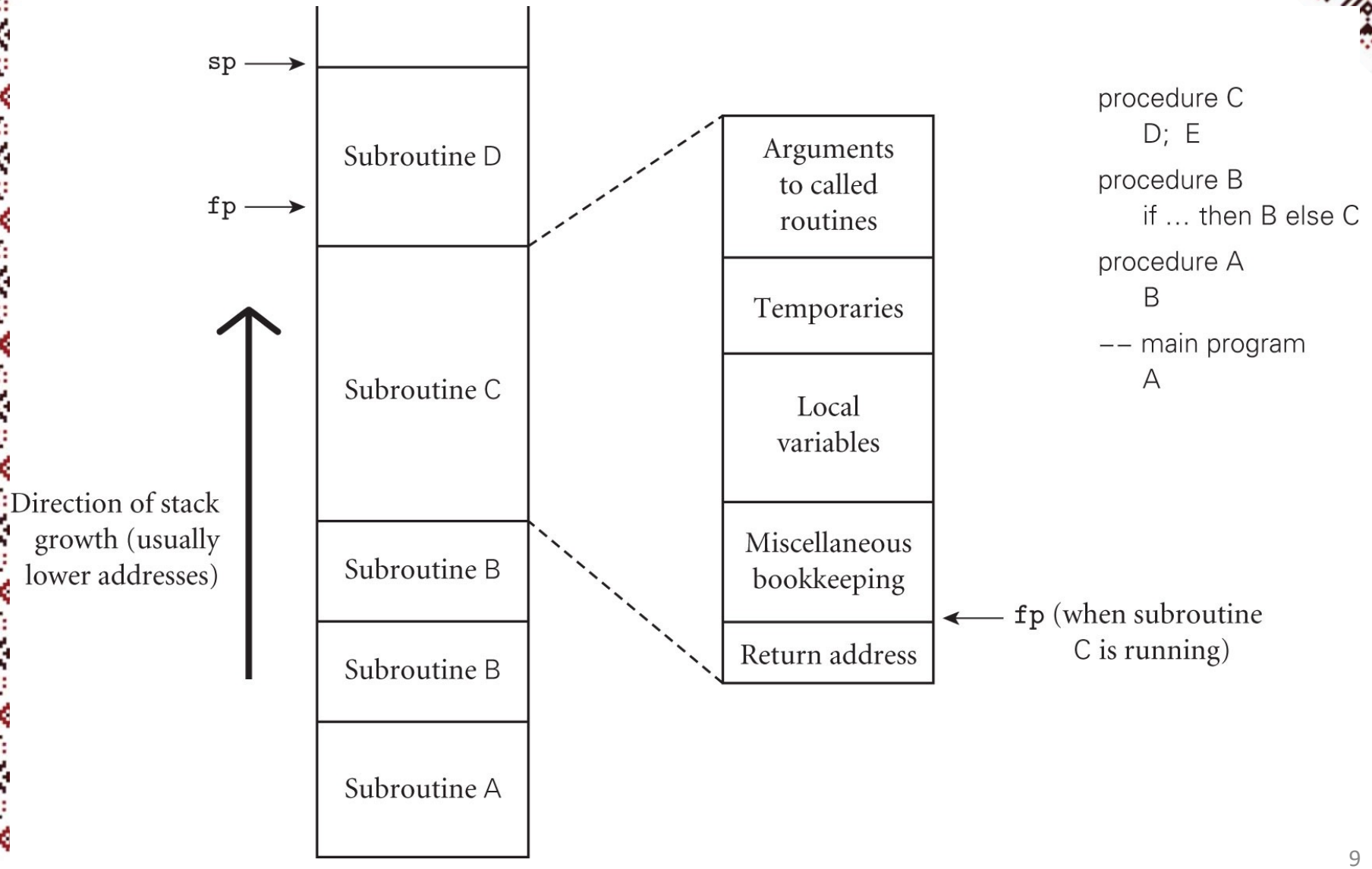
- *Static allocation*:
  - global variables
  - code instructions
  - explicit constants (including strings, sets, etc.)
    - `A = B / 14.7`
    - `printf("hello, world\n")`
    - small constants may be stored in the instructions
  - C++ `static` variables (or Algol `own`)

  - Statically allocated objects that do not change value are allocated in read-only memory
    - constants, instructions

Lifetime and Storage Management

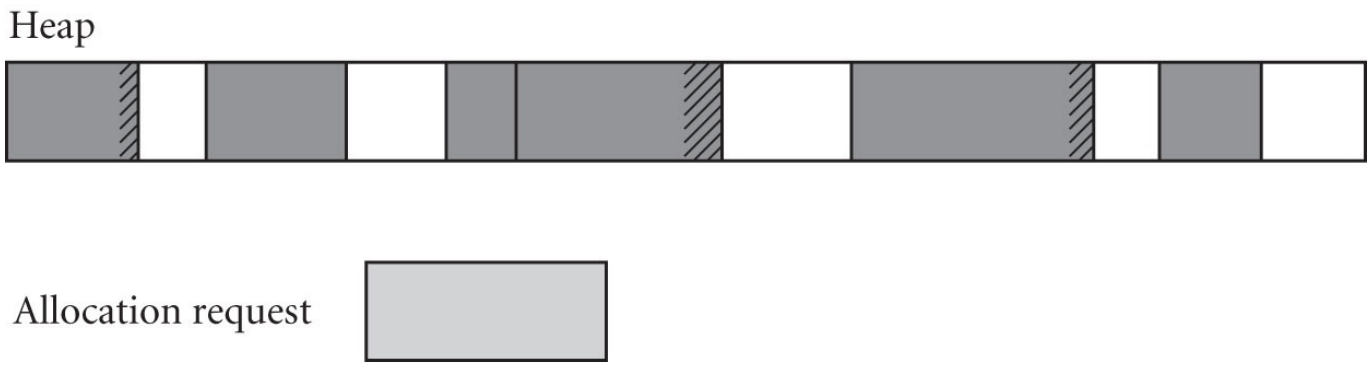
- *Stack-based allocation*:
  - parameters, local variables, temporaries
  - allocate space for recursive routines
  - reuse space
- *Frame* (*activation record*) for each subroutine call:
  - position in stack: *frame pointer*
  - arguments and returns
  - local variables, temporaries:
    - *fixed offset* from the frame pointer at compile time
  - return address
  - *dynamic link*: reference to (stack frame of) caller
  - *static link:* reference to (stack frame of) routine inside which it was declared

Lifetime and Storage Management



Lifetime and Storage Management

- *Heap allocation*
- (different from “heap” data structure for priority queues)
- dynamic allocation: lists, sets, strings (size can change)
- single linked list of free blocks
- fragmentation: internal, external



Lifetime and Storage Management

- Heap allocation
- allocation algorithms
  - first fit, best fit– $O(n)$ time
  - pool allocation – $O(1)$ time
    - separate free list of blocks for different sizes
    - *buddy system*: blocks of size 2^k
    - *Fibonacci heap*: blocks of size Fibonacci numbers
- defragmentation

Lifetime and Storage Management

- Heap maintenance
- Explicit deallocation
  - C, C++
  - simple to implement
  - efficient
  - object deallocated too soon – *dangling reference*
  - object not deallocated at the end of lifetime – *memory leak*
  - deallocation errors are very difficult to find
- Implicit deallocation: *garbage collection*
  - functional, scripting languages
  - C#, Java, Python
  - avoid memory leaks (difficult to find otherwise)
  - recent algorithms more efficient
  - the trend is towards automatic collection

Scope Rules

- *Scope of a binding*:
  - textual region of the program in which binding is active
- Subroutine entry – usually creates a new scope:
  - create bindings for new local variables
  - deactivate bindings for redeclared global variables
  - make references to variables
- Subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for deactivated global variables
- *Scope*: maximal program section in which no bindings change
  - *block*: module, class, subroutine
  - C: { … }
  - *Elaboration time*: when control first enters a scope

Scope Rules

- *Referencing environment*
  - the set of active bindings; determined by:
  - *Scope rules* (static or dynamic)
  - *Binding rules* (deep or shallow)

- *Static Scoping* (*Lexical Scoping*)
  - almost all languages employ static scoping
  - determined by examining the text of the program
  - at compile time
  - *closest nested rule*
    - identifiers known in the scope where they are declared and in each enclosed scope, unless re-declared
    - examine local scope and statically enclosing scopes until a binding is found

Scope Rules

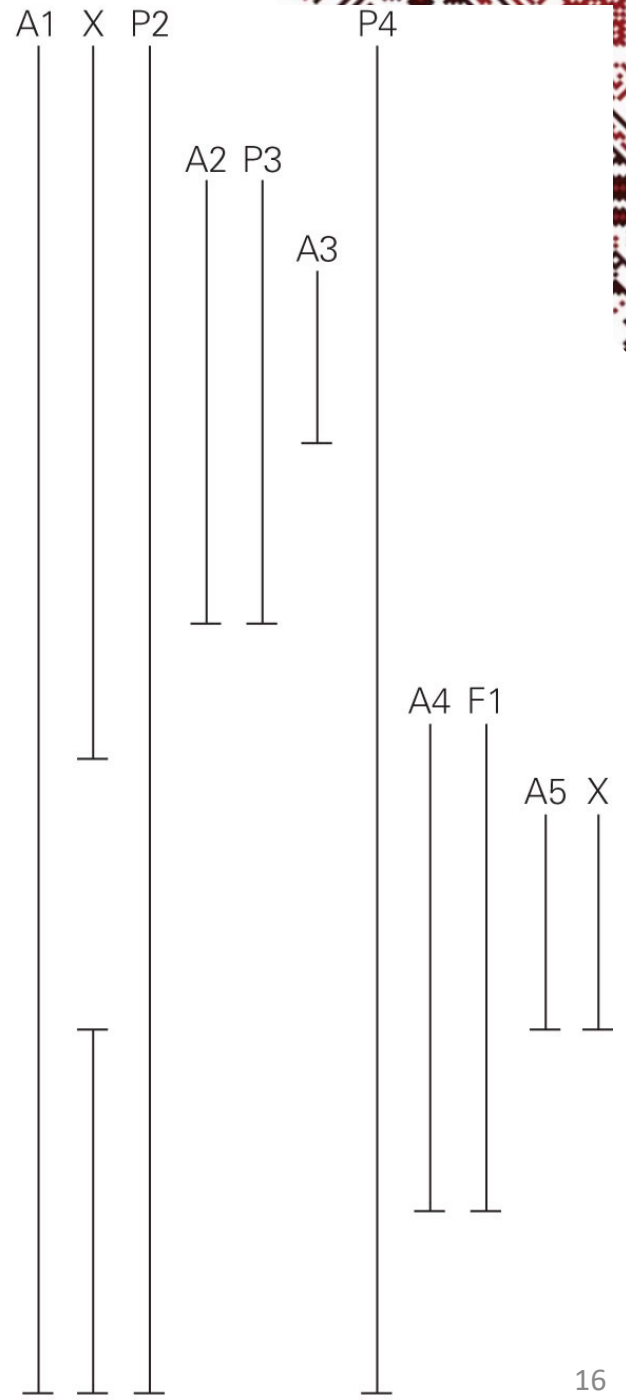
- Subroutines
  - bindings created are destroyed at subroutine exit
    - exception: `static` (C), `own` (Algol)
  - nested subroutines: *closest nested scope*
    - Python, Scheme, Ada, Common Lisp
    - not in: C, C++, Java
  - access to non-locals: *scope resolution operator*
    - C++ (global): `::X`
    - Ada: `MyProc.X`
  - built-in objects
    - outermost scope
    - outside global

Scope Rules

- Example:

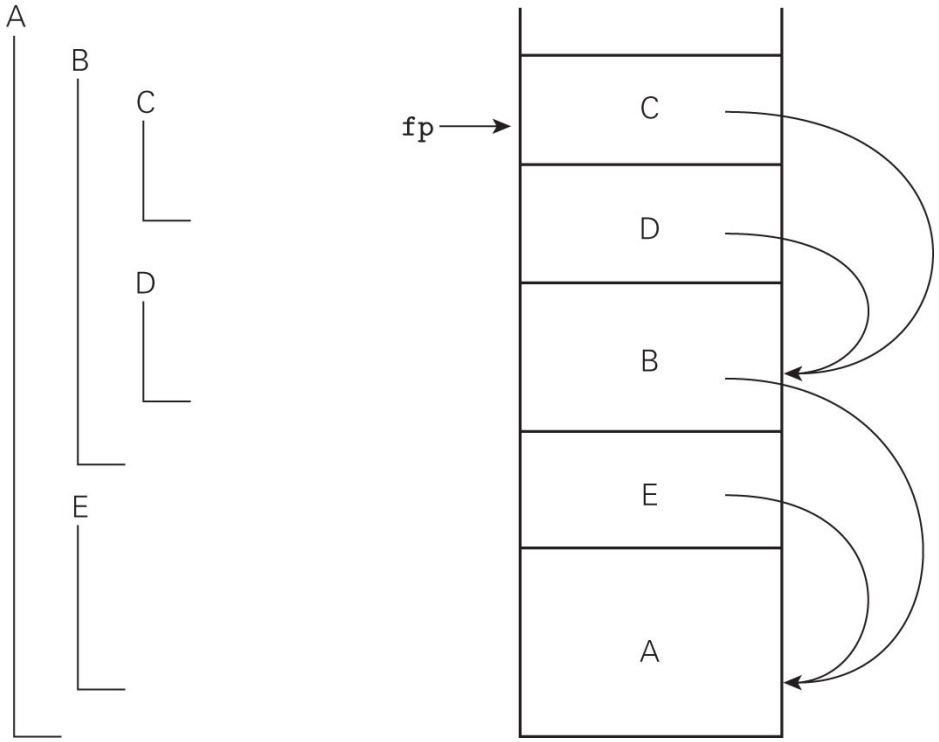
Nested subroutines

```
procedure P1(A1)
    var X              -- local to P1
    ...
    procedure P2(A2)
        ...
        procedure P3(A3)
            ...
        begin
            ...        -- body of P3
        end
        ...
    begin
        ...            -- body of P2
    end
    ...
    procedure P4(A4)
        ...
        function F1(A5)
            var X      -- local to F1
            ...
        begin
            ...        -- body of F1
        end
        ...
    begin
        ...            -- body of P4
    end
    ...
begin
    ...                -- body of P1
end
```



Scope Rules

- Access to non-locals: *static links*
  - each frame points to the frame of the routine inside which it was declared
  - access a variable in a scope *k* levels out by following *k* static links and then using the known offset within the frame



Scope Rules

- Declaration order
  - object *x* declared inside block *B*
  - the scope of *x* may be:
    - the entire block *B* or
    - only the part of *B* after *x*'s declaration

Scope Rules

- Declaration order

- Example: C++

```
int n = 1;
void f(void){
    int m = n;  // global n
    int n = 2;  // local n
}
```

- Example: Python – no declarations

```
n = 1
def f():
   m = n # error
         # add “global n” to use the global n
   n = 2
```

Scope Rules

- Declaration order

- Example: Scheme

```
(let ((A 1))
  (let ((A 2)
        (B A))
    B))          ; return 1

(let ((A 1))
  (letrec ((A 2)
           (B A))
    B))          ; return 2
```

Scope Rules

- *Dynamic Scoping*
- binding depends on flow at run time
  - use the most recent, active binding made at run time
- Easy to implement – just a stack with names
- Harder to understand
  - not used any more
  - why learn? – history

Scope Rules

- Example: Dynamic Scoping

```
n: integer                  – global
procedure first()
    n := 1
procedure second()
    n : integer             – local
    first()
n := 2
if read_integer() > 0
    second()
else
    first()
write_integer(n)
```

- Static scoping: prints 1
- Dynamic scoping: prints 2 for positive input, 1 for negative

Scope Rules

- Example: Dynamic scoping problem
  - `scaled_score` uses the wrong `max_score`

```
max_score : integer      -- maximum possible score
function scaled_score(raw_score : integer) : real
   return raw_score / max_score * 100
...
procedure foo( )
   max_score : real := 0 -- highest % seen so far
   ...
   foreach student in class
      student.percent := scaled_score(student.points)
      if student.percent > max_score
         max_score := student.percent
```

Binding of Referencing Environments

- Referencing environment: the set of active bindings
  - static: lexical nesting
  - dynamic: order of declarations at run time
- Reference to subroutine: when are the scope rules applied?
  - *Shallow binding*: when routine is called
    - default in dynamic scoping
  - *Deep binding*: when reference is created
    - default in static scoping
- Example (next slides)

Binding of Referencing Environments

- `print_routine`
  - shallow binding
  - to pick `line_length`
- `older_than_threshold`
  - deep binding
  - otherwise, if `print_selected_records` has a variable `threshold`, it will hide the one in the main program

```
type person = record
        ...
        age : integer
        ...
threshold : integer
people : database

function older_than_threshold(p : person) : boolean
        return p.age ≥ threshold

procedure print_person(p : person)
        -- Call appropriate I/O routines to print record on standard output.
        -- Make use of nonlocal variable line_length to format data in columns.
        ...
```

Binding of Referencing Environments

```
procedure print_selected_records(db : database;
          predicate, print_routine : procedure)
     line_length : integer

     if device_type(stdout) = terminal
          line_length := 80
     else       -- Standard output is a file or printer.
          line_length := 132
     foreach record r in db
          -- Iterating over these may actually be
          -- a lot more complicated than a 'for' loop.
          if predicate(r)
               print_routine(r)

-- main program
...
threshold := 35
print_selected_records(people, older_than_threshold, print_person)
```

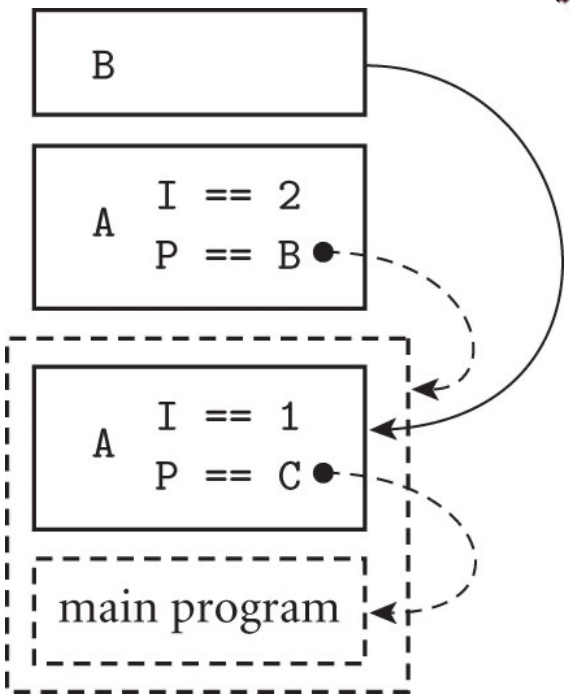
Binding of Referencing Environments

- Deep binding implementation: *subroutine closure*
  - explicit representation of a referencing environment
    (the one in which the subroutine would execute if called now)
  - reference to subroutine
- Why binding time matters with static scoping?
  - the running program may have two instances of an object
  - only for objects that are neither local nor global
  - Examples when it does not matter:
    - subroutines cannot be nested: C
    - only outermost subroutines can be passed as parameters: Modula-2
    - subroutines cannot be passed as parameters: PL/I, Ada 83

Binding of Referencing Environments

- Example: Deep binding in Python

```
def A(I, P):
    def B():
        print(I)
    # body of A:
    if I > 1:
        P()
    else:
        A(2, B)
def C():
    pass # do nothing
A(1, C)  # main program; output 1
```



- referencing environment captured in closures: dashed boxes, arrows
- when `B` is called via `P`, two instances of `I` exist
- the closure for `P` was created in the initial invocation of `A`
- `B`'s static link (solid arrow) points to the frame of the earlier invocation

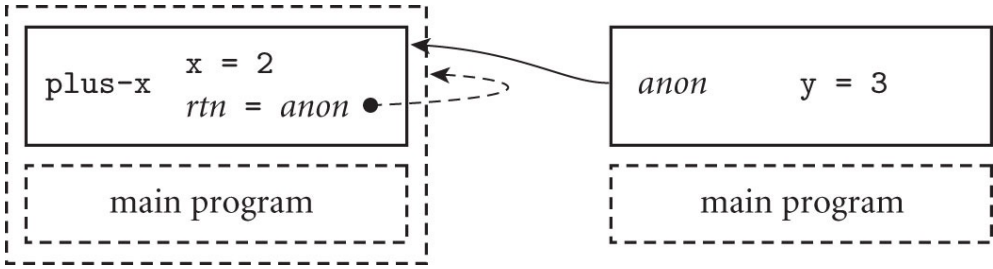
Binding of Referencing Environments

- *First-class* values
  - can be passed as a parameter
  - can be returned from subroutine
  - can be assigned into a variable
- *Second-class* values
  - can only be passed as a parameter
- *Third-class*: none
  - Other authors may have different definitions: no second-class; first-class may require anonymous function definition (lambda expressions)
- Subroutines:
  - first-class: functional and scripting languages, C#
    - C, C++: pointers to functions are first-class
  - second-class: most imperative languages
  - third class: Ada83

Binding of Referencing Environments

- First-class subroutines: additional complexity
  - a reference to a subroutine may outlive the execution of the scope in which that subroutine was declared
  - Example: Scheme

```
(define plus-x
  (lambda (x)
   (lambda (y)(+ x y))))
(let ((f (plus-x 2)))
  (f 3))        ; return 5
```



- `plus-x` returns an unnamed function (3rd line), which uses the parameter `x` of `plus-x`
- when `f` is called in 5th line, its referencing environment includes the `x` in `plus-x`, even though `plus-x` has already returned
- `x` must be still available – *unlimited extent* – allocate on heap (C#)

Binding of Referencing Environments

- *Lambda expressions*
  - come from lambda calculus: anonymous functions
  - Example: Scheme

```
((lambda (i j) (> i j) i j) 5 8)  ;return 8
```

  - Example: C#: `delegate` or `=>`

```
(int i, int j) => i > j ? i : j
```

Binding of Referencing Environments

- First-class subroutines
  - are increasingly popular; made their way into C++, Java
  - Problem: C++, Java do not support unlimited extent
- Example: C++

```
for_each(V.begin(), V.end(),
    [](int e){ if (e < 50) cout << e << " "; }
);
```

Binding of Referencing Environments

- *Lambda functions* in Python
  - Example

```
ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100’]

# Lexicographic sort
print(sorted(ids))
  => ['id1', 'id100', 'id2', 'id22', 'id3', 'id30’]

# Integer sort
sorted_ids = sorted(ids, key=lambda x: int(x[2:]))
print(sorted_ids)
  => ['id1', 'id2', 'id3', 'id22', 'id30', 'id100’]
```

Binding of Referencing Environments

- Lambda functions in Python
  - Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
print(mydoubler(11))
  => 22

mytripler = myfunc(3)
print(mytripler(11))
  => 33
```

Semantic Analysis

Chapter 4

Role of Semantic Analysis

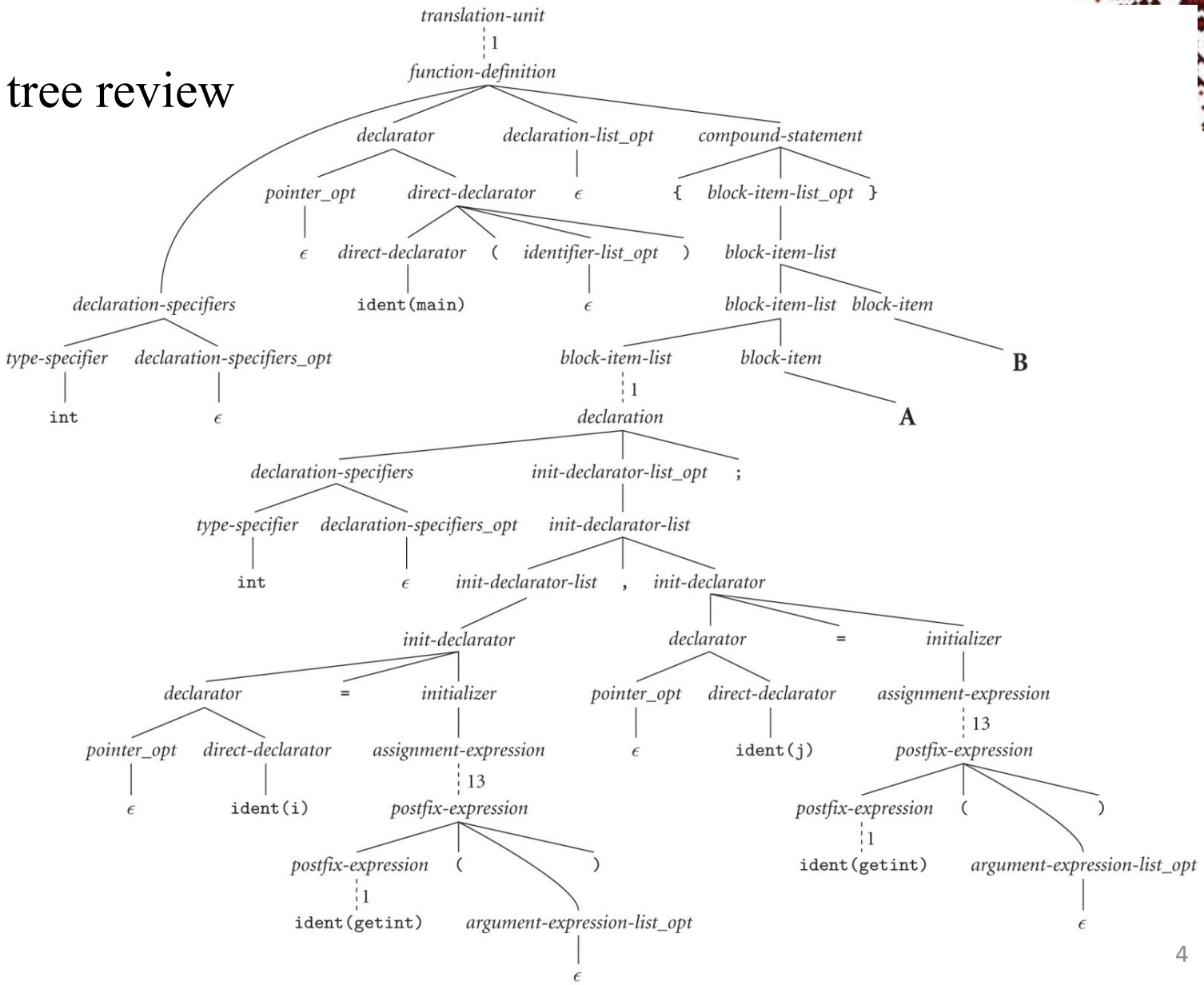
- Syntax
  - “form” of a program
  - “easy”: check membership for CFG
  - linear time
- Semantics
  - meaning of a program
  - *impossible*: program correctness **undecidable**!
  - we do whatever we can

Role of Semantic Analysis

- *Static* semantics – compile time
  - enforces static semantic rules at compile time
  - generates code to enforce dynamic semantic rules
  - constructs a syntax tree
  - information gathered for the code generator
- *Dynamic* semantics – run time
  - division by zero
  - index out of bounds
- Semantic analysis (and intermediate code generation) – described in terms of annotation (decoration) of parse tree or syntax tree
  - annotations are attributes – *attribute grammars*

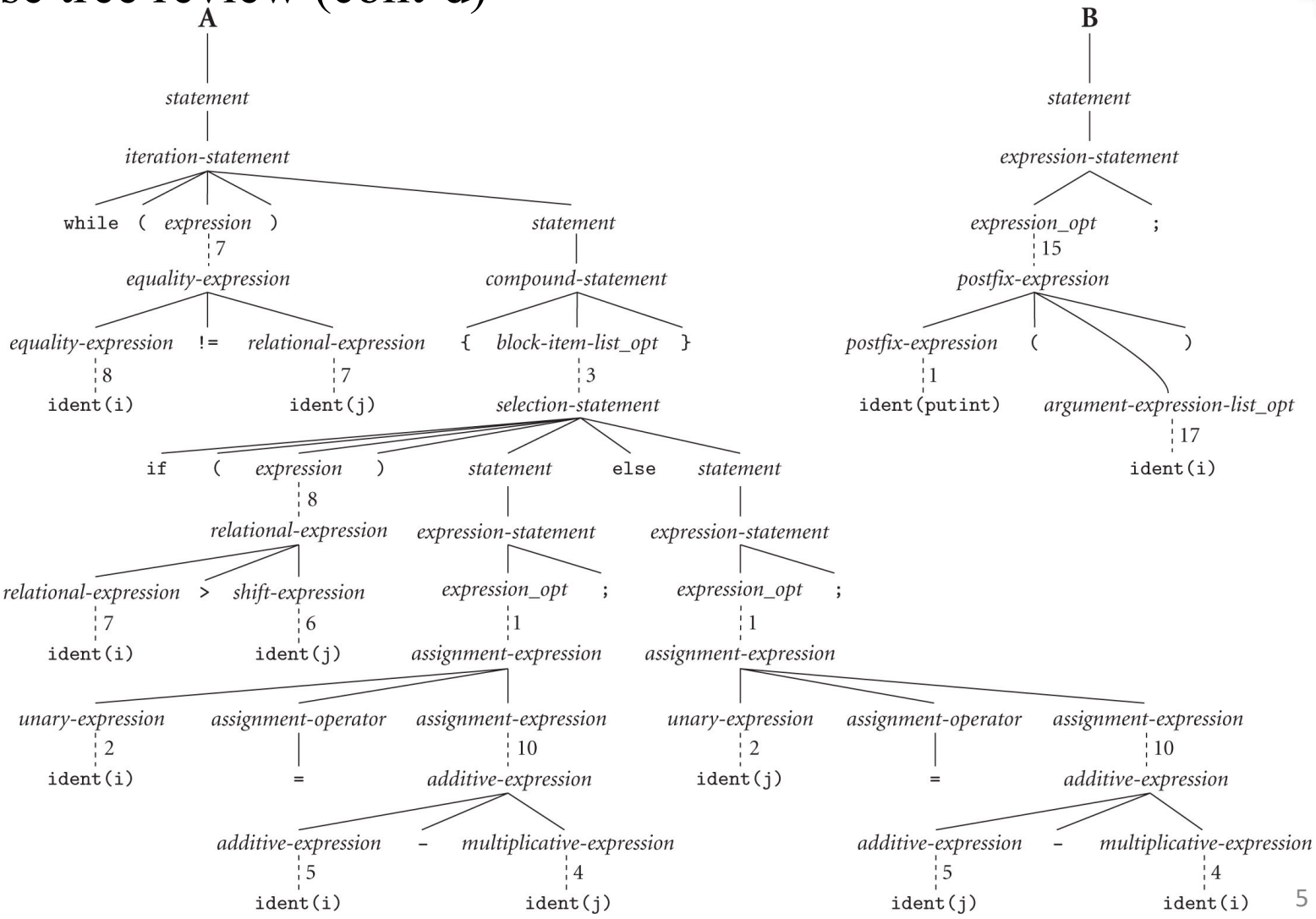
Role of Semantic Analysis

- Parse tree review



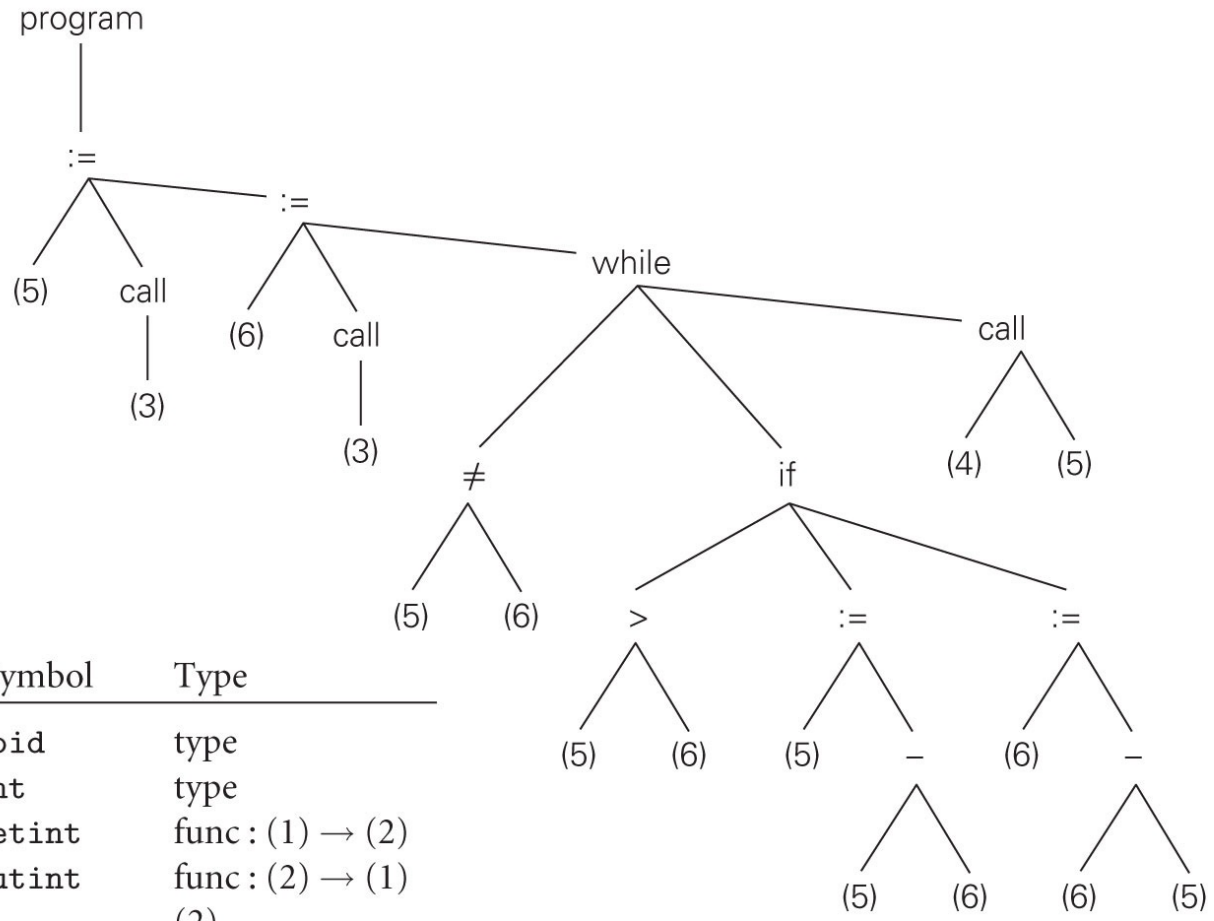
Role of Semantic Analysis

- Parse tree review (cont'd)



Role of Semantic Analysis

- Parse tree can be replaced by the smaller *syntax tree* (review)



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)

Role of Semantic Analysis

- Dynamic checks
- compiler generates code for dynamic checking
- can be disabled for increased speed
  - Tony Hoare: “The programmer who disables semantic checks is like a sailing enthusiast who wears a life jacket when training on dry land but removes it when going to sea.”
- C – almost no checks
- Java – as many checks as possible
- trend is towards stricter rules
- Example: `3 + “four”`
  - Perl – attempts to infer meaning
  - Python – run-time error

Role of Semantic Analysis

- *Logical Assertions*
- Java:

```
assert denominator != 0;
```

`AssertionError` – exception thrown if semantic check fails

- C:

```
assert(denominator != 0);
myprog.c:42: failed assertion ‘denominator != 0’
```

- Python:

```
assert denominator != 0, “Zero denominator!”
AssertionError: Zero denominator!
```

- Invariants, preconditions, postconditions
  - Euclid, Eiffel, Ada
  - invariant: expected to be true at all check points
  - pre/postconditions: true at beginning/end of subroutines

Role of Semantic Analysis

- Static analysis
  - compile-time algorithms that predict run-time behavior
  - extensive static analysis eliminates the need for some dynamic checks
    - precise type checking
    - enforced initialization of variables

Attribute Grammars

- *Attribute grammar*:
  - formal framework for decorating the parse or syntax tree
  - for semantic analysis
  - for (intermediate) code generation
- Implementation
  - automatic
    - tools that construct semantic analyzers (attribute evaluator)
  - ad hoc
    - action routines

Attribute Grammars

- Example: LR (bottom-up) grammar
  - arithmetic expr. with constants, precedence, associativity
  - the grammar alone says nothing about the meaning
  - *attributes*: connection with mathematical concept

1. $E_1 \rightarrow E_2$ `+` T
2. $E_1 \rightarrow E_2$ `-` T
3. $E \rightarrow T$
4. $T_1 \rightarrow T_2$ `*` F
5. $T_1 \rightarrow T_2$ `/` F
6. $T \rightarrow F$
7. $F_1 \rightarrow$ `-` F_2
8. $F \rightarrow$ `(` E `)`
9. $F \rightarrow$ `const`

Attribute Grammars

- Attribute grammar
- `S.val`: the arithmetic value of the string derived from `S`
- `const.val`: provided by the scanner
- copy rules: 3, 6, 8, 9
- semantic functions: `sum`, `diff`, `prod`, `quot`, `add_inv`
  - use only attributes of the current production

1. $E_1 \rightarrow E_2 + T$ ▷ E_1`.val := sum(`E_2`.val, T.val)`
2. $E_1 \rightarrow E_2 - T$ ▷ E_1`.val := diff(`E_2`.val, T.val)`
3. $E \rightarrow T$ ▷ `E.val := T.val`
4. $T_1 \rightarrow T_2 * F$ ▷ T_1`.val := prod(`T_2`.val, F.val)`
5. $T_1 \rightarrow T_2 / F$ ▷ T_1`.val := quot(`T_2`.val, F.val)`
6. $T \rightarrow F$ ▷ T_1`.val := F.val`
7. $F_1 \rightarrow -F_2$ ▷ F_1`.val := add_inv(`F_2`.val)`
8. $F \rightarrow (E)$ ▷ `F.val := E.val`
9. $F \rightarrow$ **const** ▷ `F.val := const.val`

Attribute Grammars

- Example: LL (top-down) grammar
  - count the elements of a list
  - “in-line” notation of semantic functions

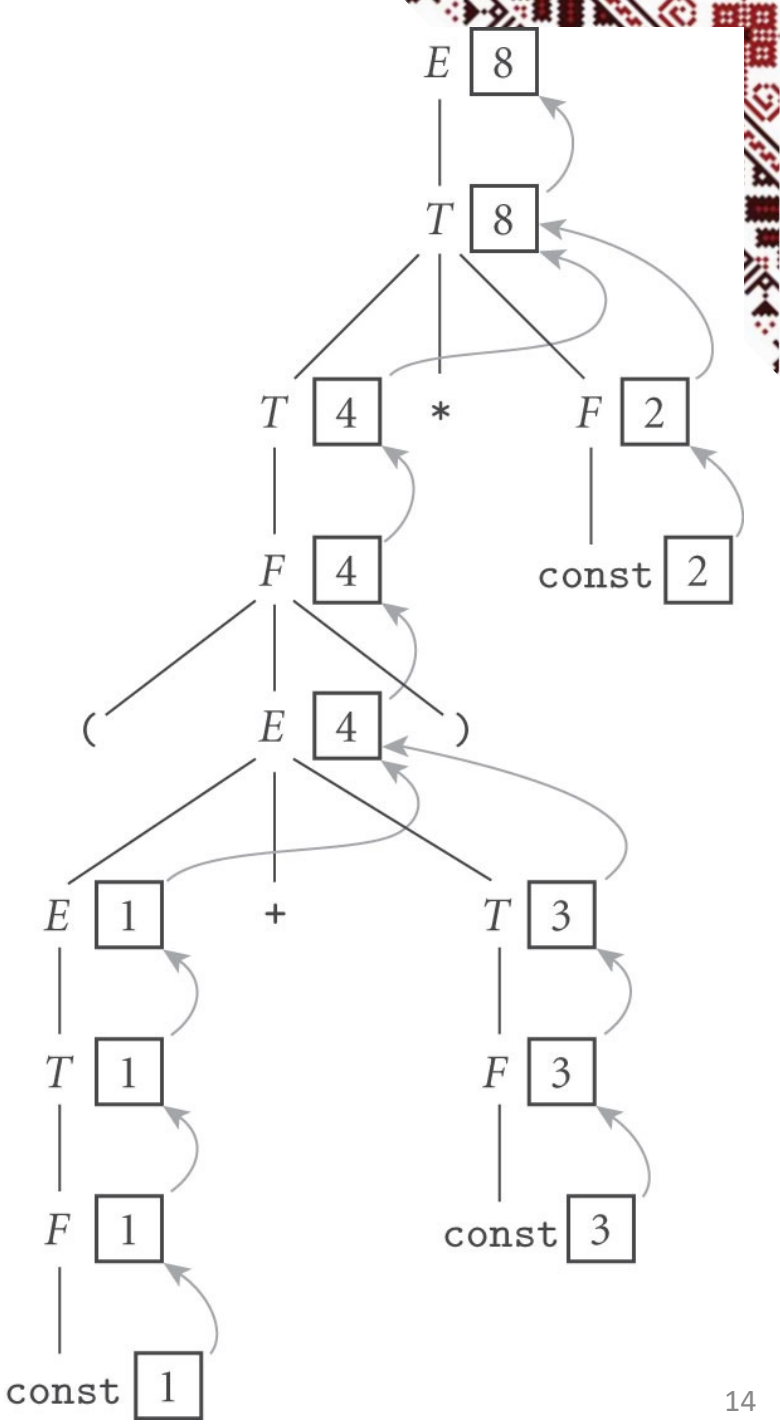
$L \rightarrow$ `id` LT ▷ `L.c := 1 + LT.c`

$LT \rightarrow$ `,` L ▷ `LT.c := L.c`

$LT \rightarrow \varepsilon$ ▷ `LT.c := 0`

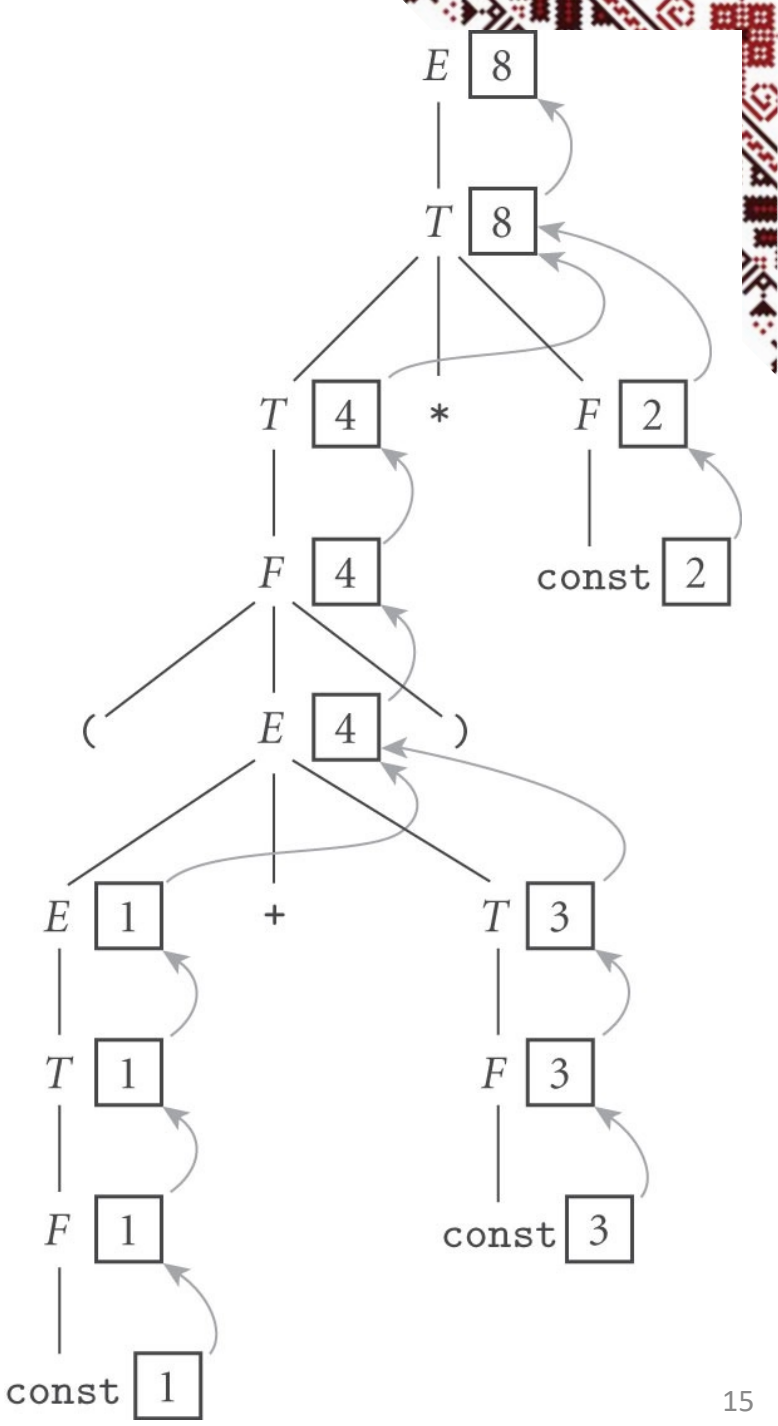
Evaluating Attributes

- *Annotation* of parse tree:
  - evaluation of attributes
  - also called *decoration*
- Example:
  - LR(1) grammar (arithm. exp.)
  - string: `(1+3)*2`
  - *val* attribute of root will hold the value of the expression



Evaluating Attributes

- Types of attributes:
  - synthesized
  - inherited
- *Synthesized* attributes:
  - values calculated only in productions where they appear only on the left-hand side
  - *attribute flow*: bottom-up only
- *S-attributed* grammar: all attributes are synthesized



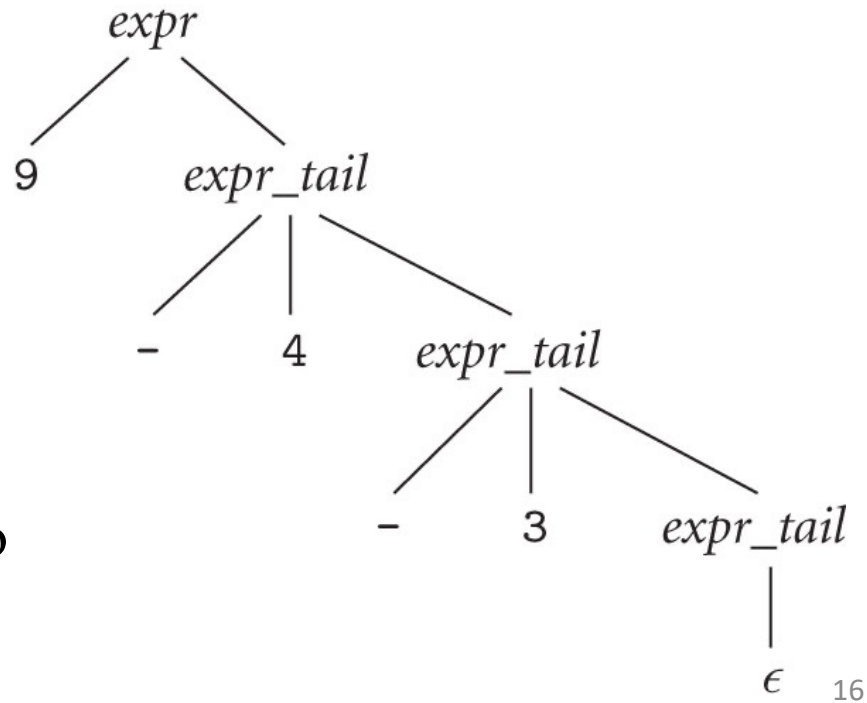
Evaluating Attributes

- *Inherited* attributes:
  - values calculated when their symbol is on RHS of the production
  - Example: LL(1) grammar for subtraction

expr → `const` *expr_tail*

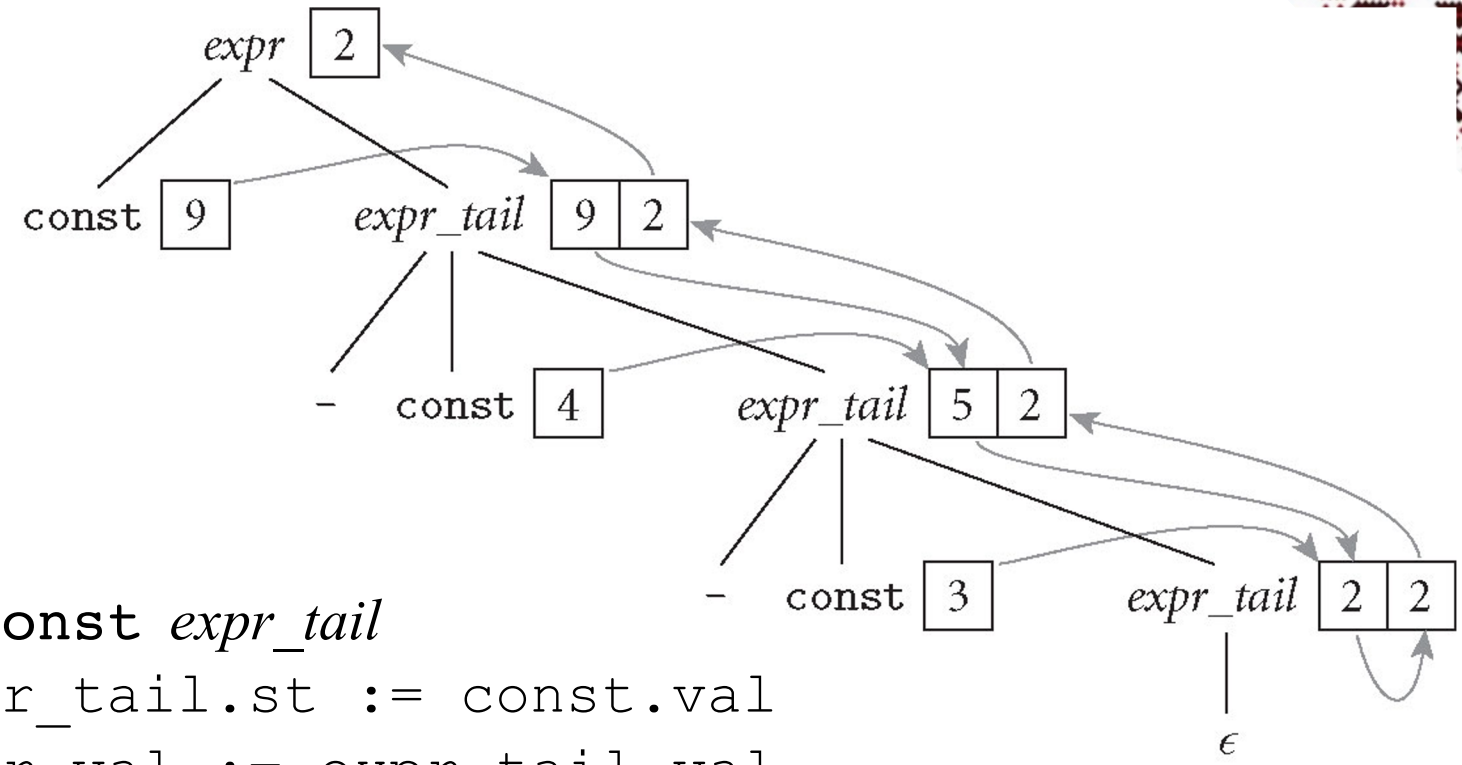
expr_tail → `-` `const` *expr_tail*

→ ε



- string: `9 - 4 – 3`
- `'-'` left-associative means cannot have only bottom-up
- need to pass `9` to *expr_tail* to combine with `4`

Evaluating Attributes



expr → **const** *expr_tail*

- ▷ expr_tail.st := const.val
- ▷ expr.val := expr_tail.val

$expr\_tail_1$ → **-** **const** $expr\_tail_2$

- ▷ $expr\_tail_2$.st := $expr\_tail_1$.st − const.val
- ▷ $expr\_tail_1$.val := $expr\_tail_2$.val

expr_tail → ε

- ▷ expr_tail.val := expr_tail.st

Evaluating Attributes

- Example: Complete LL(1) grammar for arithmetic expressions
- Complicated because:
  - Operators are left-associative but grammar cannot be left-recursive
  - Left and right operands of an operator are in separate productions

1. $E \longrightarrow T\ TT$
   ▷ TT.st := T.val ▷ E.val := TT.val
2. $TT_1 \longrightarrow +\ T\ TT_2$
   ▷ TT_2.st := TT_1.st + T.val ▷ TT_1.val := TT_2.val
3. $TT_1 \longrightarrow -\ T\ TT_2$
   ▷ TT_2.st := TT_1.st − T.val ▷ TT_1.val := TT_2.val
4. $TT \longrightarrow \epsilon$
   ▷ TT.val := TT.st
5. $T \longrightarrow F\ FT$
   ▷ FT.st := F.val ▷ T.val := FT.val

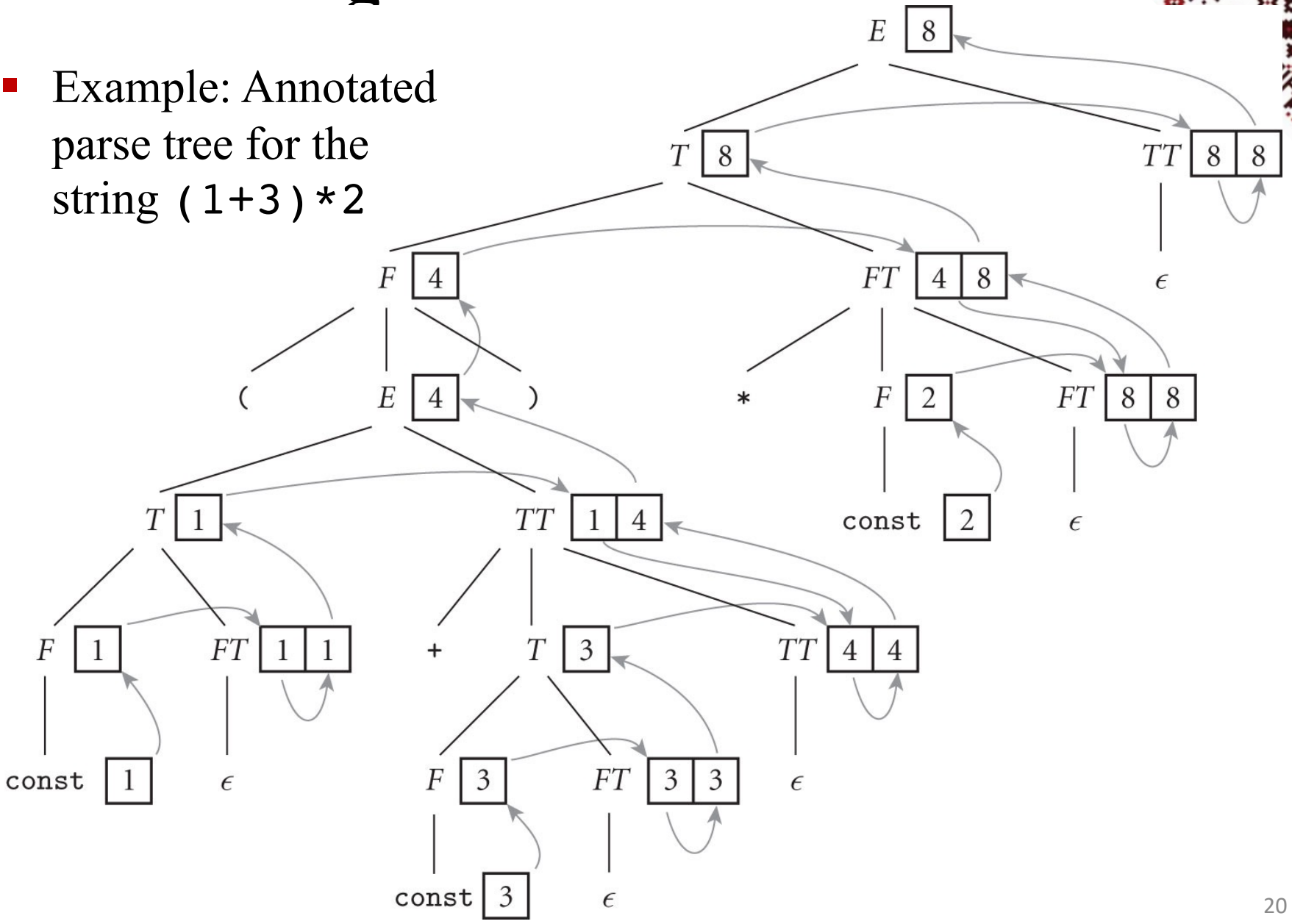
Evaluating Attributes

- Example: LL(1) grammar for arithmetic expressions (cont'd)

6. $FT_1 \longrightarrow * \; F \; FT_2$
   ▷ FT_2.st := FT_1.st × F.val ▷ FT_1.val := FT_2.val
7. $FT_1 \longrightarrow / \; F \; FT_2$
   ▷ FT_2.st := FT_1.st ÷ F.val ▷ FT_1.val := FT_2.val
8. $FT \longrightarrow \epsilon$
   ▷ FT.val := FT.st
9. $F_1 \longrightarrow - \; F_2$
   ▷ F_1.val := − F_2.val
10. $F \longrightarrow (\; E \;)$
    ▷ F.val := E.val
11. $F \longrightarrow$ `const`
    ▷ F.val := const.val

Evaluating Attributes

- Example: Annotated parse tree for the string `(1+3)*2`



Evaluating Attributes

- *Attribute flow*
- *Declarative* notation:
  - no evaluation order specified for attributes
- *Well-defined* grammar:
  - its rules determine unique values for attributes in any parse tree
- *Non-circular* grammar:
  - no attribute depends on itself in any parse tree
- *Translation scheme*:
  - algorithm that decorates parse tree in agreement with the attribute flow

Evaluating Attributes

- Translation scheme:
  - Obvious scheme: repeated passes until no further changes
    - halts only if well defined
  - Dynamic scheme: better performance
    - topologically sort the attribute flow graph
- *Static* scheme: fastest, $O(n)$
  - based on the structure of the grammar
- S-attributed grammar – simplest static scheme
  - flow is strictly bottom-up; attributes can be evaluated in the same order the nodes are generated by an LR-parser

Evaluating Attributes

- Attribute `A.s` is said to *depend* on attribute `B.t` if `B.t` is ever passed to a semantic function that returns a value for `A.s`

- *L-attributed grammar*:
  - each synthesized attribute of a LHS symbol depends only on that symbol's own inherited attributes or on attributes (synthesized or inherited) of the RHS symbols
  - each inherited attribute of a RHS symbol depends only on inherited attributes of the LHS symbol or on attributes (synthesized or inherited) of symbols to its left in the RHS

- L-attributed grammar
  - attributes can be evaluated by a single left-to-right depth-first traversal

Evaluating Attributes

- S-attributed implies L-attributed (but not vice versa)
- S-attributed grammar: the most general class of attribute grammars for which evaluation can be implemented on the fly during an LR parse
- L-attributed grammar: the most general class of attribute grammars for which evaluation can be implemented on the fly during an LL parse
- If semantic analysis interleaved with parsing:
  - bottom-up parser paired with S-attribute translation scheme
  - top-down parser paired with L-attributed translation scheme

Syntax Tree

- *One-pass compiler*
  - interleaved: parsing, semantic analysis, code generation
  - saves space (older computers)
    - no need to build parse tree or syntax tree
- *Multi-pass compiler*
  - possible due to increases in speed and memory
  - more flexible
  - better code improvement
    - Example: forward references
    - declaration before use no longer necessary

Syntax Tree

- *Syntax Tree*
  - separate parsing and semantics analysis
  - attribute rules for CFG are used to build the syntax tree
  - semantics easier on syntax tree
    - syntax tree reflects semantic structure better
    - can pass the tree in different order than that of parser

Syntax Trees Construction

- Bottom-up (S-attributed) attribute grammar to construct syntax tree

$E_1 \longrightarrow E_2 + T$

▷ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)

$E_1 \longrightarrow E_2 - T$

▷ E_1.ptr := make_bin_op("−", E_2.ptr, T.ptr)

$E \longrightarrow T$

▷ E.ptr := T.ptr

$T_1 \longrightarrow T_2 * F$

▷ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)

$T_1 \longrightarrow T_2 / F$

▷ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)

Syntax Trees Construction

- Bottom-up (S-attributed) attribute grammar to construct syntax tree (cont'd)

$T \longrightarrow F$
▷ T.ptr := F.ptr

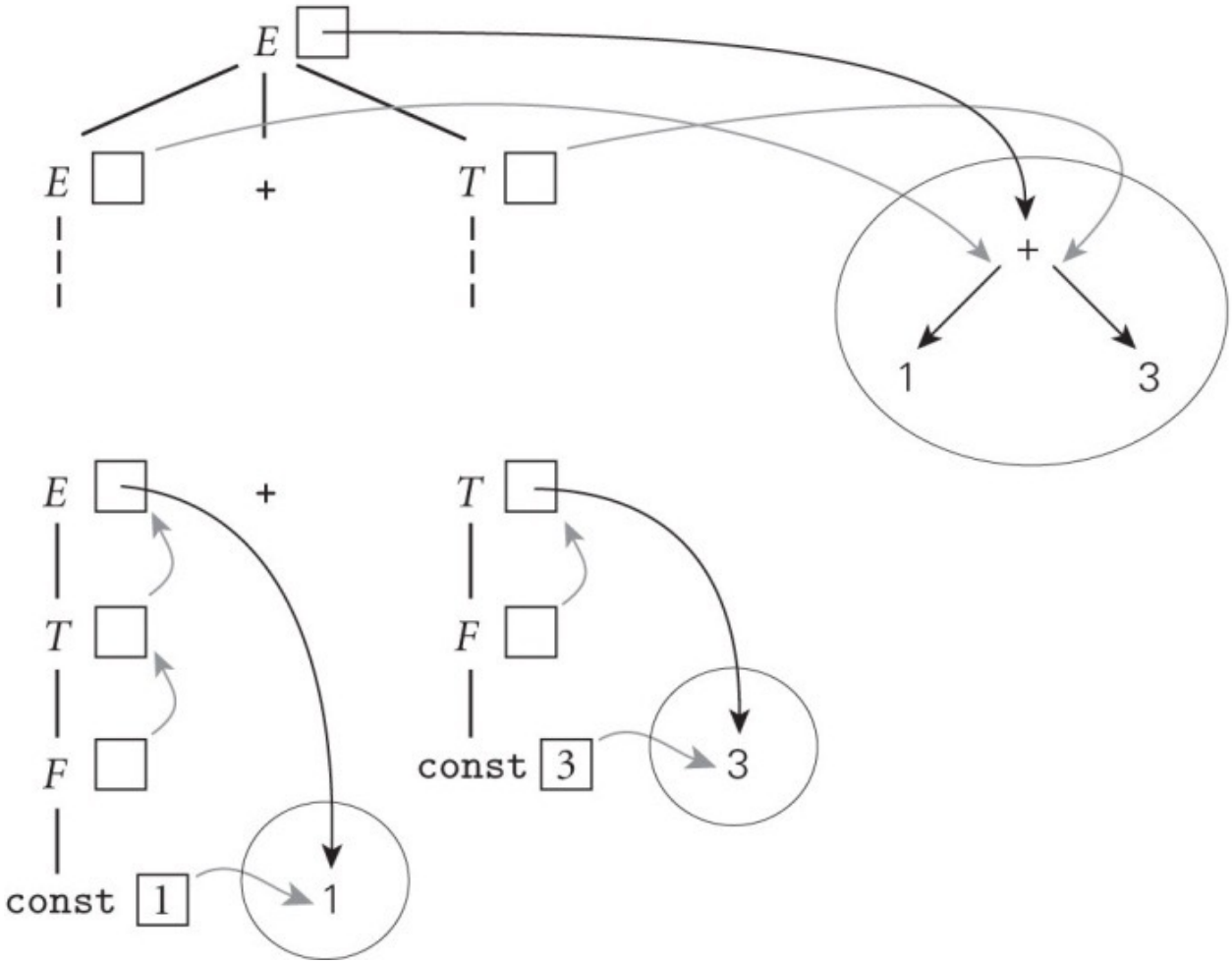
$F_1 \longrightarrow$ - F_2
▷ F_1.ptr := make_un_op("+/_", F_2.ptr)

$F \longrightarrow$ (E)
▷ F.ptr := E.ptr

$F \longrightarrow$ `const`
▷ F.ptr := make_leaf(const.val)

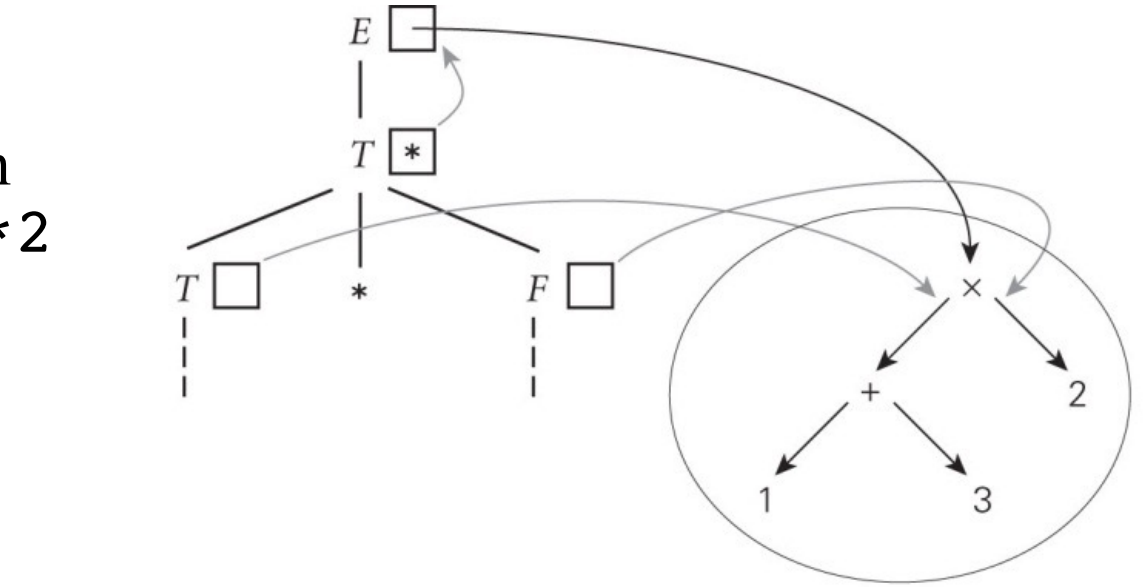
Syntax Trees Construction

- Syntax tree construction for `(1+3)*2`

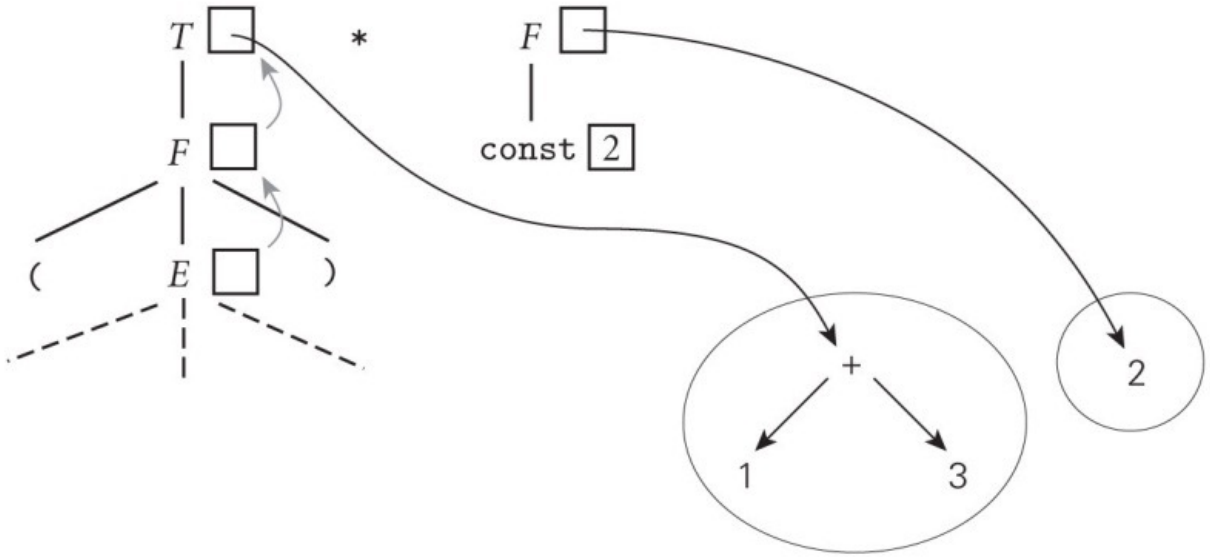


Syntax Trees Construction

- Syntax tree construction for `(1+3)*2` (cont'd)



(d)



(c)

Syntax Trees Construction

- Top-down (L-attributed) attribute grammar to construct syntax tree

$E \longrightarrow T\ TT$

▷ TT.st := T.ptr

▷ E.ptr := TT.ptr

$TT_1 \longrightarrow +\ T\ TT_2$

▷ TT_2.st := make_bin_op("+", TT_1.st, T.ptr)

▷ TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow -\ T\ TT_2$

▷ TT_2.st := make_bin_op("−", TT_1.st, T.ptr)

▷ TT_1.ptr := TT_2.ptr

$TT \longrightarrow \epsilon$

▷ TT.ptr := TT.st

$T \longrightarrow F\ FT$

▷ FT.st := F.ptr

▷ T.ptr := FT.ptr

Syntax Trees Construction

- Top-down (L-attributed) attribute grammar to construct syntax tree (cont'd)

$FT_1 \longrightarrow$ * F FT_2

▷ FT_2.st := make_bin_op("×", FT_1.st, F.ptr)

▷ FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow$ / F FT_2

▷ FT_2.st := make_bin_op("÷", FT_1.st, F.ptr)

▷ FT_1.ptr := FT_2.ptr

$FT \longrightarrow \epsilon$

▷ FT.ptr := FT.st

$F_1 \longrightarrow$ - F_2

▷ F_1.ptr := make_un_op("+/_", F_2.ptr)

$F \longrightarrow$ (E)

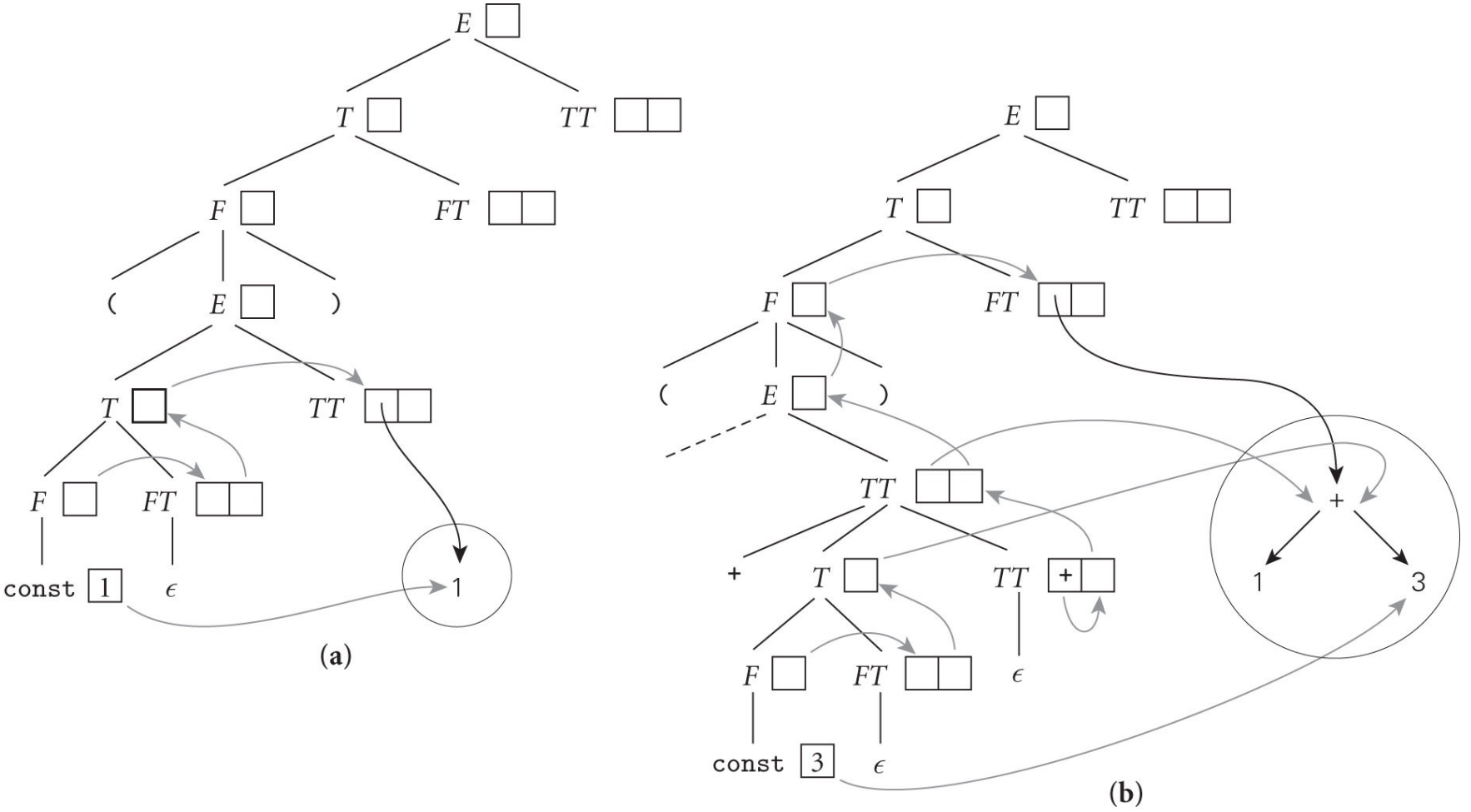
▷ F.ptr := E.ptr

$F \longrightarrow$ `const`

▷ F.ptr := make_leaf(const.val)

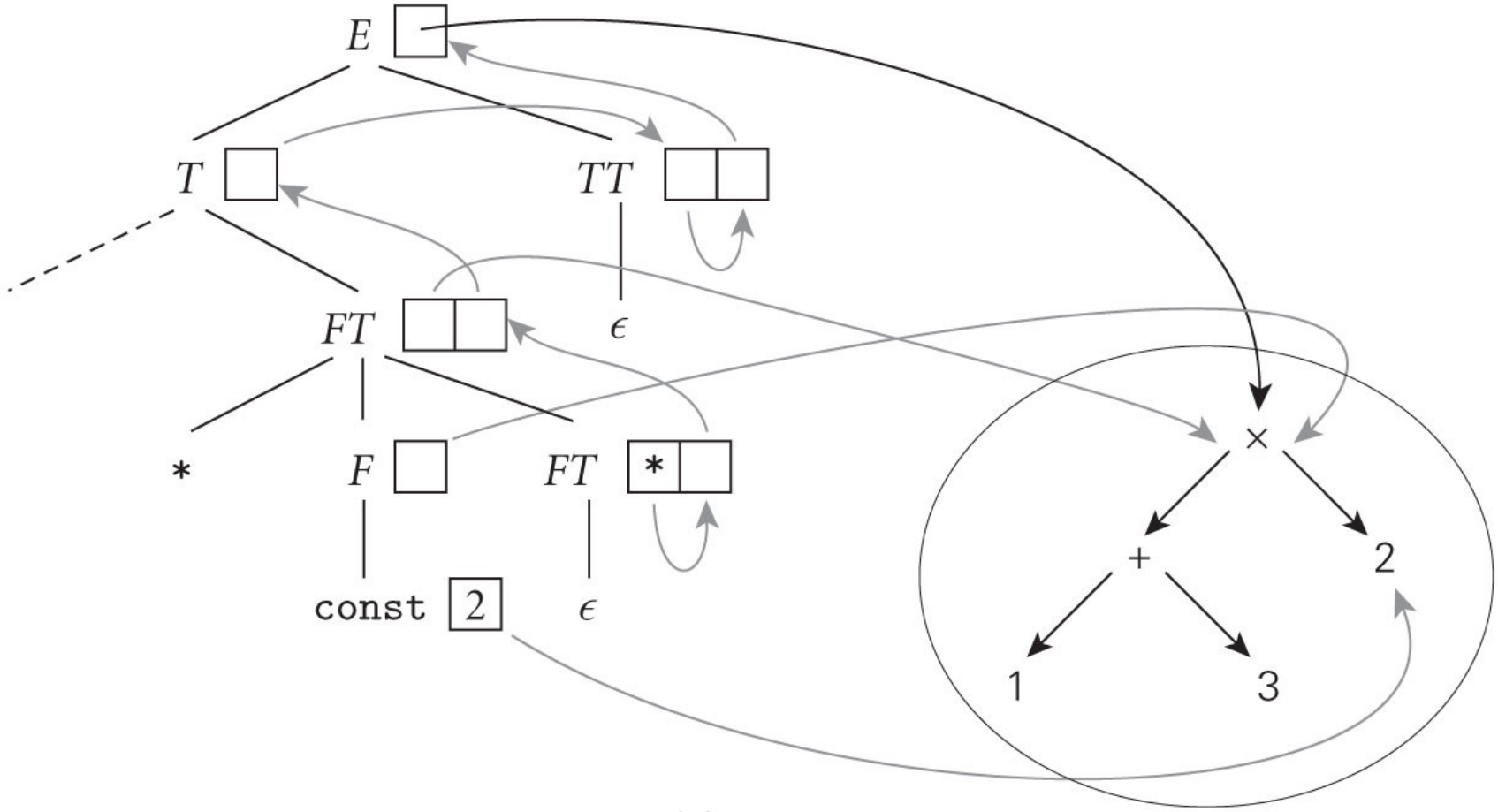
Syntax Trees Construction

- Syntax tree for `(1+3)*2`



Syntax Trees Construction

- Syntax tree for `(1+3)*2` (cont'd)



(c)

Action Routines

- There are automatic tools for:
  - Context-free grammar ⟹ parser
  - Attribute grammar ⟹ semantic analyzer (attrib. eval.)
- *Action routines*
  - ad-hoc approach; most ordinary compilers use (!)
  - Interleave parsing, syntax tree construction, other aspects of semantic analysis, code generation
  - Action routine: Semantic function that the programmer (grammar writer) instructs the compiler to execute at some point in the parse
  - In an LL grammar, can appear anywhere in the RHS; called as soon as the parser matched the (yield of the) symbol to the left

Action Routines - Example

- LL(1) grammar for expressions
  - with action routines for building the syntax tree
  - only difference from before: actions embedded in RHS

$E \longrightarrow T$ { TT.st := T.ptr } TT { E.ptr := TT.ptr }

$TT_1 \longrightarrow$ + T { TT_2.st := make_bin_op("+", TT_1.st, T.ptr) } TT_2 { TT_1.ptr := TT_2.ptr }

$TT_1 \longrightarrow$ - T { TT_2.st := make_bin_op("−", TT_1.st, T.ptr) } TT_2 { TT_1.ptr := TT_2.ptr }

$TT \longrightarrow \epsilon$ { TT.ptr := TT.st }

$T \longrightarrow F$ { FT.st := F.ptr } FT { T.ptr := FT.ptr }

$FT_1 \longrightarrow$ * F { FT_2.st := make_bin_op("×", FT_1.st, F.ptr) } FT_2 { FT_1.ptr := FT_2.ptr }

$FT_1 \longrightarrow$ / F { FT_2.st := make_bin_op("÷", FT_1.st, F.ptr) } FT_2 { FT_1.ptr := FT_2.ptr }

$FT \longrightarrow \epsilon$ { FT.ptr := FT.st }

$F_1 \longrightarrow$ - F_2 { F_1.ptr := make_un_op("+/_", F_2.ptr) }

$F \longrightarrow$ (E) { F.ptr := E.ptr }

$F \longrightarrow$ `const` { F.ptr := make_leaf(const.ptr) }

Action Routines - Example

- Recursive descent parsing with embedded action routines:

```
procedure term_tail(lhs : tree_node_ptr)
    case input_token of
        +, - :
            op : string := add_op()
            return term_tail(make_bin_op(op, lhs, term()))
                -- term() is a recursive call with no arguments
        ), id, read, write, $$ :        -- epsilon production
            return lhs
        otherwise parse_error
```

- does the same job as productions 2-4:

$TT_1 \longrightarrow$ + T { TT$_2$.st := make_bin_op("+", TT$_1$.st, T.ptr) } TT_2 { TT$_1$.ptr := TT$_2$.ptr }

$TT_1 \longrightarrow$ - T { TT$_2$.st := make_bin_op("–", TT$_1$.st, T.ptr) } TT_2 { TT$_1$.ptr := TT$_2$.ptr }

$TT \longrightarrow \epsilon$ { TT.ptr := TT.st }

Action Routines - Example

- Bottom-up evaluation
  - In LR-parser action routines cannot be embedded at arbitrary places in the RHS
  - the parser needs to see enough to identify the production, i.e., the RHS suffix that identifies the production uniquely
  - Previous bottom-up examples are identical with the action routine versions

Control Flow

Chapter 6

Control Flow

- Basic paradigms for control flow:
  - Sequencing
  - Selection
  - Iteration
  - Procedural Abstraction
  - Recursion
  - Concurrency
  - Exception Handling
  - Nondeterminacy

Control Flow

- Sequencing:
  - major role in imperative languages
  - minor role in functional languages
- Recursion
  - major role in functional languages
  - less important in imperative languages (iteration)
- Logic programming
  - no control flow at all
  - programmer specifies a set of rules
  - the implementation finds the order to apply the rules

Expression Evaluation

- Expression: operands and operators
- Operator
  - function: *a* + *b* means +(*a*, *b*)
  - Ada: `a+b` is short for `"+"(a,b)`
  - C++: `a+b` is short for `a.operator+(b)`
- Notation
  - *prefix* + a b or +(a, b) or (+ a b)
  - *infix* a + b
  - *postfix* a b +
- Infix: common notation; easy to work with
- Pre/Postfix: precedence/associativity not needed

Expression Evaluation

- Infix: binary operators:

```
a + b
```

- Prefix: unary operators, function calls (with parentheses)

```
-4, f(a, b)
```

- Scheme: prefix always – *Cambridge Polish* notation

```
(+ (* 1 2) 3)
(append x y my_list)
```

- Postfix: Pascal dereferencing `^`, C post in/decrement

```
a++, a--
```

- Ternary operators: C++ conditional operator '`?:`'

```
(a > b) ? a : b
```

Expression Evaluation

- Precedence, associativity
- Fortran example: `a + b * c ** d ** e / f`
- Precedence levels
- C, C++, Java, C#: too many levels to remember (15)
- Pascal: too few for good semantics

  `if A < B and C < D then …` means

  `if A < (B and C) < D then …`
- Fortran has 8 levels
- Ada has 6 (it puts *and* & *or* at same level)
- Associativity: usually left associative
  - Right associative; C: `a = b = c` means `a = (b = c)`
- **Lesson**: when unsure, use parentheses!

Expression Evaluation

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /= , <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if…then…else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Expression Evaluation

- *Side Effect*:
  - any effect other than returning a value to surrounding context
  - essential in imperative programming
    - computing by side effects
  - (pure) functional languages: no side effects
    - same value returned by an expression at any point in time
- *Value vs Reference*
  - `d = a` value of `a`
  - `a = b + c` location of `a`
  - *Value model:* a variable is a named container for a value
    - C, Pascal, Ada
  - *Reference model*: a variable is a named reference to a value
    - Scheme, Lisp, Python, Clu

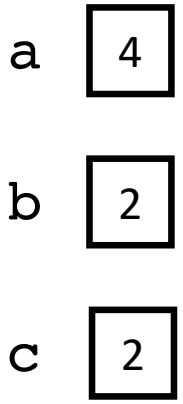
Expression Evaluation

- Example:

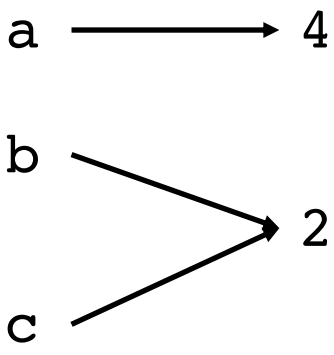
```
b := 2
c := b
a := b + c
```

- Pascal (value model):
  - any variable can contain value 2
- Clu (reference model):
  - there is only one 2

- value model



- reference model



Expression Evaluation

- Value vs Reference
- Java: in-between
  - built-in types – value model
  - user-defined types – reference model
  - drawback: built-in types cannot be passed when user-defined is expected – wrapping is used (boxing)
- C#: user can choose
  - `class` – reference
  - `struct` – value
- Important to distinguish between variables referring to:
  - the same object or
  - different objects whose values happen to be equal
  - Scheme, Lisp provide several notions of “equality”

Subroutines: Parameter Passing

- Call by *value*: pass the value
  - C, C++, Pascal, Java, C#
- Call by *reference*: pass the address
  - Fortran, C++, C (pointers)
- Call by *sharing*:
  - Java, C#, Python, Scheme
- Call by *name*: direct substitution; evaluated each time it is needed
  - Algol 60, Simula
- Call by *need*: call by name with memoization
  - Haskell, R

Short-circuiting

- Short-circuiting

```
(a < b) && (c < d)
```

  - if `a` > `b` then the second part does not matter
- *Short-circuit evaluation*: evaluate only what is needed
  - *Lazy evaluation*
  - can save time:

```
if (unlikely_cond && expensive_cond) ...
```

- Semantics change:
  - Avoiding out-of-bounds indices:

```
if (i >= 0 && i < MAX && A[i] > foo) ...
```

  - Avoiding division by zero:

```
if (d == 0 || n/d < threshold) ...
```

Short-circuiting: example

- C list searching:

```
while (p && p->key != val)
    p = p -> next;
```

- Pascal does not have short circuit:

```
p := my_list;
still_searching := true;
while still_searching do
    if p = nil then
        still_searching := false
    else if p^.key = val then
        still_searching := false
    else p := p^.next;
```

- Sometimes side effects are desired
  - C has also non-short-circuit: `&`, `|`

Short-circuiting: implementation

`if` ((A > B) `and` (C > D)) `or` (E ≠ F) `then` *then_clause*
`else` *else_clause*

- Without short circuit

```
    r1 := A    -- load
    r2 := B
    r1 := r1 > r2
    r2 := C
    r3 := D
    r2 := r2 > r3
    r1 := r1 & r2
    r2 := E
    r3 := F
    r2 := r2 ≠ r3
    r1 := r1 | r2
    if r1 = 0 goto L2
L1: then_clause -- (L1 unused)
    goto L3
L2: else_clause
L3:
```

- With short circuit (*jump code*)

```
    r1 := A
    r2 := B
    if r1 <= r2 goto L4
    r1 := C
    r2 := D
    if r1 > r2 goto L1
L4: r1 := E
    r2 := F
    if r1 = r2 goto L2
L1: then_clause
    goto L3
L2: else_clause
L3:
```

Iteration

- Arbitrary complexity of programs:
  - Iteration – `for, while, …`
  - Recursion
- Iterate over collections
  - Iterator objects:
    - C++, Java, Euclid
  - True iterators:
    - Python, C#, Ruby, Clu
  - First-class functions
    - Scheme, Smalltalk

Iteration

- Python – user-defined iterator

```
class PowTwo:
    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n < self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

Iteration

- Python – user-defined iterator

```
a = PowTwo(3)
i = iter(a)
print(next(i))  # 1
print(next(i))  # 2
print(next(i))  # 4
print(next(i))  # raises StopIteration
```

True iterators

- Example – Python:

```
for i in range(first, last, step):
    ...
```

  - `range` – built-in iterator
  - use a call to a `yield` statement
  - like `return` but control goes back to iterator after each iteration
  - the iterator continues where it left off
- `yield` – separate thread of control
  - its own program counter
  - execution interleaved with that of the `for` loop

True iterators

- Python generator – much simpler

```
def PowTwoGen(max = 0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1

a = PowTwoGen(3)
print(next(a))  # 1
print(next(a))  # 2
print(next(a))  # 4
print(next(a))  # raises StopIteration
```

True iterators

- Python generator: can generate infinite stream

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
print(next(a))    # 0
print(next(a))    # 2
print(next(a))    # 4
print(next(a))    # 6
print(next(a))    # 8
print(next(a))    # 10
...
```

First-class functions

- Iteration with first-class functions

```
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
        (begin
          (f low)
          (uptoby (+ low step) high step f))
        '())))
(let ((sum 0))
  (uptoby 1 100 2
          (lambda (i)
            (set! sum (+ sum i))))
  sum)                                  ; 2500
```

Recursion

- Recursion vs Iteration – efficiency
- naïve implementation of recursion is less efficient
  - time and space needed for subroutine calls
- the language can generate fast code for recursion
- *Tail recursion*
  - no computation after the recursive call
  - as fast as iteration

```
int gcd(int a, int b) {   /* assume a,b > 0 */
    if (a == b) return a;
    else if (a > b) return gcd(a-b, b);
    else return gcd(a, b-a);
}
```

Recursion

- Tail recursion
  - can be implemented without the stack allocations
  - a good compiler can recast the recursive function as:

```
int gcd(int a, int b) {   /* assume a,b > 0 */
start:
    if (a == b) return a;
    else if (a > b) { a = a-b; goto start; }
    else { b = b-a; goto start; }
}
```

Recursion

- Scheme
- Recursive summation

```
(define sum1
 (lambda (f low high)
 (if (= low high)
  (f low)                                          ; then
  (+ (f low) (sum1 f (+ low 1) high))))) ; else
(sum1 + 1 10)              ; 55
```

Recursion

- Scheme
- Tail recursive summation

```
(define sum2
  (lambda (f low high st)
    (if (= low high)
        (+ st (f low))
        (sum2 f (+ low 1) high (+ st (f low))))))
(sum2 + 1 10 0)          ; 55
```

- Eliminate `st` (subtotal)

```
(define sum3
  (lambda (f low high)
    (sum2 f low high 0)))
```

Recursion

- Careless recursion can be very bad
- Exponential

```
def fib1(n):
    if n == 0 or n == 1:
        return 1
    return fib1(n-1) + fib1(n-2)
```

- Linear

```
def fib2(n):
    f1 = f2 = 1
    for i in range(n-1):
        f1, f2 = f2, f1 + f2
    return f2
```

Recursion

- Evaluation order (of subroutine arguments)
- *Applicative*: evaluate before passing
  - used by most languages
- *Normal-order*: pass unevaluated; evaluate when needed
  - *lazy evaluation*
  - short-circuit evaluation
  - macros
  - Scheme: used for infinite data structures
    - *lazy data structures*

Recursion

- Example: Scheme lazy (infinite) data structures
  - `delay` – a promise
  - `force` – forces evaluation

```
(define naturals
  (letrec ((next (lambda (n) (cons n (delay (next
(+ n 1)))))))
    (next 1)))
(define head car)
(define tail (lambda (stream) (force (cdr
stream))))
(head naturals)                    => 1
(head (tail naturals))             => 2
(head (tail (tail naturals)))      => 3
...
```

Types

Chapters 7, 8

Data Types

- Types provide implicit context for operations
- C: `a + b`
  - integer/floating point addition
- Pascal: `new p`
  - allocate right size
- C: `new my_type()`
  - allocate right size
  - call right constructor

Data Types

- Boolean
  - true/false; one byte, sometimes one bit
  - C: integers, true = non-0, false = 0
- Character
  - one byte – ASCII
  - two bytes - Unicode
- Numeric
  - integers, reals
  - complex: C, Fortran, Scheme (pair of floats)
  - rational: Scheme (pair of integers)
- Discrete (or ordinal)
  - integers, Booleans, characters
  - countable, well-define predecessor/successor
- Scalar (or simple): discrete, rational, real, complex

Data Types

- Enumeration
  - introduced in Pascal:

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
```

  - ordered: `mon < tue`; can index an array
- Subrange: `type test_score = 0..100`
- Composite (non-scalar)
  - Records (struct) – collection of fields
  - Arrays – most common; map from index to elements
    - strings = arrays of characters
  - Sets – powerset of base type
  - Pointers –references to objects; recursive data types
  - Lists – sequence; no map; recursive definition, fundamental in functional programming
  - Files – like arrays but with current position

Type checking

- *Type equivalence*: two types are the same
- *Type compatibility*: a type can be used in a context
- *Type inference*: deduce the type from components
- *Type clash*: violation of type rules

Type Systems

- *Strongly typed* language
  - prohibits any application of an operation to an object that is not intended to support that operation
- *Statically typed* language
  - strongly typed
  - at compile time – good performance
  - C, C++, Java
  - C: more strongly typed with each new version
- *Dynamically typed* language
  - at run time - ease of programming
  - Scheme, Lisp, Smalltalk – strongly typed
  - Scripting: Python, Ruby – strongly typed

Type Checking: Equivalence

- *Structural equivalence*
  - same components put together in the same way
  - C, Algol-68, Modula-3, ML
- *Name equivalence*
  - lexical occurrence
  - each definition is a new type
  - Java, C#, Pascal

Type Checking: Equivalence

- Structural equivalence:
  - format should not matter

```
type R1 = record
  a, b : integer
end;
```

```
type R1 = record
  a : integer;
  b : integer;
end;
```

  - What about order? (most languages consider it equivalent)

```
type R3 = record
 b : integer;
 a : integer
end;
```

Type Checking: Equivalence

- Structural equivalence: problem

```
type student = record
   name, address : string
   age : integer
type school = record
   name, address : string
   age : integer
x : student;
y : school;
…
x := y; -- is this an error?
```

- compiler says it's okay
- programmer most likely says it's an error

Type Checking: Equivalence

- Name equivalence
  - Distinct definitions mean distinct types
  - If the programmer takes the time to write two type definitions, then they are different types
  - Aliases

```
type new_type = old_type (* Algol syntax *)
typedef old_type new_type /* C syntax */
```

  - Are aliases the same or different types?
  - Different: *strict name equivalence*
  - Same: *loose name equivalence*

Type Checking: Equivalence

- Strict name equiv.:
  - `blink` different from `alink`
  - `p,q` – same type; `r,u` – same type
- Loose name equiv.:
  - `blink, alink` – same type
  - `p,q` – same type; `r,s,u` – same type
- Structural equiv.:
  - `p,q,r,s,t,u` – same type

```
type cell = ...              -- whatever
type alink = pointer to cell
type blink = alink           -- alias
p, q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```

Type Checking: Conversion

- *Type conversion* (*cast*)*:* **explicit** conversion

```
r = (float) n;
```

- *Type coercion:* **implicit** conversion
  - very useful
  - weakens type security
  - dynamically typed languages: very important
  - statically typed languages: wide variety
    - C: arrays and pointers intermixed
    - C++: programmer-defined coercion to and from existing types to a new type (class)

Type Checking: Compatibility

- *Type compatibility*
  - more important than equivalence
  - most of the time we need compatibility
  - assignment: RHS compatible with LHS
  - operation: operands types compatible with a common type that supports the operation
  - subroutine call: arguments types compatible with formal parameters types

Type Checking: Inference

- *Type inference*
  - infer expression type from components
  - int + int => int
  - float + float => float
  - subranges cause complications

```
type Atype = 0..20; Btype = 10..20;
var a : Atype; b : Btype;
```

    - What is the type of `a + b` ?

Type Checking: Inference

- Type inference
  - declarations: type inferred from the context
  - C#: `var`

```
var i = 123;
// equiv. to:
int i = 123;

var map = new Dictionary<string, int>();
// equiv. to:
Dictionary<string, int> map = new
Dictionary<string, int>();
```

Type Checking: Inference

- C++: `auto`

```
auto reduce = [](list<int> L, int f(int, int), int s) {
                for (auto e : L) { s = f(e, s); }
                return s;
                };
...
int sum = reduce(my_list, [](int a,int b){return a+b;}, 0);
int prod = reduce(my_list, [](int a,int b){return a+b;}, 1);
```

- the `auto` keyword allows to omit the type:

```
int (*reduce) (list<int>, int (*)(int, int), int)
       = [](list<int> L, int f(int, int), int s) {
                for (auto e : L) { s = f(e, s); }
                return s;
                };
```

Type Checking: Inference

- C++: `decltype`
  - match the type of an existing expression
  - the type of `sum` depends on the types of `A` and `B` under the coercion rules of C++
  - both `int` gives `int`
  - one is `double` gives `double`

```
template <typename A, typename B>
...
   A a;
   B b;
   decltype(a + b) sum;
```

Polymorphism

- *Polymorphism* (polymorphous = multiform)
- code works with multiple types
  - must have common characteristics
- *parametric polymorphism*: take a type as parameter
  - explicit parametric polymorphism (*generics,* C++: *templates*) appears in statically typed languages
  - implemented at compile time
- *subtype polymorphism*: code works with subtypes
  - object-oriented languages
- combination (subtype + parametric polymorphism)
  - container classes
  - `List<T>`, `Stack<T>`; `T` instantiated later

Polymorphism

- *Implicit*:
  - Scheme:

```
(define min (lambda (a b) (if (< a b) a b)))
```

  - applied to arguments of any type to which it can be applied
  - disadvantage: checked dynamically
- *Explicit*: *generics*
  - C++: *templates*
  - checked statically
- Generics in object-oriented programming
  - parametrize entire class
  - *container*

Polymorphism

- Ada example: overloading (left) vs generics (right)

```
function min(x, y : integer)
    return integer is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function min(x, y : long_float)
    return long_float is
begin
    if x < y then return x;
    else return y;
    end if;
end min;
```

```
generic
    type T is private;
    with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;

function min(x, y : T) return T is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function int_min is new min(integer, "<");
function real_min is new min(long_float, "<");
function string_min is new min(string, "<");
function date_min is new min(date, date_precedes);
```

- C++ example

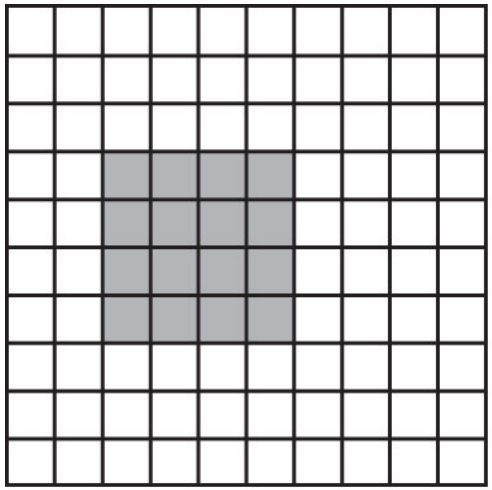
```
template<class item, int max_items = 100>
class queue {
    item items[max_items];
    int next_free, next_full, num_items;
public:
    queue() : next_free(0), next_full(0), num_items(0) { }
    bool enqueue(const item& it) {
        if (num_items == max_items) return false;
        ++num_items;  items[next_free] = it;
        next_free = (next_free + 1) % max_items;
        return true;
    }
    bool dequeue(item* it) {
        if (num_items == 0) return false;
        --num_items;  *it = items[next_full];
        next_full = (next_full + 1) % max_items;
        return true;
    }
};
...
queue<process> ready_list;
queue<int, 50> int_queue;
```

Arrays

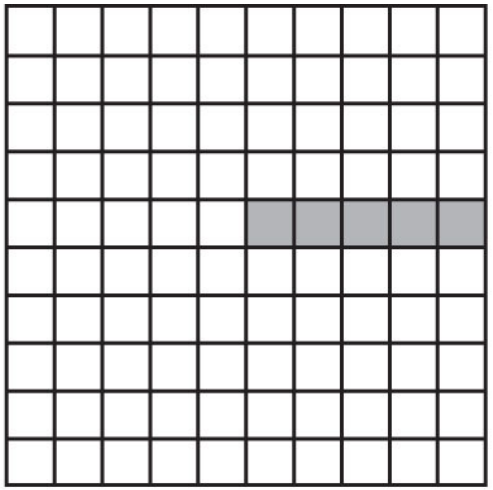
- Arrays
  - the most important composite data type
  - semantically, map: index type → element type
- Homogenous data
- Index type
  - usually discrete type: int, char, enum, subranges of those
  - non-discrete type: *associative array*, *dictionary*, *map*
    - implemented using hash tables or search trees
- Dense – most positions non-zero
  - sparse arrays – stored using linked structures

Arrays

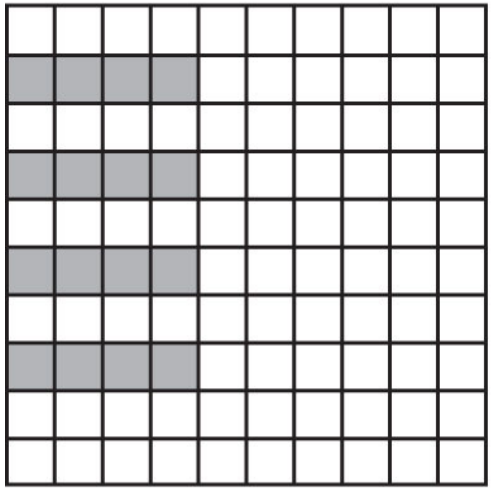
- Slices



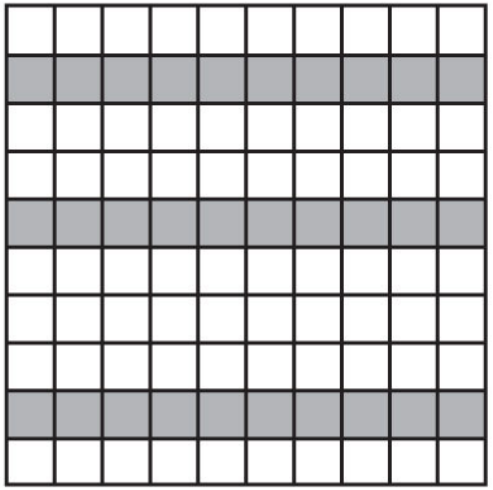
matrix(3:6, 4:7)



matrix(6:, 5)



matrix(:4, 2:8:2)



matrix(:, (/2, 5, 9/))

Arrays: Dimensions, Bounds, Allocation

- Static allocation:
  - array with lifetime the entire program
  - shape known at compile time
- Stack allocation:
  - array with lifetime inside subroutine
  - shape known at compile time
- Heap / stack allocation
  - dynamically allocated arrays
  - *dope vector*: holds shape information at run time
  - compiled languages need the number of dimensions
  - shape known at elaboration time – can allocate on stack
  - shape changes during execution: allocated on heap

Arrays

- Example: C dynamic local array

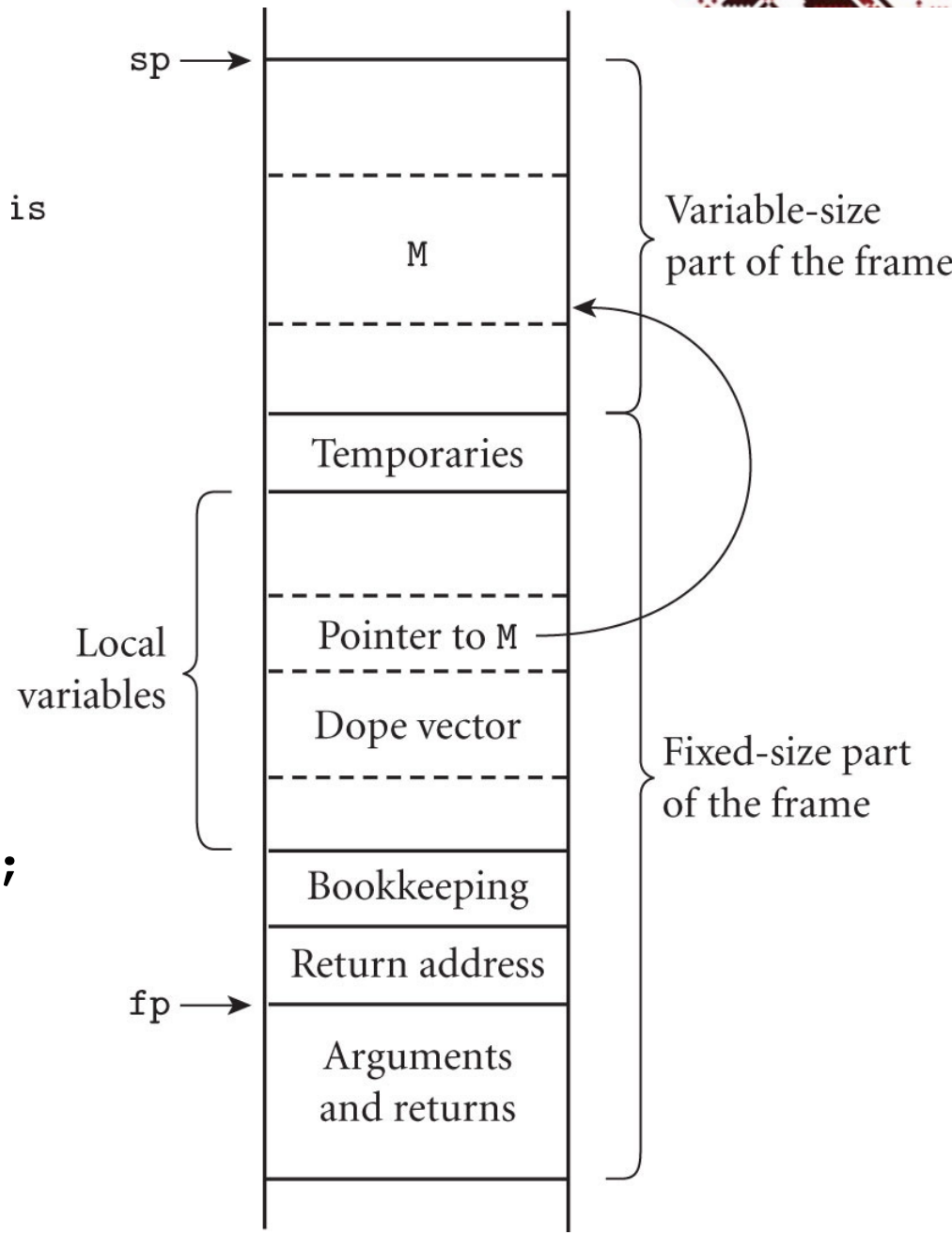
```
void square(int n, double M[n][n]) {
   double T[n][n];
   for (int i = 0; i < n; i++) { // copy product to T
      for (int j = 0; j < n; j++) {
         double s = 0;
         for (int k = 0; k < n; k++)
            s += M[i][k] * M[k][j];
         T[i][j] = s;
      }
   }
   for (int i = 0; i < n; i++) {// copy T back to M
      for (int j = 0; j < n; j++)
         M[i][j] = T[i][j];
   }
}
```

Arrays

- Shape known at elaboration time
  - can be allocated on stack
  - in the variable-size part

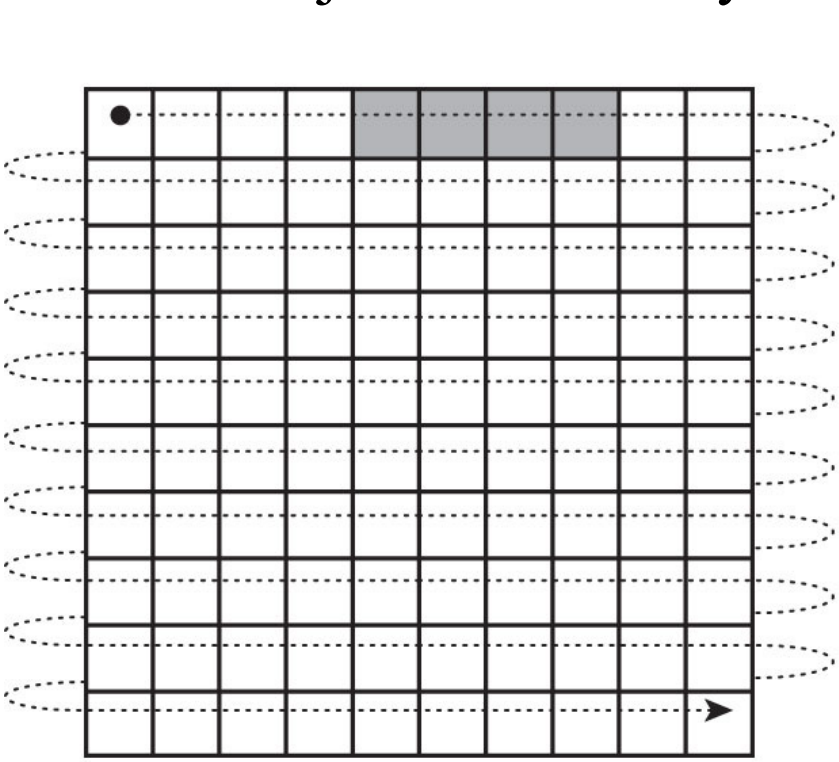
- Example:

```
// C99:
void foo (int size) {
   double M[size][size];
   ...
}
```

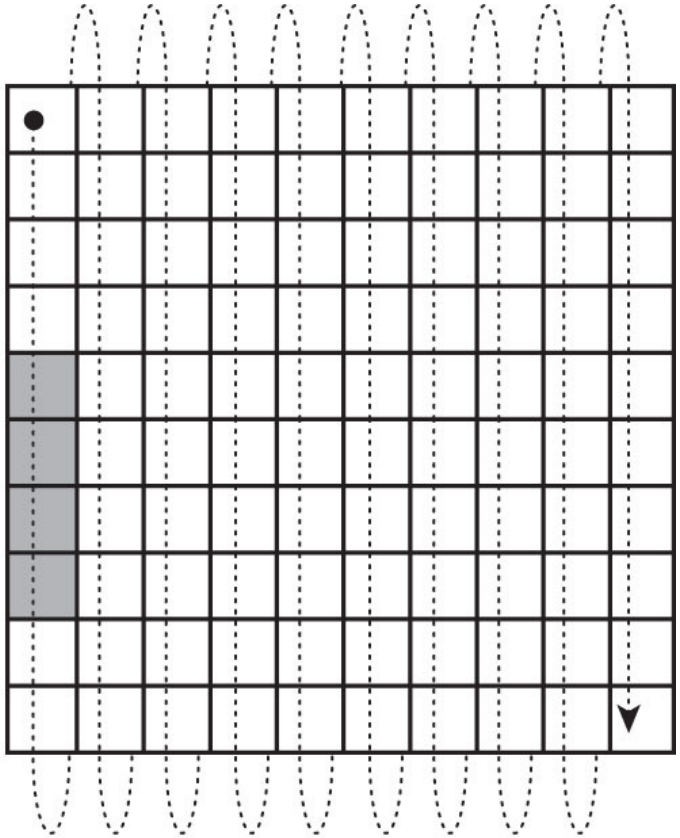


Arrays

- Memory layout
  - column major order – Fortran
  - row major order – everybody else



Row-major order



Column-major order

Arrays

- Memory layout
- Contiguous allocation
  - consecutive locations in memory: `A[2,4], A[2,5]`
  - consecutive rows adjacent in memory
- Row pointers
  - consecutive rows anywhere in memory
  - extra memory for pointers
  - rows can have different lengths (ragged array)
  - can construct an array from existing rows without copying
- C, C++, C# - allow both
- Java – only row-pointer for all arrays

Arrays

- Example: C – array of strings
  - true two-dimensional array

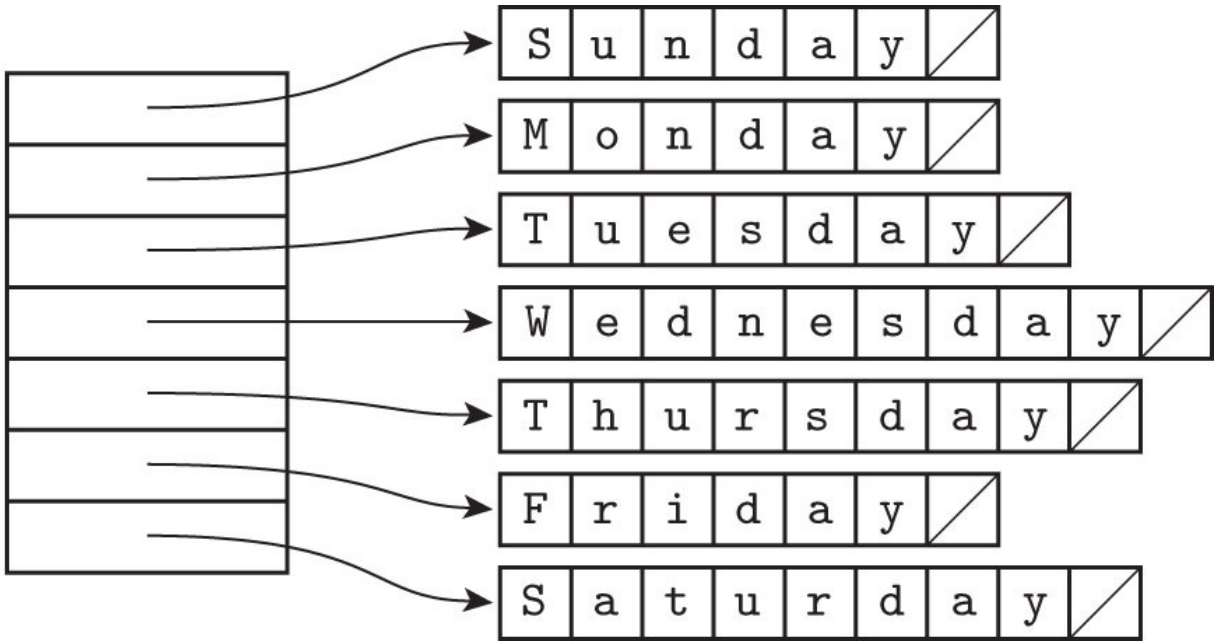
```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

S	u	n	d	a	y				
M	o	n	d	a	y				
T	u	e	s	d	a	y			
W	e	d	n	e	s	d	a	y	
T	h	u	r	s	d	a	y		
F	r	i	d	a	y				
S	a	t	u	r	d	a	y		

Arrays

- Example: C – array of strings
  - array of pointers

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```



Arrays

- Address calculation

`A : array [L1..U1] of array [L2..U2] of array [L3..U3] of elem_type;`

$S_3 =$ size of `elem_type`

$S_2 = (U_3 - L_3 + 1) \times S_3$

$S_1 = (U_2 - L_2 + 1) \times S_2$

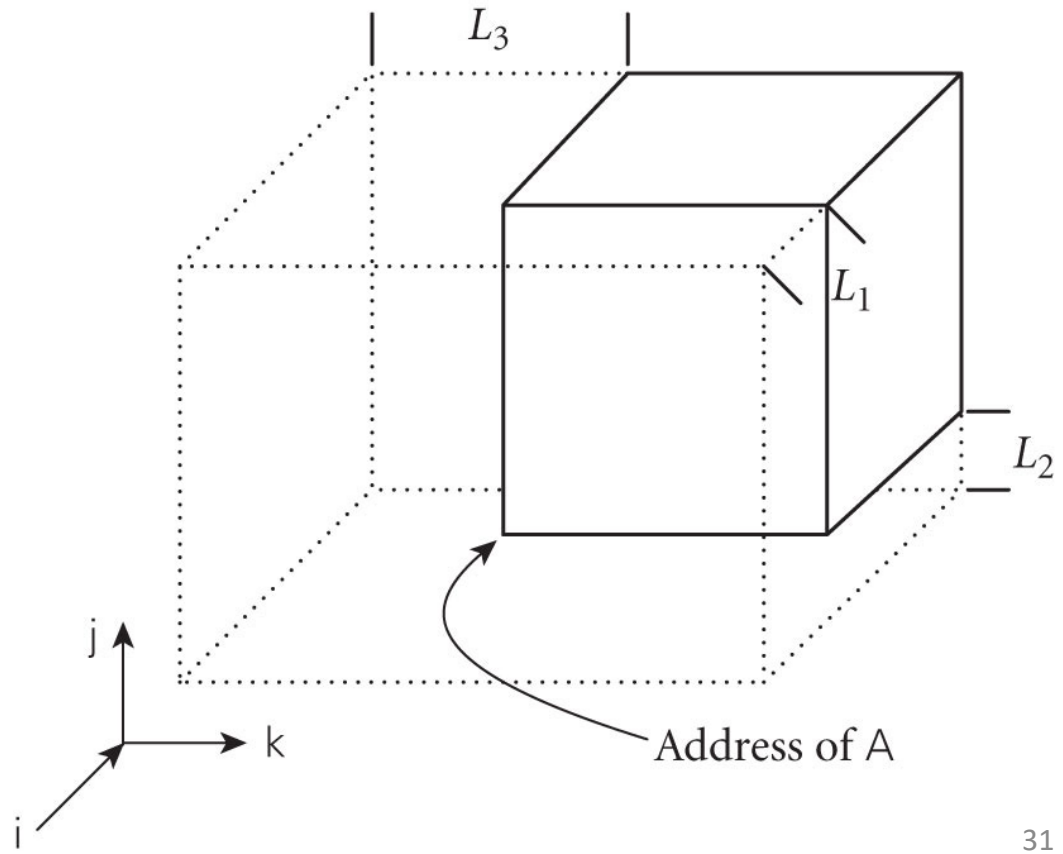
address of `A[i,j,k]`

$=$ address of `A`

$+ (i - L_1) \times S_1$

$+ (j - L_2) \times S_2$

$+ (k - L_3) \times S_3$



Arrays

- Faster address calculation

address of `A[i,j,k]`

$$= \text{address of } \texttt{A} + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3$$

- Fewer operations
  - $C = [(L_1 \times S_1) + (L_2 \times S_2) + (L_3 \times S_3)]$
  - C – known at compile time

address of `A[i,j,k]`

$$= \text{address of A} + (i \times S_1) + (j \times S_2) + (k \times S_3) - C$$

Sets

- *Set*: unordered collection of an arbitrary number of distinct values of a common type
- Implementation
  - *characteristic array* – one bit for each value (small base type)
    - efficient operations – bitwise op
  - general implementation: hash tables, trees, etc.
  - Python, Swift – built-in sets
  - Others use dictionaries, hashes, maps

```
X = set(['a', 'b', 'c', 'd']) # set constructor
Y = {'c', 'd', 'e', 'f'}      # set literal
U = X | Y                     # union
I = X & Y                     # intersection
D = X - Y                     # difference
O = X ^ Y                     # symmetric diff.
'c' in I                      # membership
```

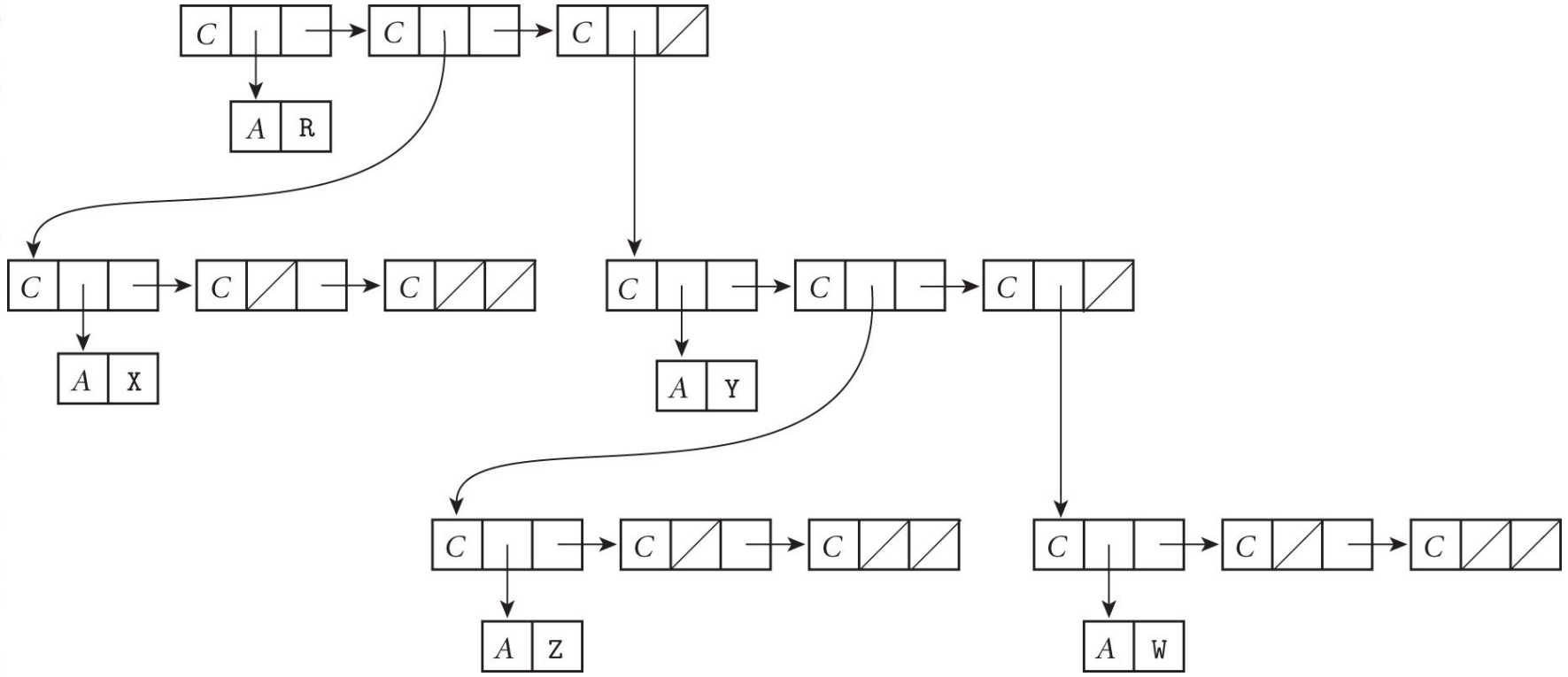
Pointers and Recursive Types

- *Pointer*
  - a variable whose value is a reference to some object
  - not needed with a reference model of variables
  - needed with a value model of variables
  - efficient access to complicated objects
- *Recursive type*
  - objects contain references to other objects
  - can create dynamic data structures
- Pointer ≠ address
  - pointer = high-level concept; reference to object
  - address = low-level concept; location in memory
  - pointers are implemented as addresses

Pointers and Recursive Types

- Reference model example: Tree in Scheme
- Two types of objects: (1) `cons` cells (2) atoms

```
'(#\R (#\X ()()) (#\Y (#\Z ()()) (#\W ()())))
```



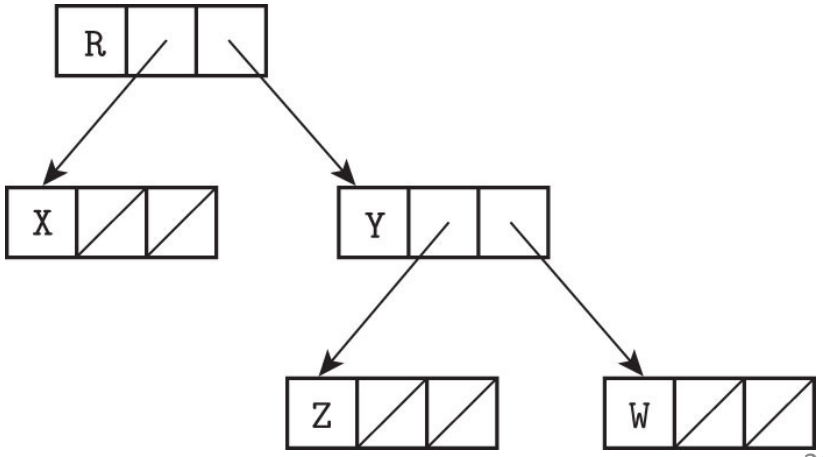
Pointers and Recursive Types

- Value model example: Tree in C

```
struct chr_tree {
   struct chr_tree *left, *right;
   char val;
};
my_ptr = malloc(sizeof(struct chr_tree));
```

- C++, Java, C# – type safe

```
     my_ptr = new chr_tree(arg_list);
(*my_ptr).val = 'X';
my_ptr->val = 'X';
```



Pointers and Recursive Types

- *Dangling reference*
  - live pointer that no longer points to a valid object
  - Example: caused by local variable after subroutine return:

```
int i = 3;
int *p = &i;
...
void foo(){ int n = 5; p = &n; }
...
cout << *p;        // prints 3
foo();
...
cout << *p; // undefined behavior: n is no
                                longer live
```

Pointers and Recursive Types

- Dangling reference
  - Example: caused by manual deallocation:

```
int *p = new int;
*p = 3;
...
cout << *p;      // prints 3
delete p;
...
cout << *p; // undefined behavior: *p has been
                                  reclaimed
```

- Problem: a dangling reference can write to memory that is part of a different object; it may even interfere with bookkeeping, corrupting the stack or heap

Garbage collection

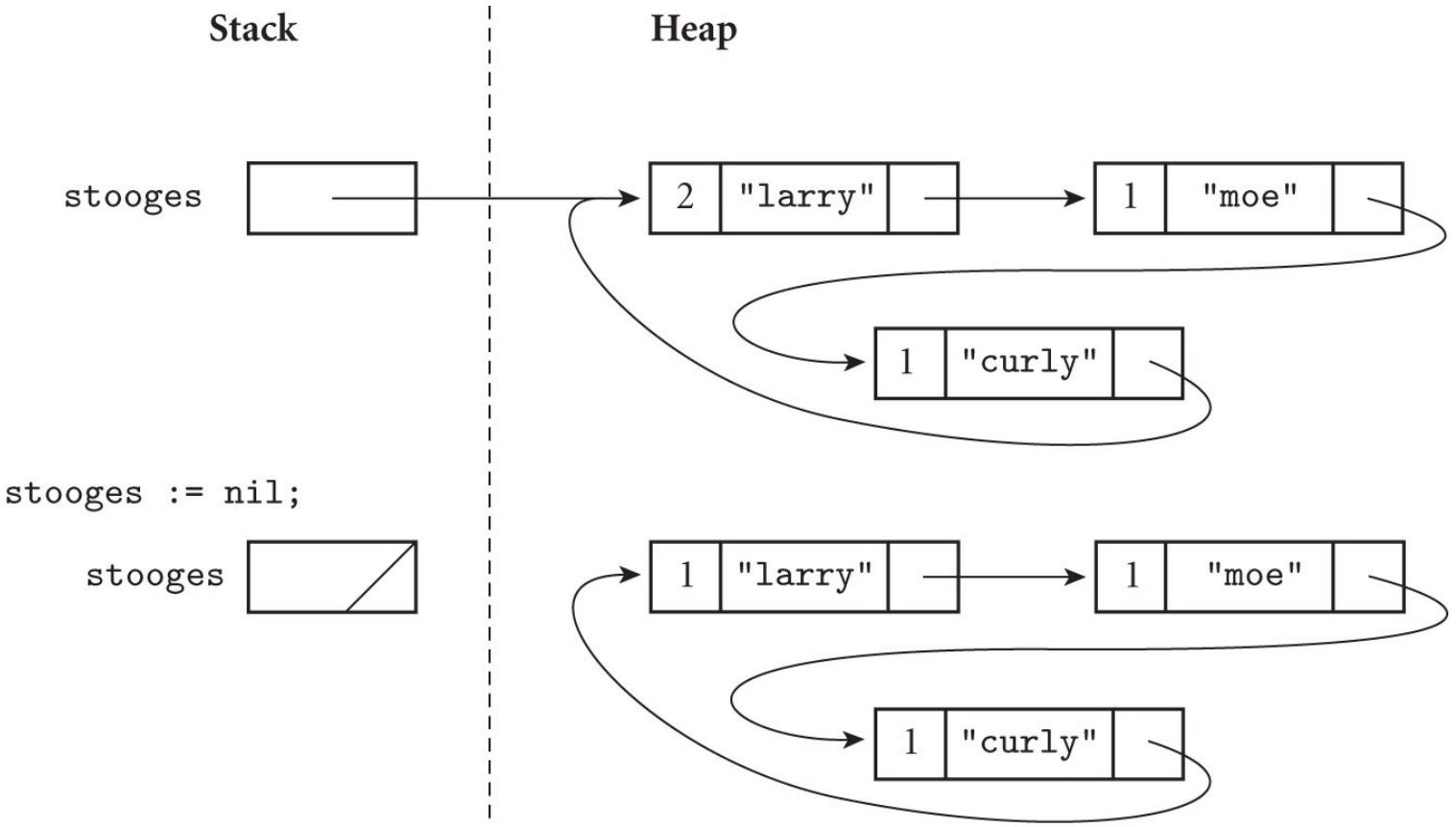
- *Garbage collection*
  - automatic reclamation of memory
  - slower than manual (`delete`)
  - difficult to implement
  - eliminates the need to check for dangling references
  - very convenient for programmers
  - essential for functional languages
  - increasingly popular in imperative languages
    - Java, C#

Garbage collection

- *Reference counts*
  - object no longer useful when no pointers to it exist
  - store *reference count* for each object
    - initially set to 1
    - update when assigning pointers
    - update on subroutine return
    - when 0, reclaim object

Garbage collection

- Reference counts
  - count ≠ 0 does not necessarily mean useful (circular lists)



Garbage collection

- Smart pointers in C++
- `unique_ptr`
  - one object only
- `shared_ptr`
  - implements a reference count
- `weak_ptr`
  - does not affect counts; for, e.g., circular structures

Garbage collection

- *Tracing collection*
  - object useful if reachable via chain of valid pointers from outside the heap
  - *Mark-and-sweep*

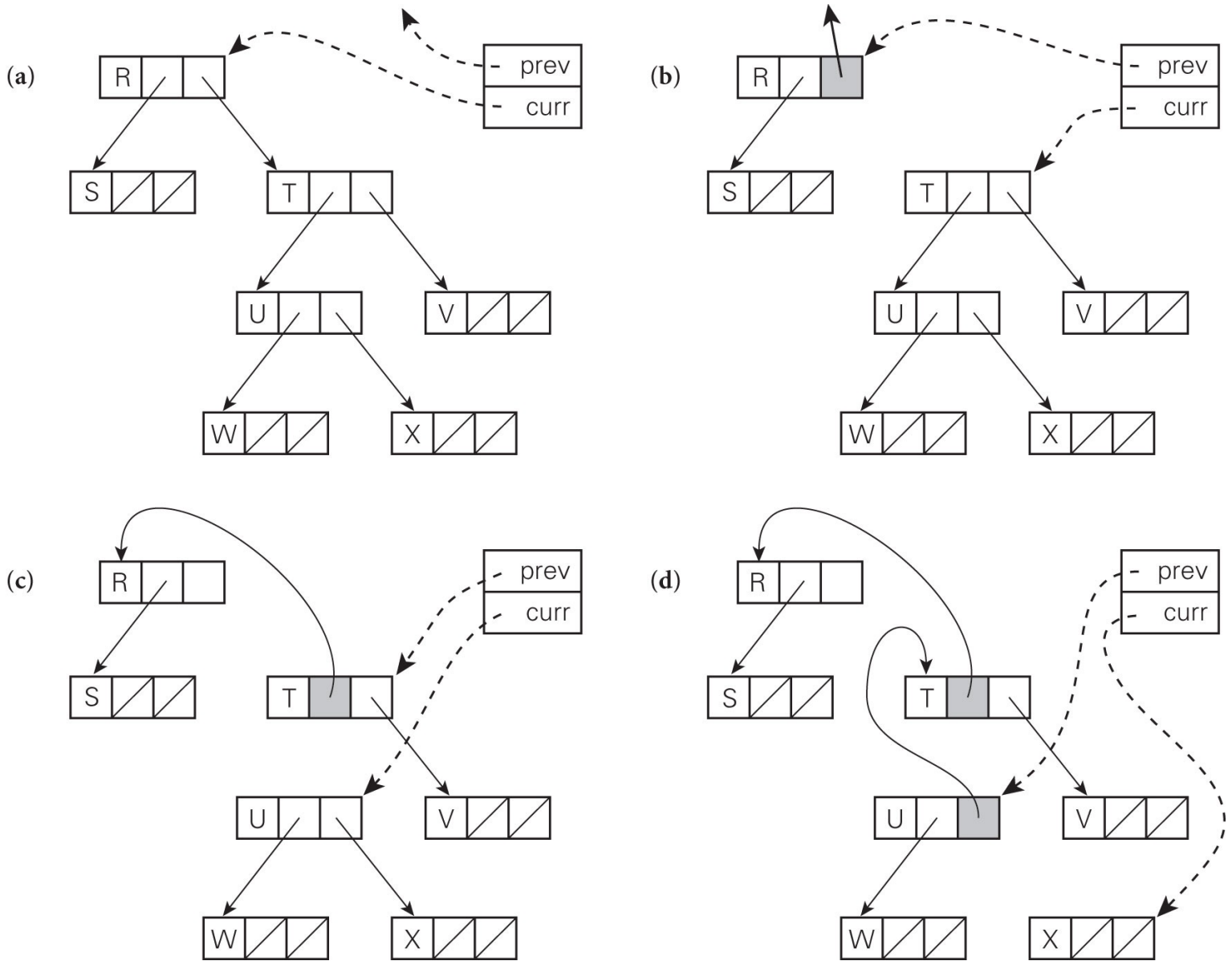
    (1) mark entire heap as “useless”

    (2) staring from outside heap, recursively mark as “useful”

    (3) move “useless” block from heap to free list

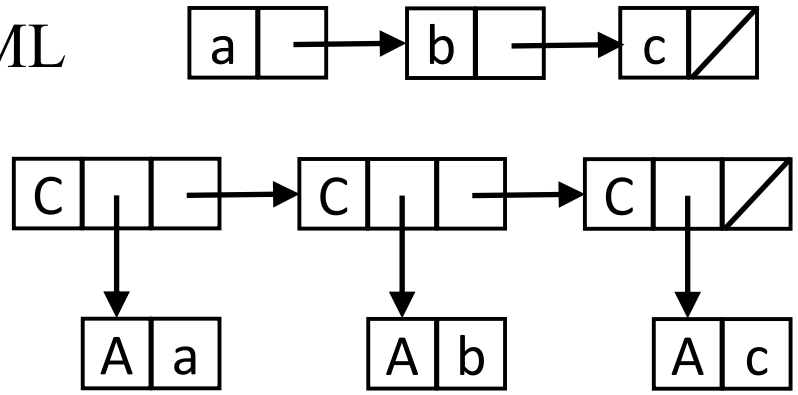
  - Step (2) requires a potentially very large stack
  - Without stack: *pointer reversal* (next slide)
- *Stop-and-copy*: defragmentation
  - use half of heap; copy useful data compactly to the other one

Pointer reversal



Lists

- *List*: empty or (head + tail)
- essential in functional and logic programming (recursive)
- used also in imperative languages
- *Homogeneous* (same type): ML
- *Heterogeneous*: Scheme



Lists

- Scheme:
  - `'(...)` prevents evaluation; also `(quote (...))`

```
(+ 1 2) ⟹ 3
'(+ 1 2) ⟹ '(+ 1 2)
```

```
(cons 'a '(b))          ⟹ '(a b)
(car '(a b))            ⟹ a
(car '())               ⟹ error
(cdr '(a b c))          ⟹ '(b c)
(cdr '(a))              ⟹ '()
(cdr '())               ⟹ error
(append '(a b) '(c d)) ⟹ '(a b c d)
```

Lists

- *List comprehension*
  - adapted from traditional math set notation:

    $\{i \times i \mid i \in \{1, \ldots, 10\} \wedge i \bmod 2 = 1\}$

  - Example: Python

```
[i*i for i in range(1, 10) if i % 2 == 1]
```

$\Longrightarrow$ `[1, 9, 25, 49, 81]`

Object-Oriented Programming

Chapter 10

Object-Oriented Programming

- Key elements:
  - Data hiding / Encapsulation
  - Inheritance
  - Dynamic method binding

Data hiding

- Data abstraction: control large software complexity
- *Data hiding*:
  - objects visible only where necessary
  - reduce cognitive load on programmer
  - global variables – no hiding
  - local variables – subroutines only but limited life
  - static variables – retained between invocations
  - modules as abstractions – *encapsulation*
    - subroutines, variables, types, etc. visible only inside module
    - *export* / *import* types
    - Java: `package`, C++: `namespace`
  - modules as types: the module *is* the type

Classes

- Class:
  - module as type
  - + inheritance
  - + dynamic method binding
- Object
  - instance of a class
  - object-oriented programming

Classes: Example

```
class list_err {                                    // exception
public:
    const char *description;
    list_err(const char *s) {description = s;}
};

class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;                                        // the actual data in a node
    list_node() {                                   // constructor
        prev = next = head_node = this;             // point to self
        val = 0;                                    // default value
    }
    list_node* predecessor() {
        if (prev == this || prev == head_node) return nullptr;
        return prev;
    }
    list_node* successor() {
        if (next == this || next == head_node) return nullptr;
        return next;
    }
```

Classes: Example (cont'd)

```
    bool singleton() {
        return (prev == this);
    }
    void insert_before(list_node* new_node) {
        if (!new_node->singleton())
            throw new list_err("attempt to insert node already on list");
        prev->next = new_node;
        new_node->prev = prev;
        new_node->next = this;
        prev = new_node;
        new_node->head_node = head_node;
    }
    void remove() {
        if (singleton())
            throw new list_err("attempt to remove node not currently on list");
        prev->next = next;
        next->prev = prev;
        prev = next = head_node = this;     // point to self
    }
    ~list_node() {                          // destructor
        if (!singleton())
            throw new list_err("attempt to delete node still on list");
    }
};
```

Classes: Example (cont'd)

```
class list {
    list_node header;
public:
    // no explicit constructor required;
    // implicit construction of 'header' suffices
    int empty() {
        return header.singleton();
    }
    list_node* head() {
        return header.successor();
    }
    void append(list_node *new_node) {
        header.insert_before(new_node);
    }
    ~list() {                    // destructor
        if (!header.singleton())
            throw new list_err("attempt to delete nonempty list");
    }
};
```

- create an empty list:

```
list* my_list_ptr = new list
```

Classes

- Data members – *fields*:
  - `prev, next, head_node, val`
- Subroutine members – *methods*:
  - `predecessor, successor, insert_before, remove`
- Accessing current object:
  - `this` (C++), `self` (Objective-C), `current` (Eiffel)
- Object creation / destruction:
  - *constructors*: `list_node()` (same name as the class)
  - *destructors* (C++): `~list_node()`

Visibility

- `public`: visible to users
- `private`: invisible to users
- C++: what is not public is private

Inheritance

- Derived class – *inherits* base class's fields and methods

```
class queue : public list {       // queue derived from list
public:
    // no specialized constructor/destructor required
    void enqueue(int v) {
        append(new list_node(v));       // append inherited
    }[

    int dequeue()
        if (empty())
            throw new list_err("dequeue from empty queue");
        list_node* p = head();            // head inherited
        p->remove();
        int v = p->val;
        delete p;
        return v;
    }
};
```

Inheritance

- `queue`: *derived class, child class, subclass*
- `list`: *base class, parent class, superclass*
- public members of the base class are always visible to methods of the derived class
- public members of the base class are visible to users only if the class is publicly derived
- we can hide public members by `private` derivation
  - exceptions made with `using`

```
class queue : private list { ...
public:
    using list::empty;
```

Inheritance

- the opposite is also possible with `delete`:

```
class queue : public list { ...
      ...
      void append(list_node *new_node) = delete;
```

- C++ `protected`
  - visible to members of its class and classes derived from it

```
class derived : protected base { ...
```

Visibility – C++ rules

- Any class can limit visibility of its members:

member	class's methods	class's and descendant's methods	anywhere (class scope)
public	✓	✓	✓
protected	✓	✓	✗
private	✓	✗	✗

- A derived class can restrict visibility of base class members but can never increase it:
  - Exceptions: `using`, `delete`

member \ derived class	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Visibility

- Java, C#
  - `private, protected, public`
  - no `protected` or `private` derivation
  - derived class can neither increase nor restrict visibility
  - can hide a field or override a method by defining a new one with the same name
    - cannot be more restrictive than the base class version
  - Java `protected`: visible in the entire package

  - `static` fields and methods
    - orthogonal to the visibility by public/protected/private
    - belong to the class as a whole: *class* fields and methods

Generics

- Previous `list` has integers only
- *Generics* allow list of any type
  - C++: *templates*

```
template<typename V>
class list_node {
    list_node<V>* prev;
    list_node<V>* next;
    list_node<V>* head_node;
public:
    V val;
    list_node<V>* predecessor() { ...
    list_node<V>* successor() { ...
    void insert_before(list_node<V>* new_node) { ...
    ...
};
```

Generics

```
template<typename V>
class list {
    list_node<V> header;
public:
    list_node<V>* head() { ...
    void append(list_node<V> *new_node) { ...
    ...
};

template<typename V>
class queue : private list<V> {
    list_node<V> header;
public:
    using list<V>::empty;
    void enqueue(const V v) { ...
    V dequeue() { ...
    V head() { ...
};
```

Generics

```
typedef list_node<int> int_list_node;
typedef list_node<string> string_list_node;
typedef list<int> int_list;
...
int_list_node n(3);
string_list_node s("boo!");
int_list L;
L.append(&n);        // ok
L.append(&s);        // error
```

Initialization and Finalization

- Initialize – *Constructor*
- Choosing a constructor
  - Can specify several constructors – C++, Java, C#
  - overloading: differentiate by number and types of parameters

```
class list_node {
    ...
    list_node(int v) {
        prev = next = head_node = this;
        val = v;
    }
...
list_node element1(1);      // int val
list_node *e_ptr = new list_node(5) // heap
list_node element0();       // default; val=0
```

Initialization and Finalization

- References and Values
  - Python, Java: variables refer to objects
    - every object is created explicitly
  - C++: variable has an object as value
    - objects created explicitly or implicitly, as result of elaboration
    - C++ requires all objects initialized by constructors

```
foo b;       // calls 0-arg constructor foo::foo()
foo b(10, 'x'); // calls foo::foo(int, char)

foo a;
foo b(a);  // calls copy constructor foo::foo(foo&)
foo b = a; // same thing ('=' is not assignment)

foo a, b;  // calls foo::foo() twice
b = a; // assignment; calls foo::operator=(foo&)
```

Initialization and Finalization

- Execution order for constructors (C++)
  - base class constructor executed first
  - also constructors of member classes
  - can specify arguments in constructor's header

```
class foo : bar {
    mem1_t member1;      // mem1_t and
    mem2_t member2;      // mem2_t are classes
    ...
}
foo::foo (foo_param) : bar (bar_args),
    member1 (mem1_init_val), member2 (mem2_init_val) {
    ...
```

Initialization and Finalization

- Finalize – *Destructor*
  - destructor of derived class called first, then base
  - C++: used for storage reclamation (manual storage)
  - Example: queue derived from list
    - default destructor calls `~list` (throws exception if non-empty)
  - If we wish destruction of non-empty queue:

```
~queue() {
    while (!empty()) {
        list_node* p = contents.head();
        p->remove();
        delete p;
    }
}      // or
~queue() {
    while (!empty()) {
        int v = dequeue();
    }
}
```

Dynamic Method Binding

- Subtype
  - Class D derived from C such that D doesn't hide any publicly visible member of C
  - a D-object can be used anywhere a C-object is expected
  - derived class is a *subtype* of base class

```
class person { ...
class student : public person { ...
class professor : public person { ...
...
student s;
professor p;
...
person *x = &s;
person *y = &p;
```

Dynamic Method Binding

- Polymorphic subroutine

```
class person { ...
void person::print_label { ...
...
s.print_label(); // print_label(s)
p.print_label(); // print_label(p)
```

- What if we redefine `print_label` in the derived classes?

```
s.print_label(); // student::print_label(s)
p.print_label(); // professor::print_label(p)
```

Dynamic Method Binding

- What about this?

```
x->print_label(); // ??
y->print_label(); // ??
```

- *Static method binding*: use the types of the variables `x` and `y`
- *Dynamic method binding*: use the classes of objects `s` and `p` to which the variables refer
- Example:
  - list of students and professors
  - print label correctly for each – dynamic method binding
  - derived class definition *overrides* the base class definition

Dynamic Method Binding

- Dynamic method binding
  - run-time overhead
  - Python, Objective-C, Ruby, Smalltalk – all methods
  - Java, Eiffel – dynamic default
    - `final` (Java) or `frozen` (Eiffel) cannot be overridden
  - C++, C#, Ada95, Simula – static default
    - static: *redefining* method
    - dynamic: *overriding* method – `virtual`

```
class person {
public:
    virtual void print_label();
    ...
```

Dynamic Method Binding

- Abstract classes
  - may omit the body of virtual functions – *abstract method*

```
abstract class person {     // Java, C#
    ...
    public abstract void print_label();
    ...
class person {                  // C++
    ...
public:
    virtual void print_label() = 0;
    ...
```

  - C++ – abstract method is called *pure virtual method*
  - *Abstract class* – has at least one abstract method
    - base for *concrete* classes
  - *Interface* – Java, C#
    - classes with abstract methods only

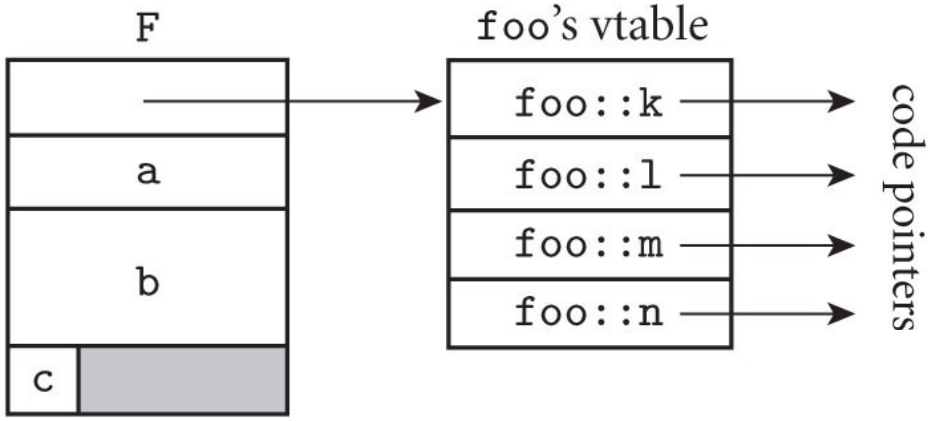
Dynamic member lookup

- Static method binding
  - the compiler knows which version of the method to call
- Dynamic method binding
  - reference variable must contain sufficient information for the code generated by compiler to find version at run time
- *Virtual method table* (*vtable*)
  - object implemented as a record whose first field contains the address of the vtable for the object's class
  - i^{th} entry of the vtable is the address of the code for the object's i^{th} virtual method

Dynamic member lookup

- Example

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```



Dynamic member lookup

- Dynamic method binding run-time overhead
- Example – code to call `f->m()`:
  - `f` is a pointer to an object of class `foo`
  - `m` is the third method of class `foo`

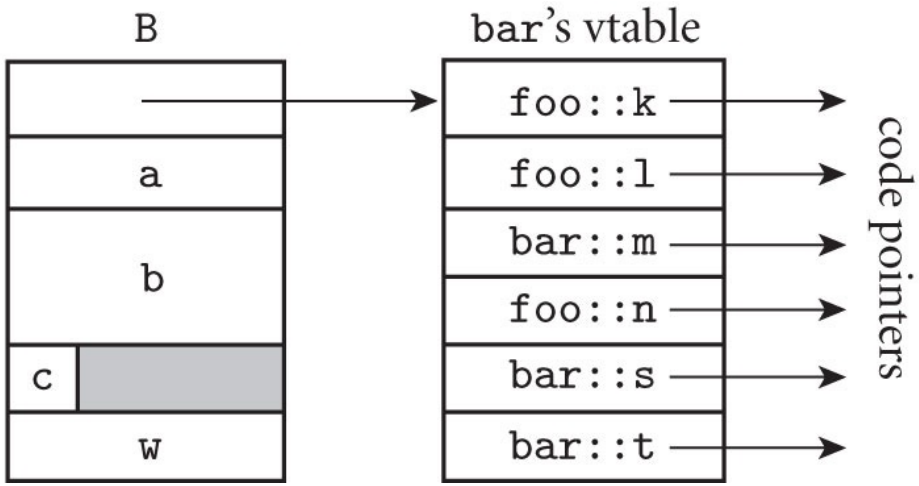
```
r1 := f
r2 := *r1          // vtable address
r2 := *(r2+(3–1)×4) // 4 = sizeof(address)
call *r2
```

  - this is two instructions longer than a call to statically identified method

Dynamic member lookup

- Inheritance

```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```



Dynamic member lookup

- Example:

```
class foo { ...
class bar : public foo { ...
...
foo F;
bar B;
foo* q;
bar* s;
...
q = &B;      // ok; uses a prefix of B's vtable
s = &F;      // static semantic error
s = dynamic_cast<bar*>(q);    // run-time check
s = (bar*)(q);              // permitted but risky
                            // no run-time check
```

λ-calculus

Chapter 11.7

What can be done by a computer?

- Algorithm formalization – 1930s
  - Church, Turing, Kleene, Post, etc.
  - *Church's thesis:*

    All intuitive computing models are equally powerful.
- Turing machine
  - automaton with an unbounded tape
  - *imperative programming*
- Church's λ-calculus
  - computes by substituting parameters into expressions
  - *functional programming*
- Logic: Horn clauses
  - collection of axioms to solve a goal
  - *logic programming*

λ-calculus

- *λ-calculus*
  - Church (1941) – to study computations with functions
  - *Everything is a function!*
- *λ-expressions* – defined recursively:
  - *name*: *x, y, z, u, v, …*
  - *abstraction*: $\lambda x.M$
    - function with parameter *x* and body *M*
  - *applications*: $M\ N$ – function *M* applied to *N*
- Examples
  - ($\lambda\ x.x*x$) - a function that maps *x* to *x*x*
  - ($\lambda\ x.x*x$) 4 - the same function applied to 4

λ-calculus

- Syntactic rules
  - application is left-associative

    $x\ y\ z$ means $(x\ y)\ z$
  - application has higher precedence than abstraction

    $\lambda x.A\ B$ means $\lambda x.(A\ B)$ (not $(\lambda x.A)\ B$)
  - consecutive abstractions:

    $\lambda x_1 x_2 \ldots x_n.e$ means $\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.e)\ldots))$

- Example:

  $\lambda xyz.x\ z\ (\mathrm{y}\ \mathrm{z}) = (\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z)))))$

λ-calculus

- Context-free grammars (CFG)

- CFG for λ-expressions

  expr → `name` | `(` λ `name` `.` *expr* `)` | `(` *expr* *expr* `)`

- CFG for λ-expressions with minimum parentheses

  expr → `name` | λ `name` `.` *expr* | *func* *arg*

  func → `name` | `(` λ `name` `.` *expr* `)` | *func* *arg*

  arg → `name` | `(` λ `name` `.` *expr* `)` | `(` *func* *arg* `)`

λ-calculus

- Examples

$\texttt{square} = \lambda x.\texttt{times}\ x\ x$

$\texttt{identity} = \lambda x.x$

$\texttt{const7} = \lambda x.7$

$\texttt{hypot} = \lambda x.\lambda y.\texttt{sqrt}\ (\texttt{plus}\ (\texttt{square}\ x)\ (\texttt{square}\ y))$

Free and bound variables

- $\lambda x.M$ - is a *binding* of the variable (or name) x
  - lexical scope
  - x is said to be *bound* in $\lambda x.M$
  - all x in $\lambda x.M$ are bound within the scope of this binding
- x is *free* in M if it is not bound
- $free(M)$ - the set of free variables in M
  - $free(x) = \{x\}$
  - $free(M\ N) = free(M) \cup free(N)$
  - $free(\lambda x.M) = free(M) - \{x\}$
- $bound(M)$ - the set of variables which are not free
  - any occurrence of a variable is free or bound; not both

Free and bound variables

- Example
  - x is free
  - y, z are bound

$$\lambda y.\lambda z.x\ z\ (y\ z)$$

Computing with pure λ-terms

- Computing idea:
  - reduce the terms into as simple a form as possible
  - $(\lambda x.M)\ N =_\beta \{N/x\}M$ – *substitute N for x in M*
  - the right-hand side is expected to be simpler
- Example:

$$(\lambda xy.x)\ u\ v =_\beta (\lambda y.u)\ v =_\beta u$$

Substitution

- $\{N/x\}M$ – *substitution* of term N for variable x in M
- Substitution rules (informal):
  - (i) if $free(N) \cap bound(M) = \emptyset$

    then just replace all free occurrences of x in M
  - (ii) otherwise, rename with fresh variables until (i) applies

Substitution rules

- In variables: the same or different variable
  - $\{N/x\}x = N$
  - $\{N/x\}y = y, y \neq x$
- In applications – the substitution distributes
  - $\{N/x\}(P\ Q) = \{N/x\}P\ \{N/x\}Q$
- In abstractions – several cases
  - no free x:

    $\{N/x\}(\lambda x.P) = \lambda x.P$
  - no interaction – y is not free in N:

    $\{N/x\}(\lambda y.P) = \lambda y.\{N/x\}P, \quad y \neq x, y \notin free(N)$
  - *renaming* – y is free in N; y is renamed to z in P:

    $\{N/x\}(\lambda y.P) = \lambda z.\{N/x\}\{z/y\}P,$

    $y \neq x, y \in free(N), z \notin free(N) \cup free(P)$

Computing with pure λ-terms

- Rewriting rules
- *α-conversion* – renaming the formal parameters

  $\lambda x.M \Longrightarrow_{\alpha} \lambda y.\{y/x\}M, y \notin free(M)$
- *β-reduction* – applying an abstraction to an argument

  $(\lambda x.M)\ N \Longrightarrow_{\beta} \{N/x\}\ M$

Equality of pure λ-terms

- Example

$(\lambda xyz.x\ z\ (y\ z))\ (\underline{\lambda x.x})\ (\lambda x.x)$

$\Longrightarrow_\alpha (\lambda xyz.x\ z\ (y\ z))\ (\lambda u.u)\ (\underline{\lambda x.x})$

$\Longrightarrow_\alpha (\underline{\lambda x}yz.x\ z\ (y\ z))\ (\underline{\lambda u.u})\ (\lambda v.v)$

$\Longrightarrow_\beta (\lambda yz.(\underline{\lambda u}.u)\ \underline{z}\ (y\ z))\ (\lambda v.v)$

$\Longrightarrow_\beta (\underline{\lambda y}z.z\ (y\ z))\ (\underline{\lambda v.v})$

$\Longrightarrow_\beta \lambda z.z\ ((\underline{\lambda v}.v)\ \underline{z})$

$\Longrightarrow_\beta \lambda z.z\ z$

Equality of pure λ-terms

- Example

$(\underline{\lambda fgh.f\ g\ (h\ h)})\ (\underline{\lambda xy.x})\ h\ (\lambda x.x\ x)$

$\Longrightarrow_\beta (\lambda g\underline{h}.(\lambda xy.x)\ g\ (\underline{h\ h}))\ h\ (\lambda x.x\ x)$

$\Longrightarrow_\alpha (\underline{\lambda g}k.(\lambda xy.x)\ g\ (k\ k))\ \underline{h}\ (\lambda x.x\ x)$

$\Longrightarrow_\beta (\underline{\lambda k}.(\lambda xy.x)\ h\ (k\ k))\ (\underline{\lambda x.x\ x})$

$\Longrightarrow_\beta (\underline{\lambda xy}.x)\ \underline{h}\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$

$\Longrightarrow_\beta (\underline{\lambda y}.h)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)})$

$\Longrightarrow_\beta h$

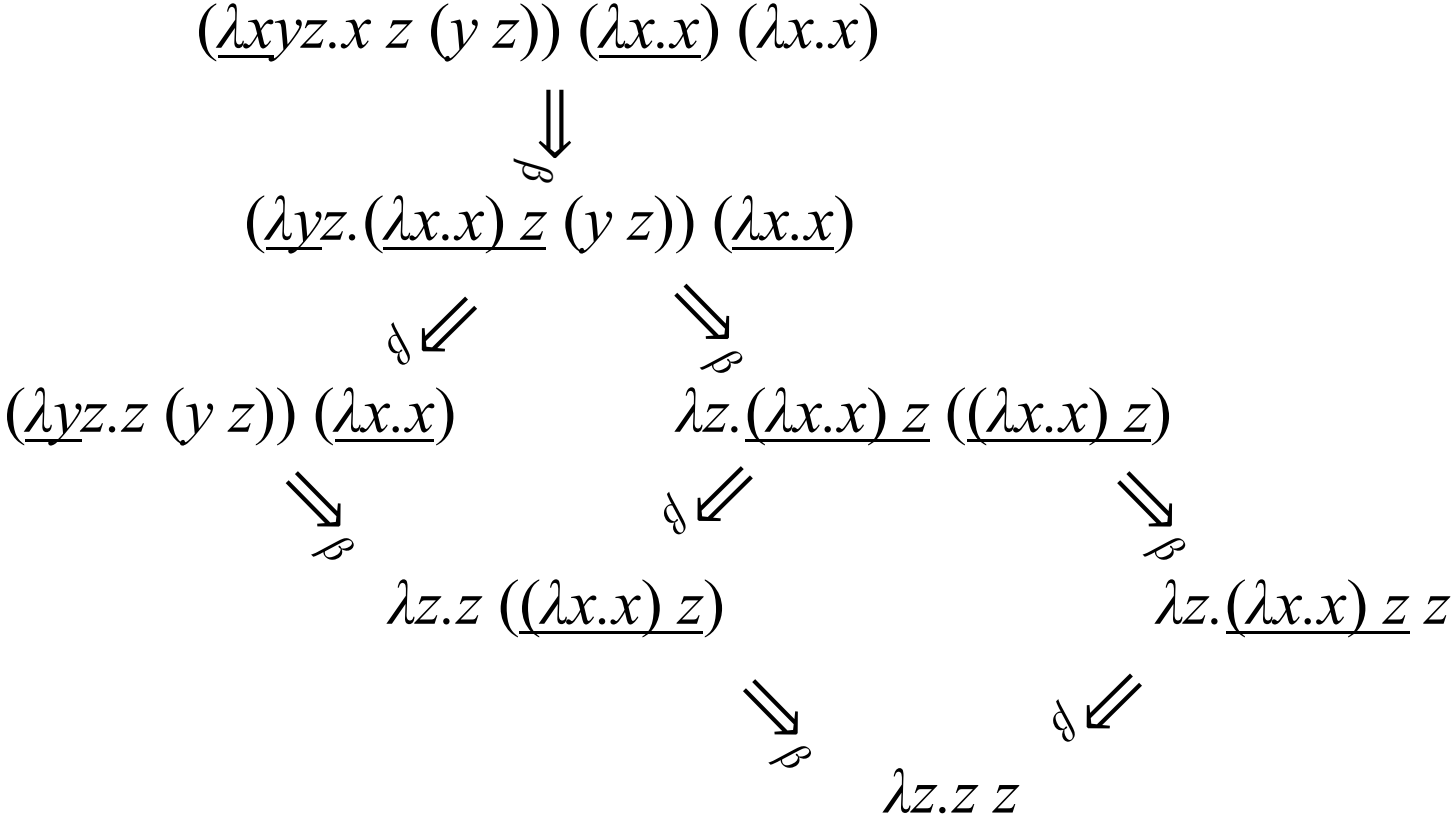
Computing with pure λ-terms

- Rewriting rules
- *Reduction*: any sequence of $\Longrightarrow_{\alpha}$, $\Longrightarrow_{\beta}$
- *Normal form*: term that cannot be β-reduced
  - β-normal form
  - Example of normal form

    $\lambda x.x\ x$ – cannot be reduced

Computing with pure λ-terms

- There may be several ways to reduce to a normal form
- Example: any path below is such a reduction



Computing with pure λ-terms

- *Nonterminating reductions*
  - Never reach a normal form
  - Example

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \Longrightarrow_\beta (\lambda x.x\ x)\ (\lambda x.x\ x)$$

Computing with pure λ-terms

- *Theorem* (*Church-Roser,* 1936)

For all pure λ-terms *M, P, Q*, if

$$M \Longrightarrow_\beta^* P \text{ and } M \Longrightarrow_\beta^* Q,$$

then there exists a term *R* such that

$$P \Longrightarrow_\beta^* R \text{ and } Q \Longrightarrow_\beta^* R.$$

- In particular, the normal form, when exists, is unique.

Computing with pure λ-terms

- Reduction strategies
- *Call-by-value reduction* (*applicative order*)
  - parameters are evaluated first, then passed
  - might not reach a normal form even if there is one
  - leftmost innermost lambda that can be applied
- Example

$$(\lambda y.h)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)})$$
$$\Longrightarrow_\beta (\lambda y.h)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)})$$
$$\Longrightarrow_\beta (\lambda y.h)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)})$$
$$\Longrightarrow_\beta \ldots$$

Computing with pure λ-terms

- Reduction strategies
- *Call-by-name reduction* (*normal order*)
  - parameters are passed unevaluated
  - leftmost outermost lambda that can be applied
- Example

  $(\lambda y.h)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)) \Longrightarrow_\beta h$

- *Theorem* (*Church-Roser,* 1936)

Normal order reduction reaches a normal form if there is one.

- Functional languages use also call-by-value because it can be implemented efficiently and it might reach the normal form faster than call-by-name.

λ-calculus can model everything

- Boolean values
- True: $\mathtt{T} \equiv \lambda x.\lambda y.x$
  - interpretation: of a pair of values, choose the first
- False: $\mathtt{F} \equiv \lambda x.\lambda y.y$
  - interpretation: of a pair of values, choose the second

- Properties:

$$((\mathtt{T}\ \ P)\ Q) \Longrightarrow_\beta (((\lambda x.\lambda y.x)\ P)\ Q) \Longrightarrow_\beta ((\lambda y.P)\ Q) \Longrightarrow_\beta P$$
$$((\mathtt{F}\ \ P)\ Q) \Longrightarrow_\beta (((\lambda x.\lambda y.y)\ P)\ Q) \Longrightarrow_\beta ((\lambda y.y)\ Q) \Longrightarrow_\beta Q$$

λ-calculus can model everything

- Boolean functions
- $\texttt{not} \equiv \lambda x.((x\ \texttt{F})\ \texttt{T})$
- $\texttt{and} \equiv \lambda x.\lambda y.((x\ y)\ \texttt{F})$
- $\texttt{or} \equiv \lambda x.\lambda y.((x\ \texttt{T})\ y)$

- Interpretation is consistent with predicate logic:

$$\texttt{not T} \Longrightarrow_\beta (\lambda x.((x\ \texttt{F})\ \texttt{T}))\ \texttt{T} \Longrightarrow_\beta ((\texttt{T}\ \texttt{F})\ \texttt{T}) \Longrightarrow_\beta \texttt{F}$$

$$\texttt{not F} \Longrightarrow_\beta (\lambda x.((x\ \texttt{F})\ \texttt{T}))\ \texttt{F} \Longrightarrow_\beta ((\texttt{F}\ \texttt{F})\ \texttt{T}) \Longrightarrow_\beta \texttt{T}$$

λ-calculus can model everything

- Integers

$\mathtt{0} \equiv \lambda f.\lambda c.c$

$\mathtt{1} \equiv \lambda f.\lambda c.(f\ c)$

$\mathtt{2} \equiv \lambda f.\lambda c.(f\ (f\ c))$

$\mathtt{3} \equiv \lambda f.\lambda c.(f\ (f\ (f\ c)))$

…

$\mathtt{N} \equiv \lambda f.\lambda c.(\underbrace{f\ (f\ \ldots(f}_{\mathtt{N}}\ c))\ldots)$

- Interpretation:
  - c is the zero element
  - f is the successor function

λ-calculus can model everything

- Integers (cont'd)
- Example calculations:

$$(\mathtt{N}\ a) = (\lambda f.\lambda c.(\underbrace{f\ \ldots(f}_{\mathtt{N}}\ c))\ldots))\ a \Longrightarrow_\beta \lambda c.(\underbrace{a\ldots(a}_{\mathtt{N}}\ c)\ldots)$$

$$((\mathtt{N}\ a)\ b) = (\underbrace{a\ (a\ldots(a}_{\mathtt{N}}\ b))\ldots)$$

λ-calculus can model everything

- Integer operations
- *Addition*: $+ \equiv \lambda M.\lambda N.\lambda a.\lambda b.((M\ a)((N\ a)\ b))$

  $[\mathtt{M} + \mathtt{N}] = \lambda a.\lambda b.((M\ a)\ ((N\ a)\ b)) \Longrightarrow_\beta^* \lambda a.\lambda b.\underbrace{(a\ (a...(a}_{\mathtt{M+N}}\ b))...)$

- *Multiplication*: $\times \equiv \lambda M.\lambda N.\lambda a.(M\ (N\ a))$

  $[\mathtt{M} \times \mathtt{N}] = \lambda a.(M\ (N\ a)) \Longrightarrow_\beta^* \lambda a.\lambda b.\underbrace{(a\ (a...(a}_{\mathtt{M\times N}}\ b))...)$

- *Exponentiation*: $\wedge \equiv \lambda M.\lambda N.(N\ M)$

  $[\mathtt{M}^{\mathtt{N}}] = (\mathtt{N}\ \ \mathtt{M}) \Longrightarrow_\beta^* \lambda a.\lambda b.(\underbrace{a\ (a...(a}_{\mathtt{M}^{\mathtt{N}}}\ b))...)$

- This way we can develop all computable math. functions.

λ-calculus can model everything

- Control flow
- `if` $\equiv \lambda c.\lambda t.\lambda e.c\ t\ e$
  - Interpretation: c = condition, t = then, e = else

- `if T 3 4` $= (\lambda c.\lambda t.\lambda e.c\ t\ e)(\lambda x.\lambda y.x)$ `3 4`
  $\Longrightarrow_\beta^*$ $(\lambda t.\lambda e.t)$ `3 4`
  $\Longrightarrow_\beta^*$ `3`
- `if F 3 4` $= (\lambda c.\lambda t.\lambda e.c\ t\ e)(\lambda x.\lambda y.y)$ `3 4`
  $\Longrightarrow_\beta^*$ $(\lambda t.\lambda e.e)$ `3 4`
  $\Longrightarrow_\beta^*$ `4`

λ-calculus can model everything

- Recursion
- `gcd` = $\lambda a.\lambda b.$(`if` (`equal` a b) a (if (`greater` a b) (`gcd` (`minus` a b) b) (`gcd` (`minus` b a) a)))
- This is not a definition because `gcd` appears in both sides
  - If we substitute this, the definition only gets bigger
- To obtain a real definition, we rewrite using β-abstraction:

  `gcd` = ($\lambda g.\lambda a.\lambda b.$(`if` (`equal` a b) a (if (`greater` a b) (g (`minus` a b) b) (g (`minus` b a) a)))) `gcd`
- we obtain the equation

  `gcd` = f `gcd`, where

  $f = \lambda g.\lambda a.\lambda b.$(`if` (`equal` a b) a (if (`greater` a b) (g (`minus` a b) b) (g (`minus` b a) a)))
- `gcd` is a fixed point of f

λ-calculus can model everything

- Define the *fixed point combinator:*

  $\mathtt{Y} \equiv \lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x))$

- $\mathtt{Y}\ f$ is a fixed point of f
  - if the normal order evaluation of $\mathtt{Y}\ f$ terminates then $f\ (\mathtt{Y}\ f)$ and $\mathtt{Y}\ f$ will reduce to the same normal form
- We get then a good definition for gcd:

$\mathtt{gcd} \equiv \mathtt{Y}\ f = (\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x)))\ (\lambda g.\lambda a.\lambda b.(\mathtt{if}\ (\mathtt{equal}\ a\ b)\ a\ (\text{if}\ (\mathtt{greater}\ a\ b)\ (g\ (\mathtt{minus}\ a\ b)\ b)\ (g\ (\mathtt{minus}\ b\ a)\ a)))\ )$

λ-calculus can model everything

- Example

$\texttt{gcd}\ 2\ 4$

$\equiv\ \texttt{Y}\ f\ 2\ 4$

$\equiv ((\lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x)))\ f\ )\ 2\ 4$

$\Longrightarrow_{\beta} ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ 2\ 4$

$\equiv (f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))))\ 2\ 4$ denote $k\ \equiv\ \lambda x.f\ (x\ x)$

$\Longrightarrow_{\beta} (f\ (k\ k))\ 2\ 4$

$\equiv ((\lambda g.\lambda a.\lambda b.(\texttt{if}\ (=\ a\ b)\ a\ (\texttt{if}\ (>\ a\ b)\ (g\ (-\ a\ b)\ b)\ (g\ (-\ b\ a)\ a))))(k\ k))\ 2\ 4$

$\Longrightarrow_{\beta} (\lambda a.\lambda b.(\texttt{if}\ (=\ a\ b)\ a\ (\texttt{if}\ (>\ a\ b)\ ((k\ k)\ (-\ a\ b)\ b)\ ((k\ k)(-\ b\ a)\ a))))\ 2\ 4$

$\Longrightarrow_{\beta}^{*} \texttt{if}\ (=\ 2\ 4)\ 2\ (\texttt{if}\ (>\ 2\ 4)\ ((k\ k)\ (-\ 2\ 4)\ 4)\ ((k\ k)\ (-\ 4\ 2)\ 2))$

$\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (=\ 2\ 4\ )\ 2\ (\texttt{if}\ (>\ 2\ 4)\ ((k\ k)\ (-\ 2\ 4)\ 4)\ ((k\ k)\ (-\ 4\ 2)\ 2))$

λ-calculus can model everything

$\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (=\ 2\ 4\ )\ 2\ (\texttt{if}\ (>\ 2\ 4)\ ((k\ k)\ (-\ 2\ 4)\ 4)\ ((k\ k)\ (-\ 4\ 2)\ 2))$

$\Longrightarrow_\beta{}^* (=\ 2\ 4)\ 2\ (\texttt{if}\ (>\ 2\ 4)\ ((k\ k)\ (-\ 2\ 4)\ 4)\ ((k\ k)\ (-\ 4\ 2)\ 2))$

$\Longrightarrow_\delta \texttt{F}\ 2\ (\texttt{if}\ (>\ 2\ 4)\ ((k\ k)\ (-\ 2\ 4)\ 4)\ ((k\ k)\ (-\ 4\ 2)\ 2))$

$\equiv (\lambda x.\lambda y.y)\ 2\ (\texttt{if}\ (>\ 2\ 4)\ ((k\ k)\ (-\ 2\ 4)\ 4)\ ((k\ k)\ (-\ 4\ 2)\ 2))$

$\Longrightarrow_\beta{}^* \texttt{if}\ (>\ 2\ 4)\ ((k\ k)\ (-\ 2\ 4)\ 4)\ ((k\ k)\ (-\ 4\ 2)\ 2)$

$\Longrightarrow_\beta \ldots$

$\Longrightarrow_\beta (k\ k)\ (-\ 4\ 2)\ 2$

$\equiv ((\lambda x.f\ (x\ x))\ k)\ (-\ 4\ 2)\ 2$

$\Longrightarrow_\beta (f\ (k\ k))(-\ 4\ 2)\ 2$

$\equiv ((\lambda g.\lambda a.\lambda b.(\texttt{if}\ (=\ a\ b)\ a\ (\texttt{if}\ (>\ a\ b)\ (g\ (-\ a\ b)\ b)\ (g\ (-\ b\ a)\ a))))\ (k\ k))(-\ 4\ 2)\ 2$

$\Longrightarrow_\beta (\lambda a.\lambda b.(\texttt{if}\ (=\ a\ b)\ a\ (\texttt{if}\ (>\ a\ b)\ ((k\ k)\ (-\ a\ b)\ b)\ ((k\ k)\ (-\ b\ a)\ a))))(-\ 4\ 2)\ 2$

λ-calculus can model everything

$\Longrightarrow_\beta (\lambda a.\lambda b.(\texttt{if}\ (= a\ b)\ a\ (\texttt{if}\ (> a\ b)\ ((k\ k)\ (- a\ b)\ b)\ ((k\ k)\ (- b\ a)\ a))))(- 4\ 2)\ 2$

$\Longrightarrow_\beta^* \texttt{if}\ (= (- 4\ 2)\ 2)\ (- 4\ 2)\ (\texttt{if}\ (> (- 4\ 2)\ 2)\ ((k\ k)\ (- (- 4\ 2)\ 2)\ 2)\ ((k\ k)\ (- 2\ (- 4\ 2))\ (- 4\ 2)))$

$\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (= (- 4\ 2)\ 2)\ (- 4\ 2)\ (\texttt{if}\ (> (- 4\ 2)\ 2)\ ((k\ k)\ (- (- 4\ 2)\ 2)\ 2)\ ((k\ k)\ (- 2\ (- 4\ 2))\ (- 4\ 2)))$

$\Longrightarrow_\beta^* (= (- 4\ 2)\ 2)\ (- 4\ 2)\ (\texttt{if}\ (> (- 4\ 2)\ 2)\ ((k\ k)\ (- (- 4\ 2)\ 2)\ 2)\ ((k\ k)\ (- 2\ (- 4\ 2))\ (- 4\ 2)))$

$\Longrightarrow_\delta (= 2\ 2)(- 4\ 2)\ (\texttt{if}\ (> (- 4\ 2)\ 2)\ ((k\ k)(- (- 4\ 2)\ 2)\ 2)\ ((k\ k)\ (- 2\ (- 4\ 2))\ (- 4\ 2)))$

$\Longrightarrow_\delta \texttt{T}\ (- 4\ 2)\ (\texttt{if}\ (> (- 4\ 2)\ 2)\ ((k\ k)\ (- (- 4\ 2)\ 2)\ 2)\ ((k\ k)\ (- 2\ (- 4\ 2))\ (- 4\ 2)))$

$\equiv (\lambda x.\lambda y.x)\ (- 4\ 2)\ (\texttt{if}\ (> (- 4\ 2)\ 2)\ ((k\ k)(- (- 4\ 2)\ 2)\ 2)\ ((k\ k)\ (- 2\ (- 4\ 2))\ (- 4\ 2)))$

$\Longrightarrow_\beta^* (- 4\ 2)$

$\Longrightarrow_\delta 2$

λ-calculus can model everything

- Structures
- `select_first` ≡ $\lambda x.\lambda y.x$
- `select_second` ≡ $\lambda x.\lambda y.y$
- `cons` ≡ $\lambda a.\lambda d.\lambda x.x\ a\ d$
- `car` ≡ $\lambda l.l$ `select_first`
- `cdr` ≡ $\lambda l.l$ `select_second`
- `null?` ≡ $\lambda l.l\ (\lambda x.\lambda y.$`F`$)$

λ-calculus can model everything

$$\texttt{car}\ (\texttt{cons}\ A\ B)$$
$$\equiv (\lambda l.l\ \texttt{select\_first})\ (\texttt{cons}\ A\ B)$$
$$\Longrightarrow_{\beta} (\texttt{cons}\ A\ B)\ \texttt{select\_first}$$
$$\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \texttt{select\_first}$$
$$\Longrightarrow_{\beta}^{*} (\lambda x.x\ A\ B)\ \texttt{select\_first}$$
$$\Longrightarrow_{\beta} \texttt{select\_first}\ A\ B$$
$$\equiv (\lambda x.\lambda y.x)\ A\ B$$
$$\Longrightarrow_{\beta}^{*} A$$

λ-calculus can model everything

$$\texttt{cdr}\ (\texttt{cons}\ A\ B)$$
$$\equiv (\lambda l.l\ \texttt{select\_second})\ (\texttt{cons}\ A\ B)$$
$$\Longrightarrow_\beta (\texttt{cons}\ A\ B)\ \texttt{select\_second}$$
$$\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \texttt{select\_second}$$
$$\Longrightarrow_\beta^* (\lambda x.x\ A\ B)\ \texttt{select\_second}$$
$$\Longrightarrow_\beta \texttt{select\_second}\ A\ B$$
$$\equiv (\lambda x.\lambda y.x)\ A\ B$$
$$\Longrightarrow_\beta^* B$$

λ-calculus can model everything

$\texttt{null?}\ (\texttt{cons}\ A\ B)$

$\equiv (\lambda l.l\ (\lambda x.\lambda y.\texttt{select\_second}))\ (\texttt{cons}\ A\ B)$

$\Longrightarrow_{\beta} (\texttt{cons}\ A\ B)\ (\lambda x.\lambda y.\texttt{select\_second})$

$\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ (\lambda x.\lambda y.\texttt{select\_second})$

$\Longrightarrow_{\beta}^{*} (\lambda x.x\ A\ B)\ (\lambda x.\lambda y.\texttt{select\_second})$

$\Longrightarrow_{\beta} (\lambda x.\lambda y.\texttt{select\_second})\ A\ B$

$\Longrightarrow_{\beta}^{*} \texttt{select\_second}$

$\equiv \texttt{F}$

Functional Programming

Chapter 11

Functional Programming

- No side effects
  - output of a program is a mathematical function of the inputs
  - no internal state, no side effects
- Recursion and composition
  - effects achieved by applying functions: recursion, composition
- First-class functions:
  - can be passed as a parameter
  - can be returned from a subroutine
  - can be assigned in a variable
  - (more strictly) can be computed at run time

Functional Programming

- Polymorphism
  - Functions can be applied to general class of arguments
- Lists
  - Natural recursive definition
  - List = head + tail (list)
- Homogeneity
  - program is a list – can be manipulated the same as data
- Garbage collection
  - heap allocation for dynamically allocated data
  - unlimited extent

Functional vs Imperative

- Advantages
- No side effects
  - predictable behavior
- *Referential transparency*
  - Expressions are independent of evaluation order
- *Equational reasoning*
  - Expressions equivalent at some point in time are equivalent at *any* point in time

Functional vs Imperative

- Disadvantages
- *Trivial update problem*
  - Every result is a new object instead of a modification of an existing one
- Data structures different from lists more difficult to handle
  - multidimensional arrays
  - dictionaries
  - in-place mutation
- The trivial update problem is not an inherent weakness of functional programming
  - The implementation could detect whether an old version of a structure will never be used again and update in place

Scheme

- Originally developed in 1975
- Initially very small
- Now is a complete general-purpose language
- Still derived from a smalls set of key concepts

- Lexically scoped
- Functions are first class values
- Implicit storage management

Scheme vs λ-calculus

- Scheme syntax very similar with λ-calculus
- Examples:
  - λ-calculus

    $\lambda x.x$

    $(\lambda x.x * x)\ 4 \Longrightarrow_{\beta} 16$

  - Scheme

```
(lambda (x) x)
((lambda (x) (* x x)) 4)  ⟹  16
```

Scheme: Interpreter

- Interacting with the interpreter

```
"hello" ⇒ "hello"
42 ⇒ 42
22/7 ⇒ 3 1/7
3.1415 ⇒ 3.1415
+ ⇒ #<procedure:+>
(+ 5 3) ⇒ 8
'(+ 5 3) ⇒ (+ 5 3)
'(a b c d) ⇒ '(a b c d)
'(2 3) ⇒ '(2 3)
(2 3) ⇒ error; 2 is not procedure
```

Scheme: Elements

- Identifiers
  - cannot start with a character that may start a number:
    `digit`, `+`, `-`, `.`
  - case is important
- Numbers: integers: `-1234`; ratios: `1/2`; floating-point: `1.3`, `1e23`; complex numbers: `1.3 - 2.7i`
- List constants: `'(a b c d)`
- Empty list: `'()`
- Procedure applications: `(+ (* 3 5) 12)`
- Boolean values: `#t` (true), `#f` (false)
  - Any object different from `#f` is true

Scheme: Elements

- Vectors

```
#(this is a vector of symbols)
```

- Strings

```
"this is a string"
```

- Characters

```
#\a, #\b , #\c
```

- Comments:
  - `; ... end_of_line`
  - `#| ... |#`

Scheme: Functions

- Variable definitions

```
(define a 23) a  ⇒  23
```

- Function applications

```
(+ 20 10) ⇒ 30
(+ 1/4 6/3) ⇒ 9/4
(* (* 2/5 5/6) 3) ⇒ 1
```

Scheme: Functions

- Defining a function

```
(define (square x) (* x x))
(square 5) ⇒ 25
```

- Anonymous functions

```
(lambda (x) (* x x))
((lambda (x) (* x x)) 5) ⇒ 25
```

- Named functions

```
(define square (lambda (x) (* x x)))
(square 5) ⇒ 25
```

Scheme: Quoting

- (`quote` *obj*) or
- '*obj*
  - tells Scheme *not* to evaluate

```
(quote (1 2 3 4 5)) ⇒ (1 2 3 4 5)
(quote (+ 3 4)) ⇒ (+ 3 4)
(quote +) ⇒ +
+ ⇒ #<procedure:+>
'(1 2 3 4 5) ⇒ (1 2 3 4 5)
'(+ (* 3 10) 4) ⇒ (+ (* 3 10) 4)
'2 ⇒ 2                    ; unnecessary
2 ⇒ 2
'"hi" ⇒ "hi               ; unnecessary
"hi" ⇒ "hi"
```

Scheme: Lists

- (`car` *list*)
  - gives the first element
- (`cdr` *list*)
  - gives the list without the first element

```
(car '(a b c)) ⇒ a
(cdr '(a b c)) ⇒ (b c)
(car (cdr '(a b c))) ⇒ b
```

- (`cons` *list*)
  - constructs a list from an element and a list

```
(cons 'a '()) ⇒ (a)
(cons 'a (cons 'b (cons 'c '()))) ⇒ (a b c)
(cons 'a 'b) ⇒ (a . b)  ;improper list
```

Scheme: Lists

- (`list` obj_1 obj_2 ...)
  - constructs (proper) lists; arbitrarily many arguments

```
(list 'a 'b 'c) ⇒ (a b c)
(list) ⇒ ()
(list 'a '(b c)) ⇒ (a (b c))
```

- (`null?` *list*)
  - tests whether a list is empty

```
(null? ()) ⇒ #t
(null? '(a)) ⇒ #f
```

Scheme: Variable binding

- `(let ((`*var* *val*`)…)` exp_1 exp_2 `…)`
- each *var* is bound to the value of the corresponding *val*
- returns the value of the final expression
- the body of `let` is the sequence exp_1 exp_2 …
- each *var* is visible only within the body of `let`
- no order is implied for the evaluation of the expressions *val*

Scheme: Variable binding

```
(let ((x 2))                ;let x be 2 in ...
  (+ x 3)) ⇒ 5
(let ((x 2) (y 3))
  (+ x y)) ⇒ 5
(let ((a (* 4 4)))
  (+ a a)) ⇒ 32
(let ((f +) (x 2) ( y 3))
  (f x y)) ⇒ 5
(let ((+ *))
  (+ 2 5)) ⇒ 10
(+ 2 5) ⇒ 7 ; + unchanged outside previous let
```

Scheme: Variable binding

```
(let ((x 1))
  (let ((y (+ x 1)))          ;nested lets
    (+ y y))) ⇒ 4
(let ((x 1))
  (let ((x (+ x 1)))          ;new variable x
    (+ x x))) ⇒ 4
```

Scheme: Variable binding

```
(let ((x1 1))
  (let ((x2 (+ x1 1))) ; indices show bindings
    (+ x2 x2))) ⇒ 4
(let ((x1 1) (y1 10))
 (let ((x2 (+ y1 (* x1 1))))
  (+ x2 (- (let ((x3 (+ x2 y1)) (y2 (* y1 y1)))
   (- y2 x3)) y1)))) ⇒ 80
(let ((sum (lambda (ls)
             (if (null? ls)
                 0
                 (+ (car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5)))
```

Scheme: Variable binding

- `(let*` `((`*var* *val*`)...)` *exp*$_1$ *exp*$_2$ `...)`
- similar with `let`
- each *val* is within the scope of variables to its left
- the expressions *val* are evaluated from left to right

```
(let* ((x 10) (y (- x 4)))
  (* y y)) ⇒ 36

(let ((x 10) (y (- x 4)))
  (* y y))
```

Scheme: Variable binding

- (`letrec` ((*var* *val*)…) exp_1 exp_2 …)
- each *val* is within the scope of all variables
- no order is implied for the evaluation of the expressions *val*

```
(letrec ((sum (lambda (ls)
                (if (null? ls)
                    0
                    (+ (car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5))) ⇒ 15
```

- `let` – for independent variables
- `let*` – linear dependency among variables
- `letrec` – circular dependency among variables

Scheme: Variable binding

```
(letrec ((even?  (lambda (x)
                   (or (= x 0)
                       (odd?  (- x 1)))))
         (odd?  (lambda (x)
                  (and (not (= x 0))
                       (even?  (- x 1))))))
  (list (even? 132) (odd? 2)))  ⇒  '(#t, #f)
```

Scheme: Functions

- (`lambda` *formals* exp_1 exp_2 ...)
  - returns a function
- *formals* can be:
- A *proper list* of variables (var_1 ... var_n)
  - then exactly *n* parameters must be supplied, and each variable is bound to the corresponding parameter

```
((lambda (x y) (* x (+ x y))) 7 13) ⇒ 140
```

- A *single* variable *x* (not in a list): then *x* is bound to a list containing all actual parameters

```
((lambda x x) 1 2 3) ⇒ (1 2 3)
((lambda x (sum x)) 1 2 3 4) ⇒ 10
```

Scheme: Functions

- An *improper list* terminated with a variable, (*var*$_1$ ... *var*$_n$. *var*), then at least *n* parameters must be supplied and *var*$_1$ ... *var*$_n$ will be bound to the first *n* parameters and *var* will be bound to a list containing the remaining parameters

```
((lambda (x y . z) (list x y z)) 1 2 3 4)
    ⇒ (1 2 (3 4))
```

Scheme: Assignments

- (`set!` *var exp*)
  - assigns a new value to an existing variable
  - this is not a new name binding but new value binding to an existing name

```
(let ((x 3) (y 4))
  (set! x 10)
    (+ x y))  ⇒  14
```

Scheme: Assignments

```
(define quadratic-formula
  (lambda (a b c)
    (let ((root1 0) (root2 0) (minusb 0)
                              (radical 0) (divisor 0))
      (set! minusb (- 0 b))
      (set! radical (sqrt (- (* b b) (* 4 (* a c)))))
      (set! divisor (* 2 a))
      (set! root1 (/ (+ minusb radical) divisor))
      (set! root2 (/ (- minusb radical) divisor))
      (list root1 root2))))

(quadratic-formula 1 -3 2) ⇒ (2 1)
```

Scheme: Assignments

- Can be done without `set!`

```
(define quadratic-formula
  (lambda (a b c)
    (let ((minusb (- 0 b))
          (radical (sqrt (- (* b b) (* 4 (* a c)))))
          (divisor (* 2 a)))
      (let ((root1 (/ (+ minusb radical) divisor))
            (root2 (/ (- minusb radical) divisor)))
        (list root1 root2)))))

(quadratic-formula 1 -3 2) ⇒ (2 1)
```

Scheme: Assignments

- Cannot be done without `set!`
  - the following version of cons, `cons-new`, counts the number of times it is called in the variable cons-count

```
(define cons-count 0)
(define cons-new
      (let ((old-cons cons))
        (lambda (x y)
          (set! cons-count (+ cons-count 1))
          (old-cons x y))))
(cons-new 'a '(b c))
cons-count ⇒  1
(cons-new 'a (cons-new 'b (cons-new 'c '())))
cons-count ⇒  4
```

Scheme: Sequencing

- (`begin` exp_1 exp_2 ...)
  - exp_1 exp_2 ... are evaluated from left to right
  - used for operations causing side effects
  - returns the result of the last expression

Scheme: Sequencing

```
(define quadratic-form
 (lambda (a b c)
  (begin
   (define root1 0) (define root2 0)
   (define minusb 0) (define radical 0) (define
divisor 0) (set! minusb (- 0 b))
   (set! radical (sqrt (- (* b b) (* 4 (* a c)))))
   (set! divisor (* 2 a))
   (set! root1 (/ (+ minusb radical) divisor))
   (set! root2 (/ (- minusb radical) divisor))
   (list root1 root2))))
(quadratic-form 1 -3 2)  ⇒ '(2 1)
```

Scheme: Conditionals

- (`if` *test consequent alternative*)
  - returns the value of consequent or alternative depending on test

```
(define abs
  (lambda (x)
    (if (< x 0)
        (- 0 x)
        x)))
(abs 4) ⇒ 4
(abs -5) ⇒ 5
```

Scheme: Conditionals

- (`not` *obj*)
  - returns `#t` if obj is false and `#f` otherwise

```
(not #f) ⇒ #t
(not 'a) ⇒ #f
(not 0) ⇒ #f
```

Scheme: Conditionals

- (`and` *exp* ...)
  - evaluates its subexpressions from left to right and stops immediately if any expression evaluates to false
  - returns the value of the last expression evaluated

```
(and #f 4 6 'a) ⇒ #f
(and '(a b) 'a 2) ⇒ 2
(let ((x 5))
  (and (> x 2) (< x 4))) ⇒ #f
```

Scheme: Conditionals

- (`or` *exp* ...)
  - evaluates its subexpressions from left to right and stops immediately if any expression evaluates to true
  - returns the value of the last expression evaluated

```
(or #f 4 6 'a) ⇒ 4
(or '(a b) 'a 2) ⇒ (a b)
(let ((x 3))
  (or (< x 2) (> x 4))) ⇒ #f
```

Scheme: Conditionals

- (`cond` $clause_1$ $clause_2$ ...)
  - evaluates the test of each clause until one is found true or all are evaluated

```
(define memv
  (lambda (x ls)
(cond
  ((null?  ls) #f)
  ((eqv?  (car ls) x) ls)
  (else (memv x (cdr ls))))))
(memv 'a '(d a b c)) ⇒ '(a b c)
(memv 'a '(b b c)) ⇒ #f
```

Scheme: Recursion, iteration, mapping

- (`let` *name* ((*var* *val*)…) exp_1 exp_2 …)
  - this is named let
  - it is equivalent with

((`letrec` ((*name* (`lambda` (*var* …) exp_1 exp_2 …)))
name)
val …)

Scheme: Recursion, iteration, mapping

```
(define divisors
  (lambda (n)
    (let f ((i 2))
      (cond
        ((>= i n) '())
        ((integer?  (/ n i))
         (cons i (f (+ i 1))))
        (else (f (+ i 1)))))))
(divisors 5) ⇒ '()
(divisors 12) ⇒ '(2 3 4 6)
```

Scheme: Recursion, iteration, mapping

- (do ((*var val update*)...) (*test res* ...) *exp* ...)
  - variables *var*... are are initially bound to *val*... and rebound on each iteration to *update*...
  - stops when *test* is true and returns the value of the last *res*
  - when *test* is false, it evaluates *exp*..., then *update*...; new bindings for *var*... are created and iteration continues

Scheme: Recursion, iteration, mapping

```
(define factorial
  (lambda (n)
    (do ((i n (- i 1)) (a 1 (* a i)))
      ((zero?  i) a))))
(factorial 0) ⇒ 1
(factorial 1) ⇒ 1
(factorial 5) ⇒ 120
```

Scheme: Recursion, iteration, mapping

```
(define fibonacci
 (lambda (n)
  (if (= n 0) 1
   (do ((i n (- i 1))(a1 1 (+ a1 a2))(a2 0 a1))
                                    ((= i 0) a1)))))
(fibonacci 0) ⇒ 1
(fibonacci 1) ⇒ 1
(fibonacci 2) ⇒ 2
(fibonacci 3) ⇒ 3
(fibonacci 4) ⇒ 5
```

Scheme: Recursion, iteration, mapping

- (`map` *procedure* $list_1$ $list_2$ ...)
  - applies *procedure* to corresponding elements of the lists $list_1$ $list_2$ ... and returns the list of the resulting values
  - procedure must accept as many arguments as there are lists
  - the order is not specified

```
(map abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)
(map (lambda (x y) (* x y))
  '(1 2 3 4) '(5 6 7 8)) ⇒ (5 12 21 32)
```

Scheme: Recursion, iteration, mapping

- (`for-each` *procedure* $list_1$ $list_2$ ...)
  - similar to map
  - does not create and return a list
  - applications are from left to right

```
(let ((same-count 0))
  (for-each
   (lambda (x y)
     (if (= x y)
         (set! same-count (+ same-count 1))
         '()))
   '(1 2 3 4 5 6) '(2 3 3 4 7 6))
  same-count) ⇒ 3
```

Scheme: Pairs

- `cons` builds a pair (called also *dotted pair*)
- both proper and improper lists can be written in dotted notation
- a list is a chain of pairs ending in the empty list ()

- proper list: `cdr` of the last pair is the empty list
  - *x* is a proper list if there is *n* such that $\texttt{cdr}^n(\text{x}) = \texttt{'()}$
- improper list: `cdr` of the last pair is anything other than ()

```
(cons 'a '(b)) ⇒ '(a b)    ; proper
(cons 'a 'b) ⇒ '(a . b)    ; improper
(cdr (cdr (cdr '(a b c)))) ⇒ '()
(cdr (cdr '(a b . c))) ⇒ 'c
```

Scheme: Predicates

- (`boolean?` *obj*)
  - `#t` if *obj* is either `#t` or `#f`; `#f` otherwise
- (`pair?` *obj*)
  - `#t` if *obj* is a pair; `#f` otherwise

```
(pair? '(a b)) ⇒ #t
(pair? '( a . b)) ⇒ #t
(pair? 2) ⇒ #f
(pair? 'a) ⇒ #f
(pair? '(a)) ⇒ #t
(pair? '()) ⇒ #f
```

Scheme: Predicates

- (`char?` *obj*) - `#t` if *obj* is a character, else `#f`
- (`string?` *obj*) - `#t` if *obj* is a string, else `#f`
- (`number?` *obj*) - `#t` if *obj* is a number, else `#f`
- (`complex?` *obj*) - `#t` if *obj* is complex, else `#f`
- (`real?` *obj*) - `#t` if *obj* is a real number, else `#f`
- (`integer?` *obj*) - `#t` if *obj* is integer, else `#f`
- (`list?` *obj*) - `#t` if *obj* is a list, else `#f`
- (`vector?` *obj*) - `#t` if *obj* is a vector, else `#f`
- (`symbol?` *obj*) - `#t` if *obj* is a symbol, else `#f`
- (`procedure?` *obj*) - `#t` if *obj* is a function, else `#f`

Scheme: Input / Output

- (`read`)
  - returns the next object from input
- (`display` *obj*)
  - prints *obj*

```
(display "compute the square root of:")
  ⇒ compute the square root of: 2
(sqrt (read))
 ⇒ 1.4142135623730951
```

Scheme: Deep binding

```
(define A
 (lambda (i P)
  (let ((B (lambda () (display i) (newline))))
    (cond ((= i 4) (P))
          ((= i 3) (A (+ i 1) P))
          ((= i 2) (A (+ i 1) P))
          ((= i 1) (A (+ i 1) P))
          ((= i 0) (A (+ i 1) B))))))
(define C (lambda () 10))
(A 0 C) ⇒ 0
```

Scheme: Deep binding

```
(define A
 (lambda (i P)
  (let ((B (lambda () (display i) (newline))))
    (cond ((= i 4) (P))
          ((= i 3) (A (+ i 1) P))
          ((= i 2) (A (+ i 1) B))
          ((= i 1) (A (+ i 1) P))
          ((= i 0) (A (+ i 1) B))))))
(define C (lambda () 10))
(A 0 C) ⇒ 2
```

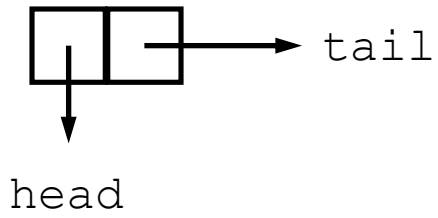
Scheme: Storage allocation for lists

- Lists are constructed with `list` and `cons`
  - `list` is a shorthand version of nested cons functions

```
(list 'apple 'orange 'grape)
      ⇒ '(apple orange grape)
(cons 'apple (cons 'orange (cons 'grape '())))
      ⇒ '(apple orange grape)
```

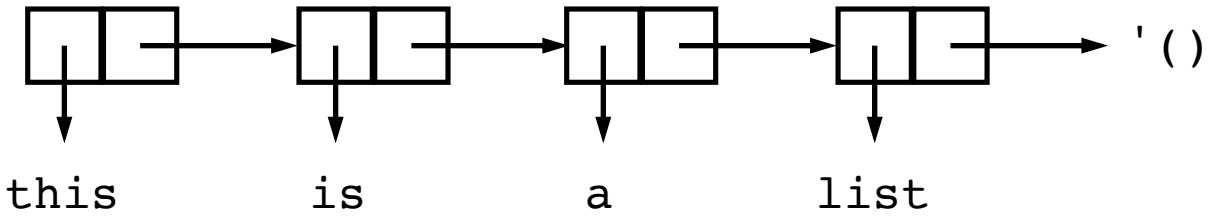
Scheme: Storage allocation for lists

- Memory allocation with `cons`
  - cell with pointers to head (`car`) and tail (`cdr`):



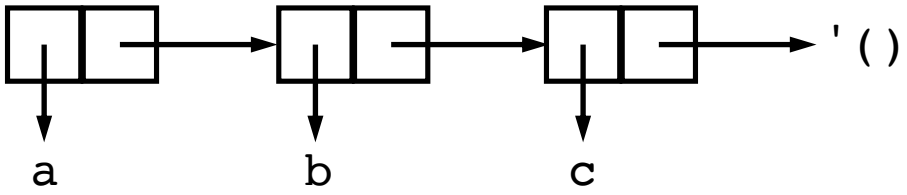
- Example

```
(cons 'this (cons 'is (cons 'a (cons 'list '()))))
```

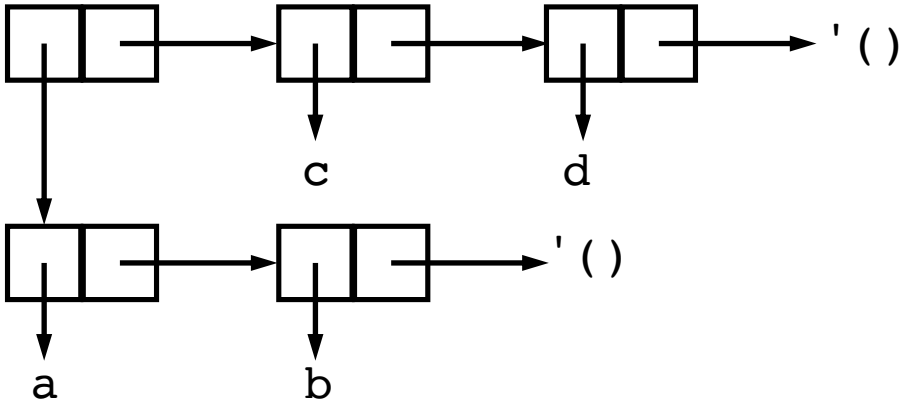


Scheme: Storage allocation for lists

```
(cons 'a '(b c)) ⇒ '(a b c)
```




```
(cons '(a b) '(c d)) ⇒ '((a b) c d)
```



Scheme: Equality

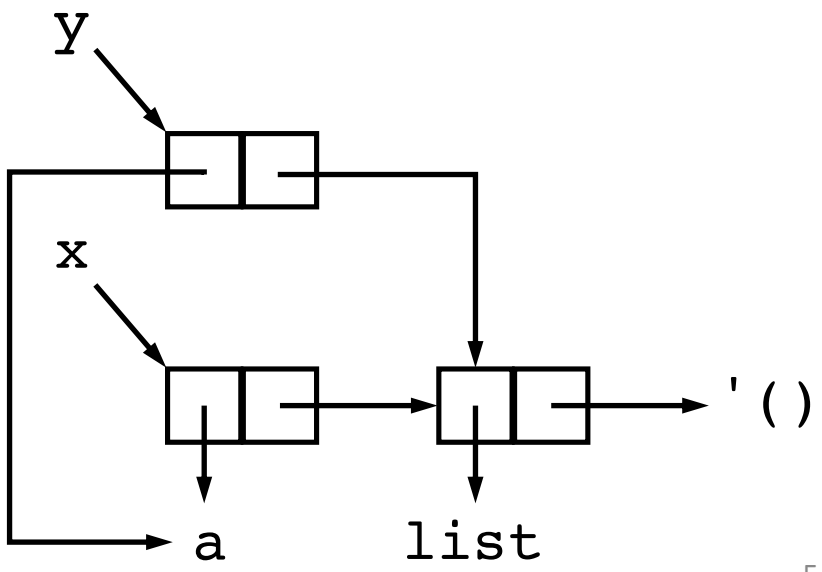
- (`eq?` obj_1 obj_2)
  - returns `#t` if obj_1 and obj_2 are identical, else `#f`
  - implementation as fast as possible
- (`eqv?` obj_1 obj_2)
  - returns `#t` if obj_1 and obj_2 are equivalent, else `#f`
  - similar to `eq?` but is guaranteed to return `#t` for two exact numbers, two inexact numbers, or two characters with the same value
- (`equal?` obj_1 obj_2)
  - returns `#t` if obj_1 and obj_2 have the same structure and contents, else `#f`

Scheme: Equality

```
(eq? 'a 3) ⇒ #f
(eqv? 'a 3) ⇒ #f
(equal? 'a 3) ⇒ #f

(eq? 'a 'a) ⇒ #t
(eqv? 'a 'a) ⇒ #t
(equal? 'a 'a) ⇒ #t

(eq? #t (null? '())) ⇒ #t
(eqv? #t (null? '())) ⇒ #t
(equal? #t (null? '())) ⇒ #t

(eq? 3.4 (+ 3.0 .4)) ⇒ #f
(eqv? 3.4 (+ 3.0 .4)) ⇒ #t
(equal? 3.4 (+ 3.0 .4)) ⇒ #t
```

Scheme: Equality

```
(eq? '(a) '(a)) ⇒ #f
(eqv? '(a) '(a)) ⇒ #f
(equal? '(a) '(a)) ⇒ #t

(define x ‘(a list))
(define y (cons (car x) (cdr x)))
(eq? x y) ⇒ #f
(eqv? x y) ⇒ #f
(equal? x y) ⇒ #t
```



Scheme: List searching

- (`memq` *obj list*)

  (`memv` *obj list*)

  (`member` *obj list*)
  - return the first tail of list whose car is equivalent to *obj* (in the sense of `eq?`, `eqv?`, or `equal?` resp.) or `#f`

```
(memq 'b '(a b c)) ⇒ '(b c)
```

Scheme: List searching

- (`assq` *obj list*)

  (`assv` *obj list*)

  (`assoc` *obj list*)
  - an *association list (alist)* is a proper list whose elements are key-value pairs (*key* `.` *value*)
  - return the first element of *alist* whose `car` is equivalent to *obj* (in the sense of `eq?`, `eqv?`, or `equal?` resp.) or `#f`

```
(assq 'b '((a . 1) (b . 2))) ⇒ '(b . 2)
(assq 'c '((a . 1) (b . 2))) ⇒ #f
(assq 2/3 '((1/3 . a) (2/3 . b))) ⇒ '(2/3 . b)
(assq 2/3 '((1/3 a) (2/3 b))) ⇒ '(2/3 b)
```

Scheme: Evaluation order

- λ-calculus:
  - applicative order (parameters evaluated before passed)
  - normal order (parameters passed unevaluated)
- Scheme uses applicative order
  - applicative may be faster
  - in general, either one can be faster

Scheme: Evaluation order

- Example: applicative order is faster

```
(double (* 3 4))
⇒ (double 12)
⇒ (+ 12 12)
⇒ 24

(double (* 3 4))
⇒ (+ (* 3 4) (* 3 4)) ⇒ (+ 12 (* 3 4))
⇒ (+ 12 12)
⇒ 24
```

Scheme: Evaluation order

- Example: normal order is faster

```
(define switch (lambda (x a b c)
  (cond ((< x 0) a)
        ((= x 0) b)
        ((> x 0) c))))

(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (switch -1 3 (+ 2 3) (+ 3 4))
⇒ (switch -1 3 5 (+ 3 4))
⇒ (switch -1 3 5 7)
⇒ (cond ((< -1 0) 3)
        ((= -1 0) 5)
        ((> -1 0) 7)
⇒ 3
```

Scheme: Evaluation order

- Example: normal order is faster (cont'd)

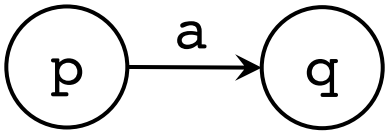
```
(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (cond ((< -1 0) (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4))
⇒ (cond (#t (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4))
⇒ (+ 1 2)
⇒ 3
```

Scheme: Higher-order functions

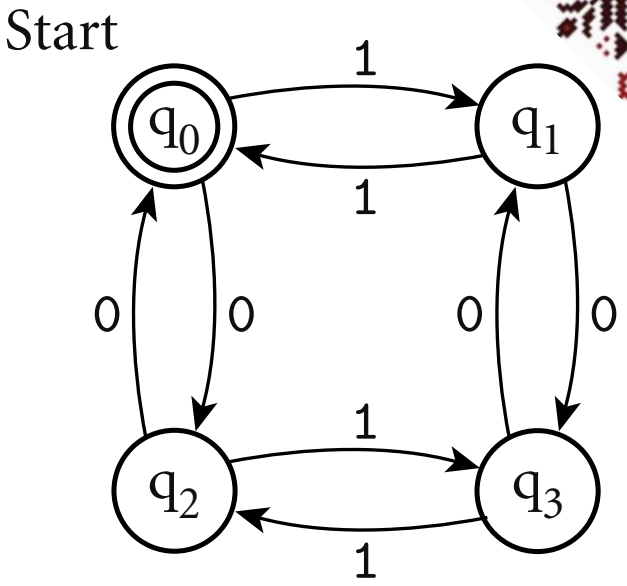
```
(define mcompose
 (lambda (flist)
  (lambda (x)
   (if (null? (cdr flist))
    ((car flist) x)
    ((car flist) ((mcompose (cdr flist)) x))))))

(define cadr
  (mcompose (list car cdr)))
(cadr '(a b c)) ⇒ ‘b

(define cadaddr
  (mcompose (list car cdr car cdr cdr)))
(cadaddr '(a b (c d))) ⇒ 'd
```

Scheme: DFA simulation

- DFA description:
  - start state
  - transitions: list of pairs
    - `((p a) q)`:



  - final states




```
(define zero-one-even-dfa
  '(q0                                    ; start state
    (((q0 0) q2) ((q0 1) q1)
      ((q1 0) q3) ((q1 1) q0)
      ((q2 0) q0) ((q2 1) q3)
      ((q3 0) q1) ((q3 1) q2))            ; transition fn
    (q0)))                                ; final states
```

Scheme: DFA simulation

- DFA simulation:

```
(simulate
 zero-one-even-dfa    ; machine description
 '(0 1 1 0 1))       ; input string
⇒  '(q0 q2 q3 q2 q0 q1 reject)

(simulate
 zero-one-even-dfa  ; machine description
 '(0 1 0 0 1 0))  ; input string
⇒  '(q0 q2 q3 q1 q3 q2 q0 accept)
```

Scheme: Differentiation

- Symbolic differentiation

$$\frac{d}{dx}(c) = \frac{d}{dx}(y) = 0, \qquad c \text{ a constant}, y \neq x$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u+v) = \frac{d}{dx}(u) + \frac{d}{dx}(v), \qquad u, v \text{ functions of } x$$

$$\frac{d}{dx}(u-v) = \frac{d}{dx}(u) - \frac{d}{dx}(v)$$

$$\frac{d}{dx}(uv) = u\frac{d}{dx}(v) + v\frac{d}{dx}(u)$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v\frac{d}{dx}(u) - u\frac{d}{dx}(v)}{v^2}$$

Scheme: Differentiation

```
(define diff
  (lambda (x expr)
    (if (not (pair? expr))
        (if (equal? x expr) 1 0)
        (let ((u (cadr expr))(v (caddr expr)))
          (case (car expr)
            ((+) (list '+ (diff x u) (diff x v)))
            ((-) (list '- (diff x u) (diff x v)))
            ((*) (list '+
                       (list '* u (diff x v))
                       (list '* v (diff x u))))
            ((/) (list '/ (list '-
                                (list '* v (diff x u))
                                (list '* u (diff x v)))
                       (list '* v v)))
            )))))
```

Scheme: Differentiation

```
(diff 'x '3) => 0
(diff 'x 'x) => 1
(diff 'x 'y) => 0
(diff 'x '(+ x 2)) => '(+ 1 0)
(diff 'x '(+ x y)) => '(+ 1 0)
(diff 'x '(* 2 x)) => '(+ (* 2 1) (* x 0))
(diff 'x '(/ 1 x)) => '(/ (- (* x 0) (* 1 1)) (* x x))
(diff 'x '(+ (* 2 x) 1)) => '(+ (+ (* 2 1) (* x 0)) 0)
(diff 'x '(/ x (- (* 2 x) (* 1 x))))
=> '(/
      (- (* (- (* 2 x) (* 1 x)) 1) (* x (- (+ (* 2 1)
(* x 0)) (+ (* 1 1) (* x 0)))))
      (* (- (* 2 x) (* 1 x)) (- (* 2 x) (* 1 x))))
```

Functional Languages

11.7 Theoretical Foundations

EXAMPLE 11.77
Functions as mappings

Mathematically, a function is a single-valued mapping: it associates every element in one set (the *domain*) with (at most) one element in another set (the *range*). In conventional notation, we indicate the domain and range by writing

$$\text{sqrt} : \mathcal{R} \longrightarrow \mathcal{R}$$

We can, of course, have functions of more than one variable—that is, functions whose domains are Cartesian products:

$$\text{plus} : [\mathcal{R} \times \mathcal{R}] \longrightarrow \mathcal{R}$$

■

If a function provides a mapping for every element of the domain, the function is said to be *total*. Otherwise, it is said to be *partial*. Our *sqrt* function is partial: it does not provide a mapping for negative numbers. We could change our definition to make the domain of the function the non-negative numbers, but such changes are often inconvenient, or even impossible: inconvenient because we should like all mathematical functions to operate on $\mathcal{R}$; impossible because we may not know which elements of the domain have mappings and which do not. Consider for example the function f that maps every natural number a to the smallest natural number b such that the digits of the decimal representation of a appear b digits to the right of the decimal point in the decimal expansion of π. Clearly $f(59) = 4$, because $\pi = 3.14159\ldots$. But what about $f(428945028)$, or in general $f(n)$ for arbitrary n? Absent results from number theory, it is not at all clear how to characterize the values at which f is defined. In such a case a partial function is essential.

EXAMPLE 11.78
Functions as sets

It is often useful to characterize functions as sets or, more precisely, as subsets of the Cartesian product of the domain and the range:

$$\text{sqrt} \subset [\mathcal{R} \times \mathcal{R}]$$
$$\text{plus} \subset [\mathcal{R} \times \mathcal{R} \times \mathcal{R}]$$

We can specify *which* subset using traditional set notation:

$$\text{sqrt} \equiv \left\{(x, y) \in \mathcal{R} \times \mathcal{R} \mid y > 0 \wedge x = y^2\right\}$$
$$\text{plus} \equiv \{(x, y, z) \in \mathcal{R} \times \mathcal{R} \times \mathcal{R} \mid z = x + y\}$$

Note that this sort of definition tells us what the value of a function like sqrt is, but it does *not* tell us how to compute it; more on this distinction below. ■

EXAMPLE 11.79
Functions as powerset elements

One of the nice things about the set-based characterization is that it makes it clear that a function is an ordinary mathematical object. We know that a function from A to B is a subset of $A \times B$. This means that it is an *element* of the *powerset* of $A \times B$—the set of all subsets of $A \times B$, denoted $2^{A \times B}$:

$$\text{sqrt} \in 2^{\mathcal{R} \times \mathcal{R}}$$

Similarly,

$$\text{plus} \in 2^{\mathcal{R} \times \mathcal{R} \times \mathcal{R}}$$

Note the overloading of notation here. The powerset 2^A should not be confused with exponentiation, though it is true that for a finite set A the number of elements in the powerset of A is 2^n, where $n = |A|$, the cardinality of A. ■

EXAMPLE 11.80
Function spaces

Because functions are single-valued, we know that they constitute only *some* of the elements of $2^{A \times B}$. Specifically, they constitute all and only those sets of pairs in which the first component of each pair is unique. We call the set of such sets the *function space* of A into B, denoted $A \to B$. Note that $(A \to B) \subset 2^{A \times B}$. In our examples:

$$\text{sqrt} \in [\mathcal{R} \to \mathcal{R}]$$
$$\text{plus} \in [(\mathcal{R} \times \mathcal{R}) \to \mathcal{R}]$$

EXAMPLE 11.81
Higher-order functions as sets

Now that functions are elements of sets, we can easily build higher-order functions:

$$\text{compose} \equiv \{(f, g, h) \mid \forall x \in \mathcal{R},\ h(x) = f(g(x))\}$$

What are the domain and range of compose? We know that f, g, and h are elements of $\mathcal{R} \to \mathcal{R}$. Thus

$$\text{compose} \in [(\mathcal{R} \to \mathcal{R}) \times (\mathcal{R} \to \mathcal{R})] \to (\mathcal{R} \to \mathcal{R})$$

Note the similarity to the notation employed by the ML type system (Section 7.2.4). ■

EXAMPLE 11.82
Curried functions as sets

Using the notion of "currying" from Section 11.6, we note that there is an alternative characterization for functions like plus. Rather than a function from pairs of reals to reals, we can capture it as a function from reals to functions from reals to reals:

$$\text{curried\_plus} \in \mathcal{R} \to (\mathcal{R} \to \mathcal{R})$$

■

We shall have more to say about currying in Section C-11.7.3.

11.7.1 Lambda Calculus

As we suggested in the main text, one of the limitations of the function-as-set notation is that it is *nonconstructive*: it doesn't tell us how to *compute* the value of a function at a given point (i.e., on a given input). Church designed the lambda calculus to address this limitation. In its pure form, lambda calculus represents *everything* as a function. The natural numbers, for example, can be represented by a distinguished zero function (commonly the identity function) and a successor function. (One common formulation uses a select_second function that takes two arguments and returns the second of them. The successor function is then defined in such a way that the number n ends up being represented by a function that, when applied to select_second n times, returns the identity function [Mic89, Sec. 3.5]; [Sta95, Sec. 7.6]; see Exercise C-11.23.) While of theoretical importance, this formulation of arithmetic is highly cumbersome. We will therefore take ordinary arithmetic as a given in the remainder of this subsection. (And of course all practical functional programming languages provide built-in support for both integer and floating-point arithmetic.)

A lambda expression can be defined recursively as (1) a *name*; (2) a lambda *abstraction* consisting of the letter λ, a name, a dot, and a lambda expression; (3) a function *application* consisting of two adjacent lambda expressions; or (4) a parenthesized lambda expression. To accommodate arithmetic, we will extend this definition to allow numeric literals.

EXAMPLE 11.83
Juxtaposition as function application

When two expressions appear adjacent to one another, the first is interpreted as a function to be applied to the second:

sqrt n

Most authors assume that application associates left-to-right (so $f\ A\ B$ is interpreted as $(f\ A)\ B$, rather than $f\ (A\ B)$), and that application has higher precedence than abstraction (so $\lambda x.A\ B$ is interpreted as $\lambda x.(A\ B)$, rather than $(\lambda x.A)\ B$). ML adopts these rules. ■

EXAMPLE 11.84
Lambda calculus syntax

Parentheses are used as necessary to override default groupings. Specifically, if we distinguish between lambda expressions that are used as functions and those that are used as arguments, then the following unambiguous CFG can be used to generate lambda expressions with a minimal number of parentheses:

expr ⟶ `name` | *number* | λ `name` `.` *expr* | *func arg*
func ⟶ `name` | `(` λ `name` `.` *expr* `)` | *func arg*
arg ⟶ `name` | *number* | `(` λ `name` `.` *expr* `)` | `(` *func arg* `)`

In words: we use parentheses to surround an abstraction that is used as either a function or an argument, and around an application that is used as an argument. ■

EXAMPLE 11.85
Binding parameters with λ

The letter λ introduces the lambda calculus equivalent of a formal parameter. The following lambda expression denotes a function that returns the square of its argument:

$$\lambda x.\text{times}\ x\,x$$

The name (variable) introduced by a λ is said to be *bound* within the expression following the dot. In programming language terms, this expression is the variable's scope. A variable that is not bound is said to be *free*. ■

EXAMPLE 11.86
Free variables

As in a lexically scoped programming language, a free variable needs to be defined in some surrounding scope. Consider, for example, the expression $\lambda x.\lambda y.\text{times}\ x\,y$. In the inner expression $(\lambda y.\text{times}\ x\,y)$, y is bound but x is free. There are no restrictions on the use of a bound variable: it can play the role of a function, an argument, or both. Higher-order functions are therefore completely natural. ■

EXAMPLE 11.87
Naming functions for future reference

If we wish to refer to them later, we can give expressions names:

$$\begin{aligned}
\text{square} &\equiv \lambda x.\text{times}\ x\,x \\
\text{identity} &\equiv \lambda x.x \\
\text{const7} &\equiv \lambda x.7 \\
\text{hypot} &\equiv \lambda x.\lambda y.\text{sqrt}\ (\text{plus}\ (\text{square}\ x)\ (\text{square}\ y))
\end{aligned}$$

Here $\equiv$ is a metasymbol meaning, roughly, "is an abbreviation for." ■

EXAMPLE 11.88
Evaluation rules

To compute with the lambda calculus, we need rules to evaluate expressions. It turns out that three rules suffice:

beta reduction: For any lambda abstraction $\lambda x.E$ and any expression M, we say

$$(\lambda x.E)\,M \rightarrow_\beta E[M\backslash x]$$

where $E[M\backslash x]$ denotes the expression E with all free occurrences of x replaced by M. Beta reduction is not permitted if any free variables in M would become bound in $E[M\backslash x]$.

alpha conversion: For any lambda abstraction $\lambda x.E$ and any variable y that has no free occurrences in E, we say

$$\lambda x.E \rightarrow_\alpha \lambda y.E[y\backslash x]$$

eta reduction: A rule to eliminate "surplus" lambda abstractions. For any lambda abstraction $\lambda x.E$, where E is of the form $F\,x$, and x has no free occurrences in F, we say

$$\lambda x.F\,x \rightarrow_\eta F$$

■

EXAMPLE 11.89
Delta reduction for arithmetic

To accommodate arithmetic we will also allow an expression of the form op x y, where x and y are numeric literals and op is one of a small set of standard functions, to be replaced by its arithmetic value. This replacement is called *delta reduction*. In our examples we will need only the functions plus, minus, and times:

$$\begin{aligned} \text{plus } 2\,3 &\rightarrow_\delta 5 \\ \text{minus } 5\,2 &\rightarrow_\delta 3 \\ \text{times } 2\,3 &\rightarrow_\delta 6 \end{aligned}$$

■

Beta reduction resembles the use of call by name parameters (Section 9.3.1). Unlike Algol 60, however, the lambda calculus provides no way for an argument to carry its referencing environment with it; hence the requirement that an argument not move a variable into a scope in which its name has a different meaning. Alpha conversion serves to change names to make beta reduction possible. Eta reduction is comparatively less important. If square is defined as above, eta reduction allows us to say that

EXAMPLE 11.90
Eta reduction

$$\lambda x.\text{square } x \rightarrow_\eta \text{square}$$

In English, square is a function that squares its argument; $\lambda x.$square x is a function of x that squares x. The latter reminds us explicitly that it's a function (i.e., that it takes an argument), but the former is a little less messy looking. ■

Through repeated application of beta reduction and alpha conversion (and possibly eta reduction), we can attempt to reduce a lambda expression to its simplest possible form—a form in which no further beta reductions are possible. An example can be found in Figure C-11.5. In line (2) of this derivation we have to employ an alpha conversion because the argument that we need to substitute for g contains a free variable (h) that is bound within g's scope. If we were to make the substitution of line (3) without first having renamed the bound h (as k), then the free h would have been *captured*, erroneously changing the meaning of the expression.

EXAMPLE 11.91
Reduction to simplest form

In line (5) of the derivation, we had a choice as to which subexpression to reduce. At that point the expression as a whole consisted of a function application in which the argument was itself a function application. We chose to substitute the main argument $((\lambda x.x\,x)\,(\lambda x.x\,x))$, unevaluated, into the body of the main lambda abstraction. This choice is known as *normal-order* reduction, and corresponds to normal-order evaluation of arguments in programming languages, as discussed in Sections 6.6.2 and 11.5. In general, whenever more than one beta reduction could be made, normal order chooses the one whose λ is left-most in the overall expression. This strategy substitutes arguments into functions before reducing them. The principal alternative, *applicative-order* reduction, reduces both the function part and the argument part of every function application to the simplest possible form before substituting the latter into the former. ■

$$
\begin{array}{lll}
 & (\underline{\lambda f}.\lambda g.\lambda h.fg(h\,h))(\underline{\lambda x.\lambda y.x})h(\lambda x.x\,x) & \\
\rightarrow_\beta & (\lambda g.\underline{\lambda h}.(\lambda x.\lambda y.x)g(\underline{h\,h}))h(\lambda x.x\,x) & (1)\\
\rightarrow_\alpha & (\underline{\lambda g}.\lambda k.(\lambda x.\lambda y.x)g(k\,k))\underline{h}(\lambda x.x\,x) & (2)\\
\rightarrow_\beta & (\underline{\lambda k}.(\lambda x.\lambda y.x)h(k\,k))\underline{(\lambda x.x\,x)} & (3)\\
\rightarrow_\beta & (\underline{\lambda x}.\lambda y.x)\underline{h}((\lambda x.x\,x)\,(\lambda x.x\,x)) & (4)\\
\rightarrow_\beta & (\underline{\lambda y}.h)\underline{((\lambda x.x\,x)\,(\lambda x.x\,x))} & (5)\\
\rightarrow_\beta & h & (6)
\end{array}
$$

Figure 11.5 **Reduction of a lambda expression.** The top line consists of a function applied to three arguments. The first argument (underlined) is the "select first" function, which takes two arguments and returns the first. The second argument is the symbol h, which must be either a constant or a variable bound in some enclosing scope (not shown). The third argument is an "apply to self" function that takes one argument and applies it to itself. The particular series of reductions shown occurs in normal order. It terminates with a simplest (normal) form of simply h.

Church and Rosser showed in 1936 that simplest forms are unique: any series of reductions that terminates in a nonreducible expression will produce the same result. Not all reductions terminate, however. In particular, there are expressions for which no series of reductions will terminate, and there are others in which normal-order reduction will terminate but applicative-order reduction will not. The example expression of Figure C-11.5 leads to an infinite "computation" under applicative-order reduction. To see this, consider the expression at line (5). This line consists of the constant function $(\lambda y.h)$ applied to the argument $(\lambda x.x\,x)\,(\lambda x.x\,x)$. If we attempt to evaluate the argument before substituting it into the function, we run through the following steps:

EXAMPLE 11.92

Nonterminating applicative-order reduction

$$
\begin{array}{ll}
 & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)}\\
\rightarrow_\beta & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)}\\
\rightarrow_\beta & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)}\\
\rightarrow_\beta & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)}\\
\ldots &
\end{array}
$$

In addition to showing the uniqueness of simplest (normal) forms, Church and Rosser showed that if any evaluation order will terminate, normal order will. This pair of results is known as the *Church-Rosser theorem*.

11.7.2 Control Flow

We noted at the beginning of the previous subsection that arithmetic can be modeled in the lambda calculus using a distinguished zero function (commonly

the identity) and a successor function. What about control-flow constructs—selection and recursion in particular?

EXAMPLE 11.93
Booleans and conditionals

The select_first function, $\lambda x.\lambda y.x$, is commonly used to represent the Boolean value true. The select_second function, $\lambda x.\lambda y.y$, is commonly used to represent the Boolean value false. Let us denote these by T and F. The nice thing about these definitions is that they allow us to define an if function very easily:

$$\text{if} \equiv \lambda c.\lambda t.\lambda e.c\ t\ e$$

Consider:

$$\begin{aligned} \text{if}\ T\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.x)\ 3\ 4 \\ &\rightarrow_\beta^* (\lambda x.\lambda y.x)\ 3\ 4 \\ &\rightarrow_\beta^* 3 \end{aligned}$$

$$\begin{aligned} \text{if}\ F\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.y)\ 3\ 4 \\ &\rightarrow_\beta^* (\lambda x.\lambda y.y)\ 3\ 4 \\ &\rightarrow_\beta^* 4 \end{aligned}$$

■

Functions like equal and greater_than can be defined to take numeric values as arguments, returning T or F.

EXAMPLE 11.94
Beta abstraction for recursion

Recursion is a little tricky. An equation like

$$\begin{aligned} \text{gcd} \equiv\ & \lambda a.\lambda b.(\text{if}\ (\text{equal}\ a\ b)\ a \\ & (\text{if}\ (\text{greater\_than}\ a\ b)\ (\text{gcd}\ (\text{minus}\ a\ b)\ b)\ (\text{gcd}\ (\text{Minus}\ b\ a)\ a))) \end{aligned}$$

is not really a definition at all, because gcd appears on both sides. Our previous definitions (T, F, if) were simply shorthand: we could substitute them out to obtain a pure lambda expression. If we try that with gcd, the "definition" just gets bigger, with new occurrences of the gcd name. To obtain a real definition, we first rewrite our equation using *beta abstraction* (the opposite of beta reduction):

$$\begin{aligned} \text{gcd} \equiv\ & (\lambda g.\lambda a.\lambda b.(\text{if}\ (\text{equal}\ a\ b)\ a \\ & (\text{if}\ (\text{greater\_than}\ a\ b)\ (g(\text{minus}\ a\ b)\ b)\ (g(\text{minus}\ b\ a)\ a)))) \ \text{gcd} \end{aligned}$$

Now our equation has the form

$$\text{gcd} \equiv f\ \text{gcd}$$

where f is the perfectly well-defined (nonrecursive) lambda expression

$$\begin{aligned} & \lambda g.\lambda a.\lambda b.(\text{if}\ (\text{equal}\ a\ b)\ a \\ & \quad (\text{if}\ (\text{greater\_than}\ a\ b)\ (g\ (\text{minus}\ a\ b)\ b)\ (g\ (\text{minus}\ b\ a)\ a))) \end{aligned}$$

Clearly gcd is a fixed point of f.

■

EXAMPLE 11.95
The fixed-point combinator **Y**

As it turns out, for any function f given by a lambda expression, we can find the least fixed point of f, if there is one, by applying the *fixed-point combinator*

$$\lambda h.(\lambda x.h(xx))\ (\lambda x.h(xx))$$

commonly denoted **Y**. **Y** has the property that for any lambda expression f, if the normal-order evaluation of $\mathbf{Y}f$ terminates, then $f(\mathbf{Y}f)$ and $\mathbf{Y}f$ will reduce to the same simplest form (see Exercise C-11.21). In the case of our gcd function, we have

$$\begin{aligned}\text{gcd} \equiv\ & (\lambda h.(\lambda x.h(x\,x))\ (\lambda x.h(x\,x))) \\ & (\lambda g.\lambda a.\lambda b.(\text{if (equal } a\ b)\ a \\ & \quad (\text{if (greater\_than } a\ b)\ (g(\text{minus } a\ b)\ b)\ (g(\text{minus } b\ a)\ a))))\end{aligned}$$

Figure C-11.6 traces the evaluation of gcd 4 2. Given the existence of the **Y** combinator, most authors permit recursive "definitions" of functions, for convenience. ■

11.7.3 Structures

EXAMPLE 11.96
Lambda calculus list operators

Just as we can use functions to build numbers and truth values, we can also use them to encapsulate values in structures. Using Scheme terminology for the sake of clarity, we can define simple list-processing functions as follows:

$$\begin{aligned}\text{cons} &\equiv \lambda a.\lambda d.\lambda x.x\,a\,d \\ \text{car} &\equiv \lambda l.l\ \text{select\_first} \\ \text{cdr} &\equiv \lambda l.l\ \text{select\_second} \\ \text{nil} &\equiv \lambda x.T \\ \text{null?} &\equiv \lambda l.l(\lambda x.\lambda y.F)\end{aligned}$$

where select_first and select_second are the functions $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$, respectively—functions we also use to represent true and false. ■

EXAMPLE 11.97
List operator identities

Using these definitions we can see that

$$\begin{aligned}\text{car}(\text{cons } A\ B) &\equiv (\lambda l.l\ \text{select\_first})\ (\text{cons } A\ B) \\ &\to_\beta (\text{cons } A\ B)\ \text{select\_first} \\ &\equiv ((\lambda a.\lambda d.\lambda x.x\,a\,d)\ A\ B)\ \text{select\_first} \\ &\to_\beta^* (\lambda x.x\,A\,B)\ \text{select\_first} \\ &\to_\beta \text{select\_first}\ A\ B \\ &\equiv (\lambda x.\lambda y.x)\ A\ B \\ &\to_\beta^* A\end{aligned}$$

$$
\begin{aligned}
\text{gcd } 2\,4 &\equiv \mathbf{Y} f\,2\,4 \\
&\equiv ((\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x)))f)\,2\,4 \\
&\to_\beta ((\lambda x.f(x\,x))\,(\lambda x.f(x\,x)))\,2\,4 \\
&\equiv (k\,k)\,2\,4, \quad \text{where } k \equiv \lambda x.f(x\,x) \\
&\to_\beta (f(k\,k))\,2\,4 \\
&\equiv ((\lambda g.\lambda a.\lambda b.(\text{if } (=a\,b)\,a\,(\text{if } (>a\,b)\,(g(-\,a\,b)\,b)\,(g(-\,b\,a)\,a))))\,(k\,k))\,2\,4 \\
&\to_\beta (\lambda a.\lambda b.(\text{if } (=a\,b)\,a\,(\text{if } (>a\,b)\,((k\,k)(-\,a\,b)\,b)\,((k\,k)(-\,b\,a)\,a))))\,2\,4 \\
&\to^*_\beta \text{if } (=2\,4)\,2\,(\text{if } (>2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\equiv (\lambda c.\lambda t.\lambda e.c\,t\,e)\,(=2\,4)\,2\,(\text{if } (>2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\to^*_\beta (=2\,4)\,2\,(\text{if } (>2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\to_\delta F\,2\,(\text{if } (>2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\equiv (\lambda x.\lambda y.y)\,2\,(\text{if } (>2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\to^*_\beta \text{if } (>2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2) \\
&\to \ldots \\
&\to (k\,k)\,(-\,4\,2)\,2 \\
&\equiv ((\lambda x.f(x\,x))k)\,(-\,4\,2)\,2 \\
&\to_\beta (f(k\,k))\,(-\,4\,2)\,2 \\
&\equiv ((\lambda g.\lambda a.\lambda b.(\text{if } (=a\,b)\,a\,(\text{if } (>a\,b)\,(g(-\,a\,b)\,b)\,(g(-\,b\,a)\,a))))\,(k\,k))\,(-\,4\,2)\,2 \\
&\to_\beta (\lambda a.\lambda b.(\text{if } (=a\,b)\,a\,(\text{if } (>a\,b)\,((k\,k)(-\,a\,b)\,b)\,((k\,k)(-\,b\,a)\,a))))\,(-\,4\,2)\,2 \\
&\to^*_\beta \text{if } (=(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if } (>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\equiv (\lambda c.\lambda t.\lambda e.c\,t\,e) \\
&\quad\ (=(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if } (>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to^*_\beta (=(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if } (>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to_\delta (=2\,2)\,(-\,4\,2)\,(\text{if } (>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to_\delta T\,(-\,4\,2)\,(\text{if } (>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\equiv (\lambda x.\lambda y.x)\,(-\,4\,2)\,(\text{if } (>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to^*_\beta (-\,4\,2) \\
&\to_\delta 2
\end{aligned}
$$

Figure 11.6 **Evaluation of a recursive lambda expression.** As explained in the body of the text, gcd is defined to be the fixed-point combinator $\mathbf{Y}$ applied to a beta abstraction f of the standard recursive definition for greatest common divisor. Specifically, $\mathbf{Y}$ is $\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x))$ and f is $\lambda g.\lambda a.\lambda b.(\text{if } (=a\,b)\,a\,(\text{if } (>a\,b)\,(g(-\,a\,b)\,b)\,(g(-\,b\,a)\,a)))$. For brevity we have used $=$, $>$, and $-$ in place of equal, greater_than, and minus. We have performed the evaluation in normal order.

$$\begin{aligned}
\text{cdr}(\text{cons}\ A\ B) &\equiv (\lambda l.l\ \text{select_second})\ (\text{cons}\ A\ B)\\
&\to_\beta (\text{cons}\ A\ B)\ \text{select_second}\\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \text{select_second}\\
&\to_\beta^* (\lambda x.x\ A\ B)\ \text{select_second}\\
&\to_\beta \text{select_second}\ A\ B\\
&\equiv (\lambda x.\lambda y.y)\ A\ B\\
&\to_\beta^* B
\end{aligned}$$

$$\begin{aligned}
\text{null? nil} &\equiv (\lambda l.l\ (\lambda x.\lambda y.\text{select_second}))\ \text{nil}\\
&\to_\beta \text{nil}\ (\lambda x.\lambda y.\text{select_second})\\
&\equiv (\lambda x.\text{select_first})\ (\lambda x.\lambda y.\text{select_second})\\
&\to_\beta \text{select_first}\\
&\equiv T
\end{aligned}$$

$$\begin{aligned}
\text{null? (cons}\ A\ B) &\equiv (\lambda l.l\ (\lambda x.\lambda y.\text{select_second}))\ (\text{cons}\ A\ B)\\
&\to_\beta (\text{cons}\ A\ B)\ (\lambda x.\lambda y.\text{select_second})\\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ (\lambda x.\lambda y.\text{select_second})\\
&\to_\beta^* (\lambda x.x\ A\ B)\ (\lambda x.\lambda y.\text{select_second})\\
&\to_\beta (\lambda x.\lambda y.\text{select_second})\ A\ B\\
&\to_\beta^* \text{select_second}\\
&\equiv F
\end{aligned}$$

EXAMPLE 11.98
Nesting of lambda expressions

Because every lambda abstraction has a single argument, lambda expressions are naturally curried. We generally obtain the effect of a multiargument function by nesting lambda abstractions:

$$\text{compose} \equiv \lambda f.\lambda g.\lambda x.f\ (g\ x)$$

which groups as

$$\lambda f.(\lambda g.(\lambda x.(f\ (g\ x))))$$

We commonly think of compose as a function that takes two functions as arguments and returns a third function as its result. We could just as easily, however, think of compose as a function of three arguments: the f, g, and x above. The official story, or course, is that compose is a function of one argument that evaluates to a function of one argument that in turn evaluates to a function of one argument.

EXAMPLE 11.99
Paired arguments and currying

If desired, we can use our structure-building functions to define a noncurried version of compose whose (single) argument is a pair:

$$\text{paired\_compose} \equiv \lambda p.\lambda x.(\text{car } p)\ ((\text{cdr } p)\ x)$$

If we consider the pairing of arguments as a general technique, we can write a curry function that reproduces the single-argument version, just as we did in Scheme in Section 11.6:

$$\text{curry} \equiv \lambda f.\lambda a.\lambda b.f\,(\text{cons } a\ b)$$

CHECK YOUR UNDERSTANDING

29. What is the difference between *partial* and *total* functions? Why is the difference important?
30. What is meant by the *function space* $A \rightarrow B$?
31. Define *beta reduction*, *alpha conversion*, *eta reduction*, and *delta reduction*.
32. How does beta reduction in lambda calculus differ from lazy evaluation of arguments in a nonstrict programming language like Haskell?
33. Explain how lambda expressions can be used to represent Boolean values and control flow.
34. What is *beta abstraction*?
35. What is the **Y** combinator? What useful property does it possess?
36. Explain how lambda expressions can be used to represent structured values such as lists.
37. State the *Church-Rosser theorem*.

N - normal
A - applicative

$(\lambda fgh.\, fg(hh))(\lambda xy.x)\, h\, (\lambda x.xx)$

$(\lambda f.\lambda g.\lambda h.\, fg(hh))(\lambda x.\lambda y.x)\, h\, (\lambda x.xx)$

$\xrightarrow[\beta]{N,A} (\lambda g.\lambda h.\, (\lambda x.\lambda y.x)\, g\, (hh))\, h\, (\lambda x.xx)$

$\xrightarrow{A}_{\beta} (\lambda g.\lambda h.\, (\lambda y.g)\, (hh))\, h\, (\lambda x.xx)$

$\Rightarrow_{\alpha} (\lambda g.\lambda k.\, (\lambda x.\lambda y.x)\, g\, (kk))\, h\, (\lambda x.xx)$

$\xrightarrow{A}_{\beta} (\lambda g.\lambda h.\, g)\, h\, (\lambda x.xx)$

$\xrightarrow{N}_{\beta} (\lambda k.\, (\lambda x.\lambda y.x)\, h\, (kk))\, (\lambda x.xx)$

$\Rightarrow_{\alpha} (\lambda g.\lambda k.\, g)\, h\, (\lambda x.xx)$

$\xrightarrow{N}_{\beta} (\lambda x.\lambda y.x)\, h\, ((\lambda x.xx)(\lambda x.xx))$

$\xrightarrow{A}_{\beta} (\lambda k.h)(\lambda x.xx)$

$\xrightarrow{N}_{\beta} (\lambda y.h)\, ((\lambda x.xx)(\lambda x.xx))$

$\xrightarrow{A}_{\beta} h$

$\xrightarrow{N}_{\beta} h$

$$(\lambda x.(\lambda y.y)\,z\,x)\,(\lambda w.(\lambda x.w)\,h)$$

$$\xRightarrow{N}_\beta (\lambda y.y)\,z\,(\lambda w.(\lambda x.w)\,h)$$

$$\xRightarrow{N}_\beta z\,((\lambda w.(\lambda x.w)\,h))$$

$$\xRightarrow{N}_\beta z\,(\lambda w.w)$$

$$\xRightarrow{A}_\beta (\lambda x.z\,x)\,(\lambda w.(\lambda x.w)\,h)$$

$$\xRightarrow{A}_\beta (\lambda x.z\,x)\,(\lambda w.w)$$

$$\xRightarrow{A}_\beta z\,(\lambda w.w)$$

Predicate Calculus

Chapter 12, Section 3

Predicate calculus

- *Predicate*: function that maps constants and variables to `true` and `false`
- *First order predicate calculus*: notation and inference rules for constructing and reasoning about propositions:
- Operators:
  - and $\wedge$
  - or $\vee$
  - not $\neg$
  - implication $\rightarrow$
  - equivalence $\leftrightarrow$
- Quantifiers:
  - existential $\exists$
  - universal $\forall$

Predicate calculus

- Examples

$\forall C(\texttt{rainy}(C) \land \texttt{cold}(C) \rightarrow \texttt{snowy}(C))$

$\forall A, \forall B(\texttt{takes}(A, C) \land \texttt{takes}(B, C) \rightarrow \texttt{classmates}(A, B))$

- Fermat's last Theorem:

$\forall N\,((N > 2) \rightarrow \neg(\exists A\ \exists B\ \exists C(A^N + B^N = C^N)))$

- $\forall$, $\exists$ bind variables like λ in λ-calculus

Predicate calculus

- Normal form
  - the same thing can be written in different ways:

$$(P \rightarrow Q) \equiv (\neg P \vee Q)$$

$$\neg \exists X\,(P(X)) \equiv \forall X\,(\neg P(X))$$

$$\neg (P \wedge Q) \equiv (\neg P \vee \neg Q)$$

  - This is good for humans, bad for machines
  - Automatic theorem proving requires a normal form

Clausal Form

- *Clausal form*
- Example:

$$\forall X\,(\neg \texttt{student}(X) \rightarrow (\neg \texttt{resident}(X) \wedge \neg \exists Y\,(\texttt{takes}(X, Y) \wedge \texttt{class}(Y))))$$

- 1. eliminate $\rightarrow$ and $\leftrightarrow$:

$$\forall X\,(\texttt{student}(X) \vee (\neg \texttt{resident}(X) \wedge \neg \exists Y\,(\texttt{takes}(X, Y) \wedge \texttt{class}(Y))))$$

Clausal Form

$\forall X(\texttt{student}(X) \vee (\neg\texttt{resident}(X) \wedge \neg\exists Y(\texttt{takes}(X, Y) \wedge \texttt{class}(Y))))$

- 2. move $\neg$ inward (using De Morgan's laws):

$\forall X(\texttt{student}(X) \vee (\neg\texttt{resident}(X) \wedge \forall Y(\neg(\texttt{takes}(X, Y) \wedge \texttt{class}(Y)))))$

$\equiv$

$\forall X(\texttt{student}(X) \vee (\neg\texttt{resident}(X) \wedge \forall Y(\neg\texttt{takes}(X, Y) \vee \neg\texttt{class}(Y))))$

Clausal Form

$\forall X\,(\texttt{student}(X) \vee (\neg\texttt{resident}(X) \wedge \forall Y\,(\neg\texttt{takes}(X, Y) \vee \neg\texttt{class}(Y))))$

- 3. eliminate existential quantifiers
  - Skolemization (not necessary in our example)
- 4. pull universal quantifiers to the outside of the proposition (some renaming might be needed)

$\forall X\,\forall Y\,(\texttt{student}(X) \vee (\neg\texttt{resident}(X) \wedge (\neg\texttt{takes}(X, Y) \vee \neg\texttt{class}(Y))))$

- convention: rules are universally quantified
  - we drop the implicit ∀'s:

$\texttt{student}(X) \vee (\neg\texttt{resident}(X) \wedge (\neg\texttt{takes}(X, Y) \vee \neg\texttt{class}(Y)))$

Clausal Form

$\texttt{student}(X) \vee (\neg\texttt{resident}(X) \wedge (\neg\texttt{takes}(X, Y) \vee \neg\texttt{class}(Y)))$

- 5. convert the proposition in *conjunctive normal form* (*CNF*)
  - conjunction of disjunctions

$(\texttt{student}(X) \vee \neg\texttt{resident}(X)) \wedge$
$(\texttt{student}(X) \vee \neg\texttt{takes}(X, Y) \vee \neg\texttt{class}(Y))$

Clausal Form

$(\texttt{student}(X) \vee \neg\texttt{resident}(X)) \wedge$
$(\texttt{student}(X) \vee \neg\texttt{takes}(X, Y) \vee \neg\texttt{class}(Y))$

- We can rewrite as:

$(\texttt{resident}(X) \rightarrow \texttt{student}(X)) \wedge$
$((\texttt{takes}(X, Y) \wedge \texttt{class}(Y)) \rightarrow \texttt{student}(X))$

$\equiv$

$(\texttt{student}(X) \leftarrow \texttt{resident}(X)) \wedge$
$(\texttt{student}(X) \leftarrow (\texttt{takes}(X, Y) \wedge \texttt{class}(Y)))$

Clausal Form

- We obtained:

$(\texttt{student}(X) \leftarrow \texttt{resident}(X)) \wedge$
$(\texttt{student}(X) \leftarrow (\texttt{takes}(X, Y) \wedge \texttt{class}(Y)))$

- which translates directly to Prolog:

```
student(X) :- resident(X).
student(X) :- takes(X, Y), class(Y).
```

`:-` means "if"

`,` means "and"

Horn Clauses

- *Horn clauses*
  - particular case of clauses: only one non-negated term:

$$\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_k \vee P \equiv$$
$$Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k \rightarrow P \equiv$$
$$P \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

  - which is a *rule* in Prolog:

```
P :- Q1, Q2,...,Qk.
```

  - for $k = 0$ we have a *fact*:

```
P.
```

Automated proving

- Rule: both sides of `:-`

`P :- Q1, Q2,...,Qk.` means $P \leftarrow Q_1 \wedge Q_2 \wedge ... \wedge Q_k$

- Fact: left-hand side of (implicit) `:-`

`P.` means $P \leftarrow$ `true`

- Query: right-hand side of (implicit) `:-`

```
?- Q1, Q2,...,Qk.
```

- *Automated proving*: given a collection of axioms (facts and rules), add the *negation* of the theorem (query) we want to prove and attempt (using *resolution*) to obtain a contradiction
  - Query negation: $\neg(Q_1 \wedge Q_2 \wedge ... \wedge Q_k)$

Automated proving

- Example

```
student(john).
?- student(john).
true.
```

- Fact: `student`(john) ← `true`
- Query (negated):

  ¬`student`(john) ≡ `false` ← `student`(john)

- We obtain a contradiction (that proves the query):

`false` ← `student`(john) ← `true`

- The above contradiction is obvious; in general, use *resolution*.

Resolution

- *Resolution* (propositional logic):
  - From hypotheses:

    $(A_1 \vee A_2 \vee \ldots \vee A_k \vee C) \wedge (B_1 \vee B_2 \vee \ldots \vee B_l \vee \neg C)$

  - We can obtain the conclusion:

    $A_1 \vee A_2 \vee \ldots \vee A_k \vee B_1 \vee B_2 \vee \ldots \vee B_l$

- Example: *modus ponens*

  $p \rightarrow q \wedge p$ gives q (because $p \rightarrow q$ is $\neg p \vee q$)

- In predicate logic:
  - C and $\neg C'$: where C, C' may not be identical but can be *unified*: that means, they can be made identical by substituting variables (details later)

Resolution example

```
student(X) :- resident(X).
student(X) :- takes(X, Y), class(Y).
resident(john).
takes(mark, 3342).
class(3342).

?- student(john).
true
```

- Resolution (add negation of query):

$(\neg \texttt{resident}(X) \vee \texttt{student}(X)) \wedge$
$(\neg \texttt{takes}(Y, Z) \vee \neg \texttt{class}(Z) \vee \texttt{student}(Y)) \wedge$
$\texttt{resident}(\text{john}) \wedge$
$\texttt{takes}(\text{mark}, 3342) \wedge$
$\texttt{class}(3342) \wedge$
$\neg \texttt{student}(\text{john})$

Resolution example

$(\neg \texttt{resident}(X) \vee \texttt{student}(X)) \wedge$
$(\neg \texttt{takes}(Y, Z) \vee \neg \texttt{class}(Z) \vee \texttt{student}(Y)) \wedge$
$\texttt{resident}(\text{john}) \wedge$
$\texttt{takes}(\text{mark}, 3342) \wedge$
$\texttt{class}(3342) \wedge$
$\neg \texttt{student}(\text{john})$

- $\texttt{student}(X)$ and $\texttt{student}(\text{john})$ *unify* for $X = \text{john}$

$(\neg \texttt{resident}(\text{john}) \vee \texttt{student}(\text{john})) \wedge$
$(\neg \texttt{takes}(Y, Z) \vee \neg \texttt{class}(Z) \vee \texttt{student}(Y)) \wedge$
$\texttt{resident}(\text{john}) \wedge$
$\texttt{takes}(\text{mark}, 3342) \wedge$
$\texttt{class}(3342) \wedge$
$\neg \texttt{student}(\text{john})$

Resolution example

$(\neg \text{resident}(\text{john}) \vee \text{student}(\text{john})) \wedge$
$(\neg \text{takes}(Y, Z) \vee \neg \text{class}(Z) \vee \text{student}(Y)) \wedge$
$\text{resident}(\text{john}) \wedge$
$\text{takes}(\text{mark}, 3342) \wedge$
$\text{class}(3342) \wedge$
$\neg \text{student}(\text{john})$

- resolution gives:

$\neg \text{resident}(\text{john}) \wedge$
$(\neg \text{takes}(Y, Z) \vee \neg \text{class}(Z) \vee \text{student}(Y)) \wedge$
$\text{resident}(\text{john}) \wedge$
$\text{takes}(\text{mark}, 3342) \wedge$
$\text{class}(3342)$

Resolution example

$\neg$resident(john) $\wedge$
($\neg$takes(Y, Z) $\vee$ $\neg$class(Z) $\vee$ student(Y)) $\wedge$
resident(john) $\wedge$
takes(mark, 3342) $\wedge$
class(3342)

- Resolution gives:

($\square$) $\wedge$
($\neg$takes(Y, Z) $\vee$ $\neg$class(Z) $\vee$ student(Y)) $\wedge$
takes(mark, 3342) $\wedge$
class(3342)

- The empty clause ($\square$) is not satisfiable
- We obtained a contradiction showing that student(john) is *provable* from the given axioms

Resolution example

```
?- student(matthew).
false.
```

- Resolution:

$(\neg \texttt{resident}(X) \vee \texttt{student}(X)) \wedge$
$(\neg \texttt{takes}(Y, Z) \vee \neg \texttt{class}(Z) \vee \texttt{student}(Y)) \wedge$
$\texttt{resident}(\text{john}) \wedge$
$\texttt{takes}(\text{mark}, 3342) \wedge$
$\texttt{class}(3342) \wedge$
$\neg \texttt{student}(\text{matthew})$

$\neg \texttt{resident}(\text{matthew}) \wedge$
$(\neg \texttt{takes}(Y, Z) \vee \neg \texttt{class}(Z) \vee \texttt{student}(Y)) \wedge$
$\texttt{resident}(\text{john}) \wedge$
$\texttt{takes}(\text{mark}, 3342) \wedge$
$\texttt{class}(3342)$

Resolution example

¬resident(matthew) ∧

(¬takes(*Y*, 3342) ∨ student(*Y*)) ∧

resident(john) ∧

takes(mark, 3342)

¬resident(matthew) ∧

student(mark) ∧

resident(john)

- cannot obtain a contradiction
- student(matthew) is *not provable* from the given axioms

Skolemization

- So far we did not worry about existential quantifiers
- What if we have:

$$\exists X\,(\texttt{takes}(X, 3342) \wedge \texttt{year}(X, 2))$$

- To get rid of the ∃, we introduce a constant, a, (as a notation for the one which is assumed to exists by ∃)

$$\texttt{takes}(\text{a}, 3342) \wedge \texttt{year}(\text{a}, 2)$$

Skolemization

- What if we do this inside the scope of a universal quantifier ∀:

$$\forall X\,(\neg \texttt{resident}(X) \vee \exists Y\,(\texttt{address}(X, Y)))$$

- We get rid again of ∃ by choosing an address which depends on X, say $\texttt{ad}(X)$:

$$\forall X\,(\neg \texttt{resident}(X) \vee (\texttt{address}(X, \texttt{ad}(X))))$$

Skolemization

- In Prolog

```
takes(a, 3342).
year(a, 2).
address(X, ad(X)) :- resident(X).
class_with_2nd(C) :- takes(X, C), year(X, 2).
has_address(X) :- address(X, Y).
resident(b).


?- class_with_2nd(C).
C = 3342
?- has_address(X).
X = b
```

Skolemization

```
?- takes(X, 3342).
X = a
```

- We cannot identify a 2nd-year student in 3342 by name

```
?- address(b, X).
X = ad(b).
```

- We cannot find out the address of b

Horn Clauses Limitations

- Horn clauses: only *one* non-negated term (*head*):

$$\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_k \vee P \equiv P \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

```
P :- Q1, Q2,...,Qk.
```

- If we have *more than one* non-negated term (two heads):

$$\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_k \vee P_1 \vee P_2 \equiv P_1 \vee P_2 \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

- then we have a disjunction in the left-hand side of ← (`:-`)

```
P1 or P2 :- Q1, Q2,...,Qk.
```

- which is not allowed in Prolog

Horn Clauses Limitations

- If we have *less than one* (zero) non-negated terms:

$$\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_k$$
$$\equiv$$
$$\texttt{false} \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

- the closest we have is:

```
:- Q1, Q2,...,Qk.
```

- which Prolog allows a query, not a rule

Horn Clauses Limitations

- Example: two heads

  "every living thing is an animal or a plant"

- Clausal form:

  $\texttt{animal}(X) \vee \texttt{plant}(X) \leftarrow \texttt{living}(X) \equiv$

  $\texttt{animal}(X) \vee \texttt{plant}(X) \vee \neg\texttt{living}(X)$

- In Prolog, the closest we can do is:

```
animal(X) :- living(X), not(plant(X)).
plant(X) :- living(X), not(animal(X)).
```

- which is not the same, because, as we'll see later, `not` indicates Prolog's inability to prove, not falsity

Logic Programming

Chapter 12

Logic Programming

Prolog says:

```
?- 1+1 = 2.
false.
```

so ... keep reading!

Logic Programming

- Algorithm = axioms + control
- Axioms
  - facts and rules
  - supplied by the programmer
- Control
  - computation is deduction
  - supplied by the language
- Given a set of axioms, the user states a theorem, or *goal*, and the language attempts to show that the axioms imply the goal

Logic Programming

- Axioms = Horn clauses

$$Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k \rightarrow P$$

or

$$P \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

- P is the *head*
- $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$ is the *body*

- $k \geq 1$: *rule*: if Q_1 and Q_2 and ... and Q_k, then P
- $k = 0$: *fact*: P (also: if `true`, then P)

- The meaning is that if all Q_i's are true, then we can deduce P

Prolog

- Imperative language:
  - runs in the context of a referencing environment, where various constants and functions have been defined
- Prolog
  - runs in the context of a database where various clauses have been defined
- Clause composed of *terms*:
  - *constants*:
    - *atoms*: id that starts with **lower** case: `foo`, `a` , `john`
    - *numbers*: 0, 2022
  - *variables*: id that starts with **upper** case: `Foo`, `X`
  - *structures*: *functor* (atom) and *argument list* (terms)
    - `student(john)`, `takes(X, cs3342)`
    - arguments can be constants, variables, (nested) structures

Prolog

- structures are interpreted as logical predicates
- predicate: functor + list of arguments
- Syntax:

term → *atom* | *number* | *variable* | *struct*
terms → *term* | *term* `,` *terms*
struct → *atom* `(` *terms* `)`
fact → *term* `.`
rule → *term* `:-` *terms* `.`
query → `?-` *terms* `.`

Prolog

- Rule:

$$P \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

- in Prolog:

```
P :- Q1, Q2,...,Qk.
```

- Fact (rule without right-hand side):

P ($P \leftarrow$ `true`)

- in Prolog:

```
P.
```

`:-`	=	$\leftarrow$
`,`	=	$\wedge$

Prolog

- Query (rule without left-hand side)

$$Q_1 \wedge Q_2 \wedge ... \wedge Q_k$$

- in Prolog:

```
?- Q1, Q2,...,Qk.
```

- the negated query is also:

$$\texttt{false} \leftarrow Q_1 \wedge Q_2 \wedge ... \wedge Q_k$$

Prolog

- Rules are implicitly universally quantified (∀)
- Example:

```
path(L, M) :- link(L, X), path(X, M).
```

- means:

$$\forall L, \forall M, \forall X \left(\texttt{path}(L, M) \textbf{ if } \left(\texttt{link}(L, X) \textbf{ and } \texttt{path}(X, M) \right) \right)$$

or

$$\forall L, \forall M \left(\texttt{path}(L, M) \textbf{ if } \left(\exists X \left(\texttt{link}(L, X) \textbf{ and } \texttt{path}(X, M) \right) \right) \right)$$

Prolog

- Queries are implicitly existentially quantified (∃)
- Example:

```
?- path(algol60, X), path(X, c).
```

- means

$$\exists X\,(\texttt{path}(\text{algol60}, X)\ \textbf{and}\ \texttt{path}(X, \text{c}))$$

Prolog

- Setting up working directory
- Checking working directory:

```
?- working_directory(X, X).
X = (//).
```

- Changing working directory:

```
?- working_directory(_,'/Users/Lucian/Documents/
4_myCourses/2021-2022/CS3342b_win2022/my_programs/Prolog').
true.
?- working_directory(X, X).
X = (_,'/Users/Lucian/Documents/4_myCourses/2021-2022/
CS3342b_win2022/my_programs/Prolog').
```

Prolog

- Facts and rules from a file:
  - reading the file "`my_file.pl`"
  - must be in the working directory

```
?- consult(my_file).
true.
```

Prolog

- Example:

```
rainy(seattle).
rainy(rochester).

?- rainy(C).
C = seattle
```

- Type ENTER if done
- Type '`;`' if you want more solutions

```
C = seattle ;
C = rochester.
```

Prolog

- Example:

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

```
?- snowy(C).
C = rochester.
```

- only one solution

Prolog

- Example:

```
link(fortran,  algol60).
link(algol60,  cpl).
link(cpl, bcpl).
link(bcpl, c).
link(c, cplusplus).
link(algol60,  simula67).
link(simula67, cplusplus).
link(simula67, smalltalk80).

path(L, L).
path(L, M) :- link(L, X), path(X, M).
```

Prolog

- Example:

```
?- link(simula67, X).
X = cplusplus ;
X = smalltalk80.

?- link(algol60, X), link(X, Y).
X = cpl,
Y = bcpl ;
X = simula67,
Y = cplusplus ;
X = simula67,
Y = smalltalk80.
```

Prolog

- Example:

```
?- path(fortran, cplusplus).
true ;
true ;
false.

?- path(X, cpl).
X = cpl ;
X = fortran ;
X = algol60 ;
false.
```

Prolog

- Example:

```
?- path(X,Y).
X = Y ;
X = fortran,
Y = algol60 ;
X = fortran,
Y = cpl ;
X = fortran,
Y = bcpl ;
X = fortran,
Y = c ;
X = fortran,
Y = cplusplus ; % ... it finds all paths
```

Lists

- `[a, b, c]` – list
- `[]` – empty list
- can use a `cons`-like predicate:

```
'[|]'(a, '[|]'(b, '[|]'(c, [])))
```

means `[a, b, c]`

- *Head* | *Tail* notation: `[H|T]`
- `[a, b, c]` can be written as:

```
[a | [b, c]]
[a, b | [c]]
[a, b, c | []]
```

Lists

```
?- [H|T] = [a, b, c].
H = a,
T = [b, c].

?- [H|T] = [[], c | [[a], b, [] | [b]]].
H = [],
T = [c, [a], b, [], b].

?- [H|[X|T]] = [[], c | [[a], b, [] | [b]]].
H = [],
X = c,
T = [[a], b, [], b].

?- [H1,H2|[X|T]] = [[],c | [[a], b, [] | [b]]].
H1 = [],
H2 = c,
X = [a],
T = [b, [], b].
```

List operations

- Searching an element in a list:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

- _ is a placeholder for a variable not needed anywhere else

List operations

- Searching an element in a list:

```
?- member(a, [b, a, c]).
true

?- member(a, [b, d, c]).
false.

?- member(a, X).
X = [a|_14708] ;
X = [_14706, a|_14714] ;
X = [_14706, _14712, a|_14720] ;
X = [_14706, _14712, _14718, a|_14726] ;
X = [_14706, _14712, _14718, _14724, a|_14732]
...
```

List operations

- Adding an element to a list:

```
add(X, L, [X|L]).
?- add(a, [b,c], L).
L = [a, b, c].
```

- Deleting an element from a list:

```
del(X, [X|T], T).
del(X, [Y|T], [Y|T1]) :- del(X, T, T1).

?- del(a, [a, b, c, a, b, a, d, a], X).
X = [b, c, a, b, a, d, a] ;
X = [a, b, c, b, a, d, a] ;
X = [a, b, c, a, b, d, a] ;
X = [a, b, c, a, b, a, d] ;
false.
```

List operations

- Appending two lists:

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

- Sublists:

```
sublist(S,L) :- append(_,L1,L), append(S,_,L1).
```

List operations

- Example:

```
?- append([a, b, c], [d, e], L).
L = [a, b, c, d, e].

?- append(X, [d, e], [a, b, c, d, e]).
X = [a, b, c]

?- append([a, b, c], Y, [a, b, c, d, e]).
Y = [d, e].
```

- Very different from imperative programming: input/output
- In Prolog: no clear notion of input and output
  - Just search for values that make the goal true

List operations

- Subset

```
subset([], S).
subset([H|T], S) :- member(H, S), subset(T, S).
```

- Reversing a list

```
reverse([], []).
reverse([H|T],R) :- reverse(T,R1), append(R1,[H],R).
```

- Permutations

```
permute([], []).
permute([H|T], P) :- permute(T, P1), insert(H, P1, P).
```

Unification

```
path(L, L).
path(L, M) :- link(L, X), path(X, M).

?- path(fortran, cplusplus).
```

- *Unification* is a type of pattern matching:
  - `L` unifies with `fortran`
  - `M` unifies with `cplusplus`

Unification

- Unification *rules*:
- a constant unifies with itself
- two structures unify if and only if:
  - have the same functor
  - have the same arity
  - corresponding arguments unify recursively
- a variable unifies with anything
  - if the other thing has a value, then the variable is instantiated
  - if the other thing is an uninstantiated variable, then the two variables are associated so that if either is given a value later, that value will be shared by both

Unification

- Equality (=) is *unifiability*:
  - The goal `=(A,B)` succeeds iff `A` and `B` can be unified
  - `A = B` – syntactic sugar
- Example:

```
?- a = a.
true.
?- a = b.
false.
?- foo(a,b) = foo(a,b).
true.
```

Unification

- Example:

```
?- X = a.
X = a.
?- foo(a,b) = foo(X,b).
X = a.
```

Arithmetic

- arithmetic operators – predicates
- `+(2,3)` - syntactic sugar `2+3`
- `+(2,3)` is a two-argument structure; does not unify with `5`

```
?- 1+1 = 2.
false.
```

- `is`: predicate that unifies first arg. with value of second arg.

```
?- is(X, 1+1).
X = 2.
?- X is 1+1.
X = 2.
```

More unification

- Substitution:
  - a function from variables to terms
  - Example: $\sigma = \{X \rightarrow [a,b],\ Y \rightarrow [a,b,c]\}$
- $T\sigma$ – the result of applying the substitution σ to the term T
  - $X\sigma = U$ if $X \rightarrow U$ is in σ, X otherwise
  - $(f(T_1, T_2,\ldots,T_n))\sigma = f(T_1\sigma, T_2\sigma,\ldots,T_n\sigma)$

- Example:

$\sigma = \{X \rightarrow [a,b],\ Y \rightarrow [a,b,c]\}$

$Y\sigma = [a,b,c]$

$Z\sigma = Z$

$append([], Y, Y)\sigma = append([], [a,b,c],[a,b,c])$

More unification

- A term U is an *instance* of T if $U=T\sigma$, for some substit. σ
- Two terms T_1 and T_2 *unify* if $T_1\sigma$ and $T_2\sigma$ are identical, for some σ; σ is called a *unifier* of T_1 and T_2
- σ is *the most general unifier* of T_1 and T_2 if, for any other unifier δ, $T_i\delta$ is an instance of $T_i\sigma$

- Example: $L = [a,b \mid X]$
- Unifiers:
  - $\sigma_1 = \{L \rightarrow [a,b \mid X_1], X \rightarrow X_1\}$
  - $\sigma_2 = \{L \rightarrow [a,b,c \mid X_2], X \rightarrow [c \mid X_2]\}$
  - $\sigma_3 = \{L \rightarrow [a,b,c,d \mid X_3], X \rightarrow [c, d \mid X_3]\}$
- σ_1 is the most general unifier

Control Algorithm

- Control algorithm
  - the way Prolog tries to satisfy a query
- Two decisions:
  - *goal order*: choose the leftmost subgoal
  - *rule order*: use the first applicable rule

Control Algorithm

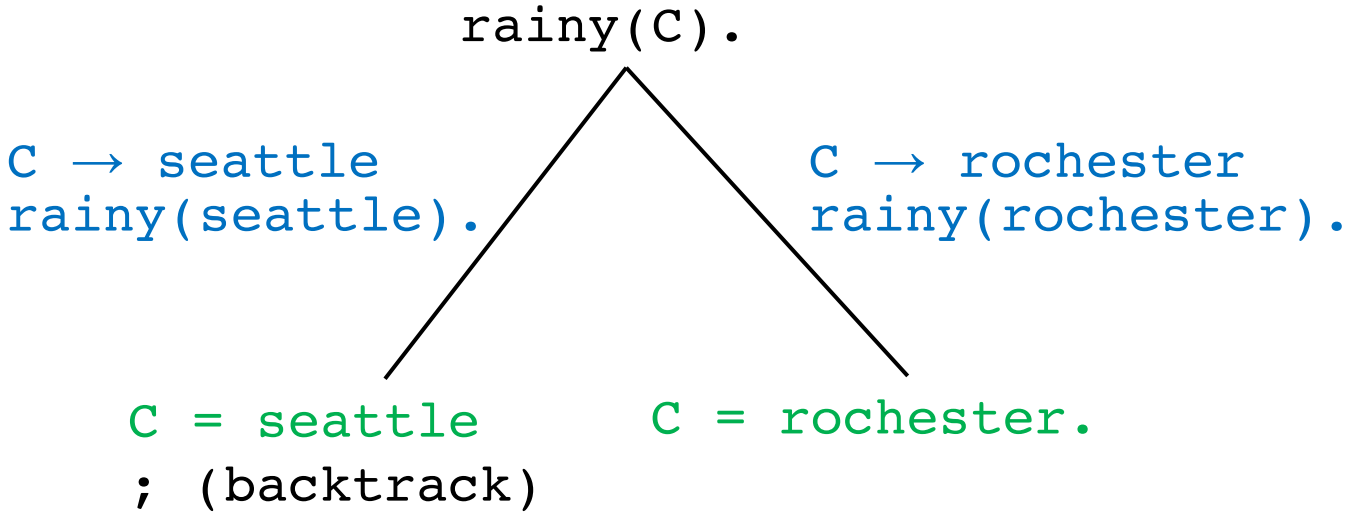
- Control algorithm

start with a query as the current goal
while (the current goal is nonempty) **do**
    choose the *leftmost subgoal*
    if (a rule applies to this subgoal) **then**
        select the *first applicable rule* not already used
        form a new current goal
    else
        if (at the root) **then**
            false
        else
            backtrack
true

Control Algorithm - Example

```
rainy(seattle).
rainy(rochester).

?- rainy(C).
C = seattle ;
C = rochester.
```

Prolog search tree:

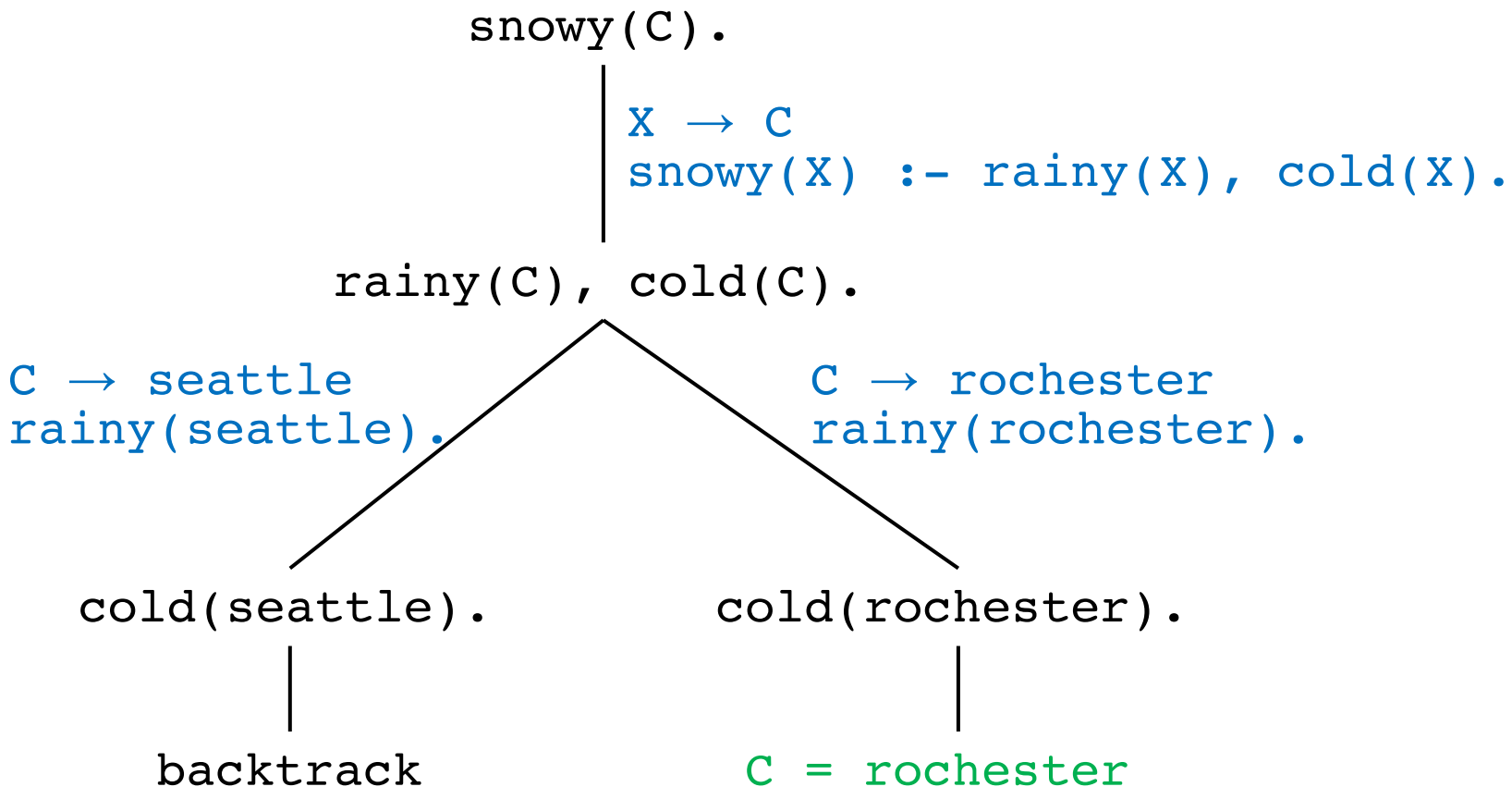


Control Algorithm - Example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).


?- snowy(C).
C = rochester.
```

Control Algorithm - Example

Prolog search tree:



Control Algorithm – details

start with a query as the current goal: $G_1, G_2, \ldots, G_k$ $(k \geq 0)$
while $(k > 0)$ **do** // the current goal is nonempty
  choose the *leftmost subgoal* G_1
  if (a rule applies to G_1) **then**
    select *first applicable rule* (not tried): A **:–** $B_1,\ldots, B_j$ $(j \geq 0)$
    let σ be *the most general unifier* of G_1 and A
    the current goal becomes: $B_1\sigma,\ldots, B_j\sigma, G_2\sigma, \ldots, G_k\sigma$
  else
    if (at the root) **then**
      false // tried all possibilities
    else
      backtrack // try something else
true // all goals have been satisfied

Control Algorithm - Example

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

prefix(P, L) :- append(P, _, L).
suffix(S, L) :- append(_, S, L).

?- suffix([a], L), prefix(L, [a, b, c]).
L = [a]    // that's the obvious solution

L = [a] ;  // if we ask for more solutions
           // we get an infinite computation
     // eventually aborting (out of stack)
```

Control Algorithm - Example

```
?- suffix([a], L), prefix(L, [a, b, c]).
```

```
L = [a] ; // infinite computation
```

- why the infinite computation?
- consider the first subgoal only:

```
?- suffix([a], L).
L = [a] ;
L = [_944, a] ;
L = [_944, _956, a] ;
L = [_944, _956, _968, a] ; ...
```

- infinitely many solutions, none (but the first) satisfying the second subgoal
- control checks an infinite subtree with no solutions

Control Algorithm - Example

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

prefix(P, L) :- append(P, _, L).
suffix(S, L) :- append(_, S, L).


?- suffix([b], L), prefix(L, [a, b, c]).
L = [a, b]   // that's the obvious solution

L = [a, b] ;// if we ask for more solutions
           // again, infinite computation
```

Goal order

- Changing the order of subgoals can change solutions:

```
?- suffix([a], L), prefix(L, [a, b, c]).
L = [a] ;
// infinite computation
```

- if we change the goal order, then no infinite computation:

```
?- prefix(L, [a, b, c]), suffix([a], L).
L = [a] ;
false.
```

Goal order

- The explanation is that the first subgoal now has finitely many solutions:

```
?- prefix(L, [a, b, c]).
L = [] ;
L = [a] ;
L = [a, b] ;
L = [a, b, c] ;
false.
```

Rule order

- Changing the order of rules can change solutions:

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

?- append(X, [c], Z).
X = [],
Z = [c] ;
X = [_576],
Z = [_576, c] ;
X = [_576, _588],
Z = [_576, _588, c] ;
X = [_576, _588, _600],
Z = [_576, _588, _600, c] ; ...
```

Rule order

- Changing the order of rules can change solutions:

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
append([], Y, Y).

?- append(X, [c], Z).
// infinite computation
```

Cuts

- ! – cut
- zero-argument predicate
- prevents backtracking, making computation more efficient
- can also implement a form of negation (we'll see later)
- General form of a cut:

$$P \text{ :- } Q_1, Q_2, \ldots, Q_{j-1}, \text{ !}, Q_{j+1}, \ldots, Q_k.$$

- Meaning: the control backtracks past

$$Q_{j-1}, Q_{j-2}, \ldots, Q_1, P$$

without considering any remaining rules for them

Cuts

- Example:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
prime_candidate(X) :- member(X,Candidates),prime(X).
```

- assume `prime(a)` is expensive to compute
- if `a` is a member of `Candidates` many times, this is slow
- solution:

```
member1(X, [X|_]) :- !.
member1(X, [_|T]) :- member1(X, T).
```

Cuts

```
?- member(a, [a,b,c,a,d,a]).
true ;
true ;
true ;
false.


?- member1(a, [a,b,c,a,d,a]).
true.
```

Negation as failure

- `not` – negation
- Definition:

```
not(X) :- X, !, fail.
not(_).
```

- `fail` always fails
- the first rule attempts to satisfy `X`
- if `X` succeeds, then `!` succeeds as well, then `fail` fails and `!` will prevent backtracking
- if `X` fails, then `not(X)` fails and, because the cut has not been reached, `not(_)` is tried and immediately succeeds

Negation as failure

- Example:

```
?- X=2, not(X=1).
X = 2.


?- not(X=1), X=2.
false.
```

12 Logic Languages

12.3 Theoretical Foundations

In mathematical logic, a *predicate* is a function that maps constants (atoms) or variables to the values true and false. *Predicate calculus* provides a notation and inference rules for constructing and reasoning about *propositions* (*statements*) composed of predicate applications, *operators*, and the *quantifiers* $\forall$ and $\exists$.[1] Operators include and ($\wedge$), or ($\vee$), not ($\neg$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$). Quantifiers are used to introduce bound variables in an appended proposition, much as λ introduces variables in the lambda calculus. The *universal* quantifier, $\forall$, indicates that the proposition is true for all values of the variable. The *existential* quantifier, $\exists$, indicates that the proposition is true for at least one value of the variable. Here are a few examples:

EXAMPLE 12.39
Propositions

$$\forall C[\mathsf{rainy}(C) \wedge \mathsf{cold}(C) \rightarrow \mathsf{snowy}(C)]$$

(For all cities C, if C is rainy and C is cold, then C is snowy.)

$$\forall A, \forall B[(\exists C[\mathsf{takes}(A, C) \wedge \mathsf{takes}(B, C)]) \rightarrow \mathsf{classmates}(A, B)]$$

(For all students A and B, if there exists a class C such that A takes C and B takes C, then A and B are classmates.)

$$\forall N[(N > 2) \rightarrow \neg(\exists A, \exists B, \exists C[A^N + B^N = C^N])]$$

(This is Fermat's last theorem.) ■

EXAMPLE 12.40
Different ways to say things

One of the interesting characteristics of predicate calculus is that there are many ways to say the same thing. For example,

1 Strictly speaking, what we are describing here is the *first-order* predicate calculus. There exist higher-order calculi in which predicates can be applied to predicates, not just to atoms and variables. Prolog allows the user to construct higher-order predicates using `call`; the formalization of such predicates is beyond the scope of this book.

$$\begin{aligned}(P_1 \rightarrow P_2) &\equiv (\neg P_1 \vee P_2)\\ (\neg \exists X[P(X)]) &\equiv (\forall X[\neg P(X)])\\ \neg(P_1 \wedge P_2) &\equiv (\neg P_1 \vee \neg P_2)\end{aligned}$$

This flexibility of expression tends to be handy for human beings, but it can be a nuisance for automatic theorem proving. Propositions are much easier to manipulate algorithmically if they are placed in some sort of *normal form*. One popular candidate is known as *clausal form*. We consider this form in the following section. ■

12.3.1 Clausal Form

As it turns out, clausal form is very closely related to the structure of Prolog programs: once we have a proposition in clausal form, it will be relatively easy to translate it into Prolog. We should note at the outset, however, that the translation is not perfect: there are aspects of predicate calculus that Prolog cannot capture, and there are aspects of Prolog (e.g., its imperative and database-manipulating features) that have no analogues in predicate calculus.

Clocksin and Mellish [CM03, Chap. 10] describe a five-step procedure (based heavily on an article by Martin Davis [Dav63]) to translate an arbitrary first-order predicate proposition into clausal form. We trace that procedure here.

EXAMPLE 12.41
Conversion to clausal form

In the first step, we eliminate implication and equivalence operators. As a concrete example, the proposition

$$\forall A[\neg \mathsf{student}(A) \rightarrow (\neg \mathsf{dorm\_resident}(A) \wedge \neg \exists B[\mathsf{takes}(A, B) \wedge \mathsf{class}(B)])]$$

would become

$$\forall A[\mathsf{student}(A) \vee (\neg \mathsf{dorm\_resident}(A) \wedge \neg \exists B[\mathsf{takes}(A, B) \wedge \mathsf{class}(B)])]$$

In the second step, we move negation inward so that the only negated items are individual terms (predicates applied to arguments):

$$\begin{aligned}&\forall A[\mathsf{student}(A) \vee (\neg \mathsf{dorm\_resident}(A) \wedge \forall B[\neg(\mathsf{takes}(A, B) \wedge \mathsf{class}(B))])]\\ \equiv\ &\forall A[\mathsf{student}(A) \vee (\neg \mathsf{dorm\_resident}(A) \wedge \forall B[\neg \mathsf{takes}(A, B) \vee \neg \mathsf{class}(B)])]\end{aligned}$$

In the third step, we use a technique known as Skolemization (due to logician Thoralf Skolem) to eliminate existential quantifiers. We will consider this technique further in Section C-12.3.3. Our example has no existential quantifiers at this stage, so we proceed.

In the fourth step, we move all universal quantifiers to the outside of the proposition (in the absence of naming conflicts, this does not change the proposition's

meaning). We then adopt the convention that all variables are universally quantified, and drop the explicit quantifiers:

$$\text{student}(A) \vee (\neg\text{dorm\_resident}(A) \wedge (\neg\text{takes}(A, B) \vee \neg\text{class}(B)))$$

Finally, in the fifth step, we use the distributive, associative, and commutative rules of Boolean algebra to convert the proposition to *conjunctive normal form*, in which the operators $\wedge$ and $\vee$ are nested no more than two levels deep, with $\wedge$ on the outside and $\vee$ on the inside:

$$(\text{student}(A) \vee \neg\text{dorm\_resident}(A)) \wedge (\text{student}(A) \vee \neg\text{takes}(A, B) \vee \neg\text{class}(B))$$

Our proposition is now in clausal form. Specifically, it is in conjunctive normal form, with negation only of individual terms, with no existential quantifiers, and with implied universal quantifiers for all variables (i.e., for all names that are neither constants nor predicates). The clauses are the items at the outer level: the things that are and-ed together. ■

EXAMPLE 12.42
Conversion to Prolog

To translate the proposition to Prolog, we convert each logical clause to a Prolog fact or rule. Within each clause, we use commutativity to move the negated terms to the right and the non-negated terms to the left (our example is already in this form). We then note that we can recast the disjunctions as implications:

$$\begin{aligned} & (\text{student}(A) \leftarrow \neg(\neg\text{dorm\_resident}(A))) \\ & \quad \wedge (\text{student}(A) \leftarrow \neg(\neg\text{takes}(A, B) \vee \neg\text{class}(B))) \\ \equiv \quad & (\text{student}(A) \leftarrow \text{dorm\_resident}(A)) \\ & \quad \wedge (\text{student}(A) \leftarrow (\text{takes}(A, B) \wedge \text{class}(B))) \end{aligned}$$

These are Horn clauses. The translation to Prolog is trivial:

```
student(A) :- dorm_resident(A).
student(A) :- takes(A, B), class(B).
```

■

12.3.2 Limitations

We claimed at the beginning of Section 12.1 that Horn clauses could be used to capture most, though not all, of first-order predicate calculus. So what is it missing? What can go wrong in the translation? The answer has to do with the number of non-negated terms in each clause. If a clause has more than one, then if we attempt to cast it as an implication there will be a disjunction on the left-hand side of the $\leftarrow$ symbol, something that isn't allowed in a Horn clause. Similarly, if we end up with no non-negated terms, then the result is a headless Horn clause, something that Prolog allows only as a query, not as an element of the database.

EXAMPLE 12.43
Disjunctive left-hand side

As an example of a disjunctive head, consider the statement "every living thing is an animal or a plant." In clausal form, we can capture this as

$$\text{animal}(X) \vee \text{plant}(X) \vee \neg\text{living}(X)$$

or equivalently

$$\text{animal}(X) \vee \text{plant}(X) \leftarrow \text{living}(X)$$

Because we are restricted to a single term on the left-hand side of a rule, the closest we can come to this in Prolog is

```
animal(X) :- living(X), \+(plant(X)).
plant(X) :- living(X), \+(animal(X)).
```

But this is not the same, because Prolog's \+ indicates inability to prove, not falsehood. ■

EXAMPLE 12.44
Empty left-hand side

As an example of an empty head, consider Fermat's last theorem (Example C-12.39). Abstracting out the math, we might write

$$\forall N[\text{big}(N) \rightarrow \neg(\exists A, \exists B, \exists C[\text{works}(A, B, C, N)])]$$

which becomes the following in clausal form:

$$\neg\text{big}(N) \vee \neg\text{works}(A, B, C, N)$$

We can couch this as a Prolog query:

```
?- big(N), works(A, B, C, N).
```

(a query that will never terminate), but we cannot express it as a fact or a rule. ■

EXAMPLE 12.45
Theorem proving as a search for contradiction

The careful reader may have noticed that facts are entered on the left-hand side of an (implied) Prolog :- sign:

```
rainy(rochester).
```

while queries are entered on the right:

```
?- rainy(rochester).
```

The former means

$$\text{rainy}(\text{rochester}) \leftarrow \text{true}$$

The latter means

$$\text{false} \leftarrow \text{rainy}(\text{rochester})$$

If we apply resolution to these two propositions, we end up with the contradiction

$$\text{false} \leftarrow \text{true}$$

This observation suggests a mechanism for automated theorem proving: if we are given a collection of axioms and we want to prove a theorem, we temporarily add the *negation* of the theorem to the database and then attempt, through a series of resolution operations, to obtain a contradiction. ■

12.3.3 Skolemization

EXAMPLE 12.46
Skolem constants

In Example C-12.41 we were able to translate a proposition from predicate calculus into clausal form without worrying about existential quantifiers. But what about a statement like this one:

$$\exists X[\text{takes}(X, \text{cs254}) \wedge \text{class\_year}(X, 2)]$$

(There is at least one sophomore in cs254.) To get rid of the existential quantifier, we can introduce a *Skolem constant* `x`:

$$\text{takes}(\text{x}, \text{cs254}), \text{class\_year}(\text{x}, 2)$$

The mathematical justification for this change is based on something called the *axiom of choice*; intuitively, we say that if there exists an X that makes the statement true, then we can simply pick one, name it `x`, and proceed. (If there does not exist an X that makes the statement true, then we can choose some arbitrary `x`, and the statement will still be false.) It is worth noting that Skolem constants are not necessarily distinct; it is quite possible, for example, for `x` to name the same student as some other constant `y` that represents a sophomore in `his201`. ■

EXAMPLE 12.47
Skolem functions

Sometimes we can replace an existentially quantified variable with an arbitrary constant `x`. Often, however, we are constrained by some surrounding universal quantifier. Consider the following example:

$$\forall X[\neg \text{dorm\_resident}(X) \vee \exists A[\text{campus\_address\_of}(X, A)]]$$

(Every dorm resident has a campus address.) To get rid of the existential quantifier, we must choose an address for X. Since we don't know who X is (this is a general statement about all dorm residents), we must choose an address that *depends on X*:

$$\forall X[\neg \text{dorm\_resident}(X) \vee \text{campus\_address\_of}(X, \text{f}(X))]$$

Here `f` is a *Skolem function*. If we used a simple Skolem constant instead, we'd be saying that there exists some single address shared by all dorm residents. ■

EXAMPLE 12.48
Limitations of Skolemization

Whether Skolemization results in a clausal form that we can translate into Prolog depends on whether we need to know what the constant is. If we are using predicates `takes` and `class_year`, and we wish to assert as a fact that there is a sophomore in `cs254`, we can write

```
takes(the_distinguished_sophomore_in_254, cs254).
class_year(the_distinguished_sophomore_in_254, 2).
```

Similarly, we can assert that every dorm resident has a campus address by writing

```
campus_address_of(X, the_dorm_address_of(X)) :- dorm_resident(X).
```

Now we can search for classes with sophomores in them:

```
sophomore_class(C) :- takes(X, C), class_year(X, 2).
?- sophomore_class(C).
C = cs254
```

and we can search for people with campus addresses:

```
has_campus_address(X) :- campus_address_of(X, Y).
dorm_resident(li_ying).
?- has_campus_address(X).
X = li_ying
```

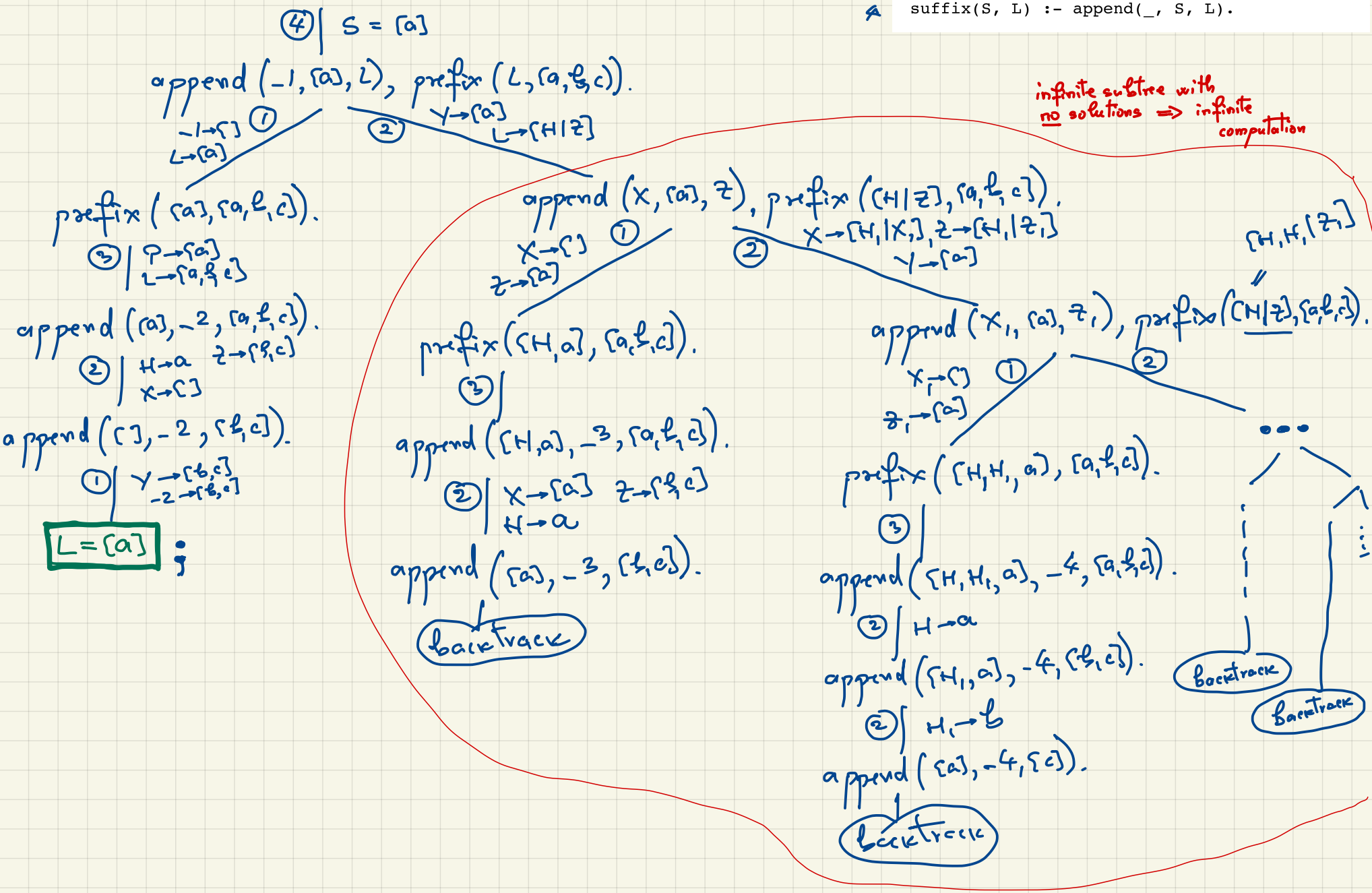
Unfortunately, we won't be able to identify a sophomore in `cs254` by name, nor will we be able to identify the address of `li_ying`. ■

✓ CHECK YOUR UNDERSTANDING

15. Define the notion of *clausal form* in predicate calculus.
16. Outline the procedure to convert an arbitrary predicate calculus statement into clausal form.
17. Characterize the statements in clausal form that cannot be captured in Prolog.
18. What is *Skolemization*? Explain the difference between Skolem constants and Skolem functions.
19. Under what circumstances may Skolemization fail to produce a clausal form that can be captured usefully in Prolog?

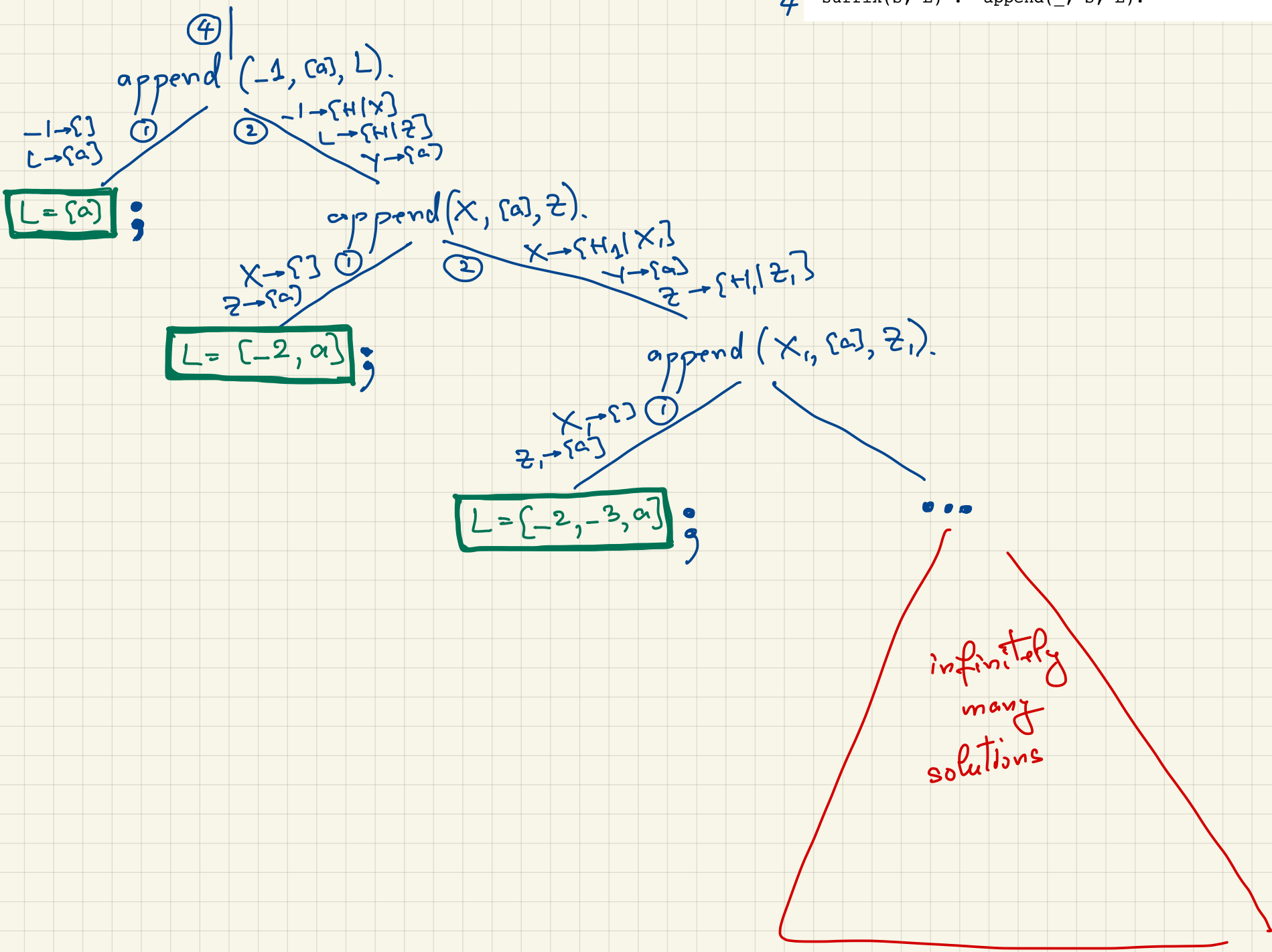
```
1  append([], Y, Y).
2  append([H|X], Y, [H|Z]) :- append(X, Y, Z).

3  prefix(P, L) :- append(P, _, L).
4  suffix(S, L) :- append(_, S, L).
```

```
?- suffix([a], L), prefix(L, [a, b, c]).
```



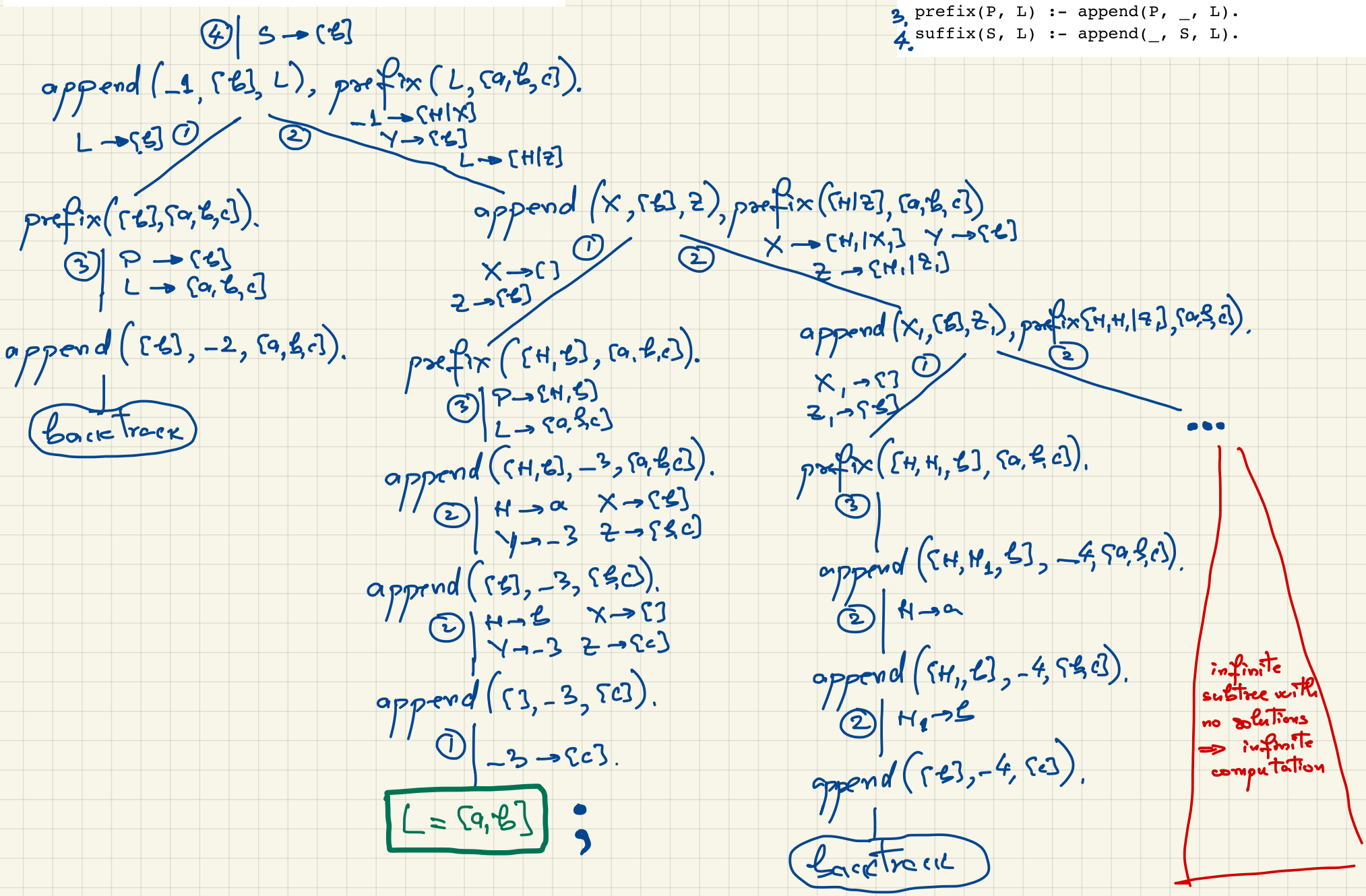
```
1  append([], Y, Y).
2  append([H|X], Y, [H|Z]) :- append(X, Y, Z).

3  prefix(P, L) :- append(P, _, L).
4  suffix(S, L) :- append(_, S, L).
```

```
?- suffix([a], L).
```



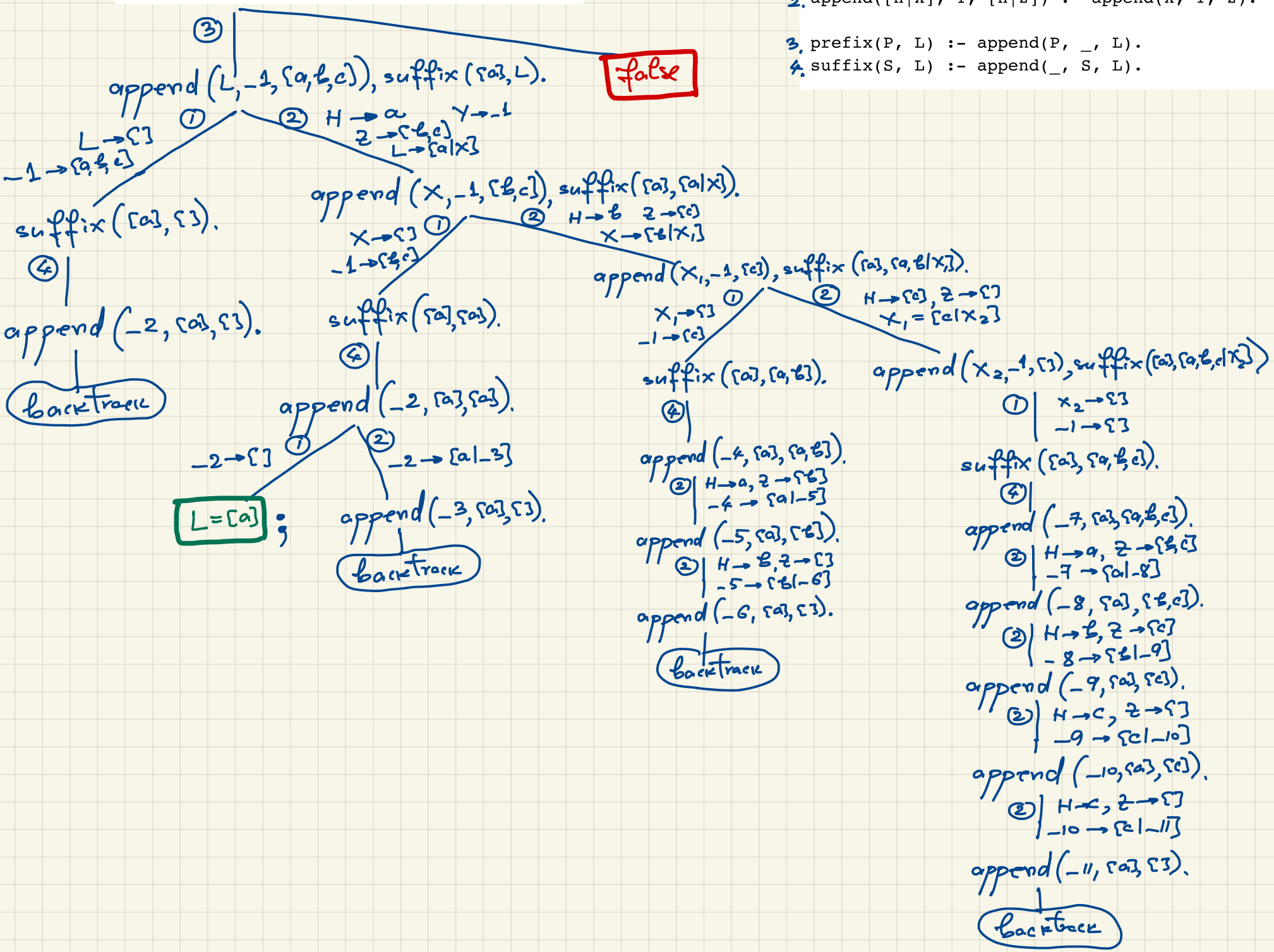
```
?- suffix([b], L), prefix(L, [a, b, c]).
```

```
1. append([], Y, Y).
2. append([H|X], Y, [H|Z]) :- append(X, Y, Z).

3. prefix(P, L) :- append(P, _, L).
4. suffix(S, L) :- append(_, S, L).
```



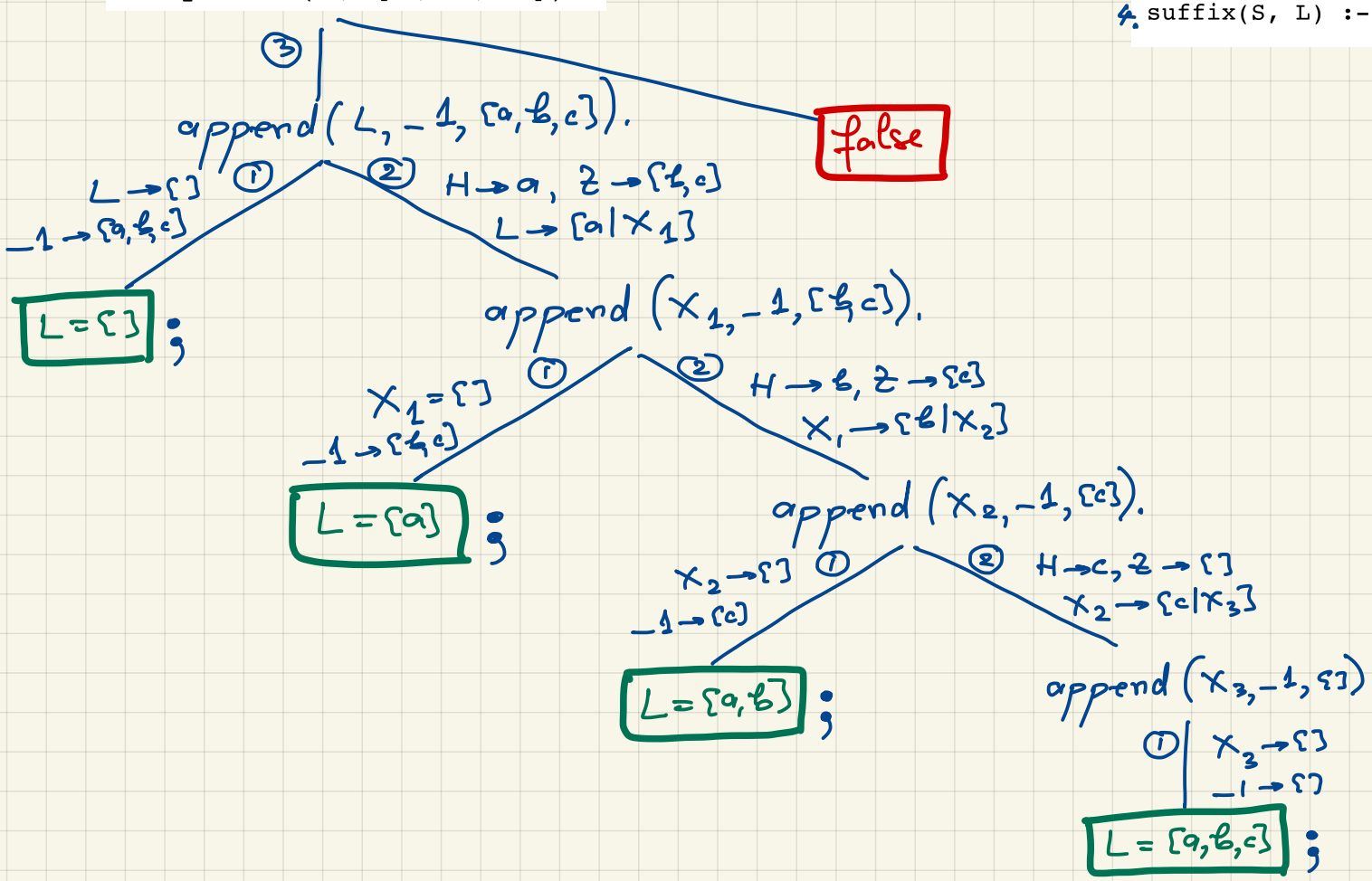
```
?- prefix(L, [a, b, c]), suffix([a], L).
```

```
1. append([], Y, Y).
2. append([H|X], Y, [H|Z]) :- append(X, Y, Z).

3. prefix(P, L) :- append(P, _, L).
4. suffix(S, L) :- append(_, S, L).
```

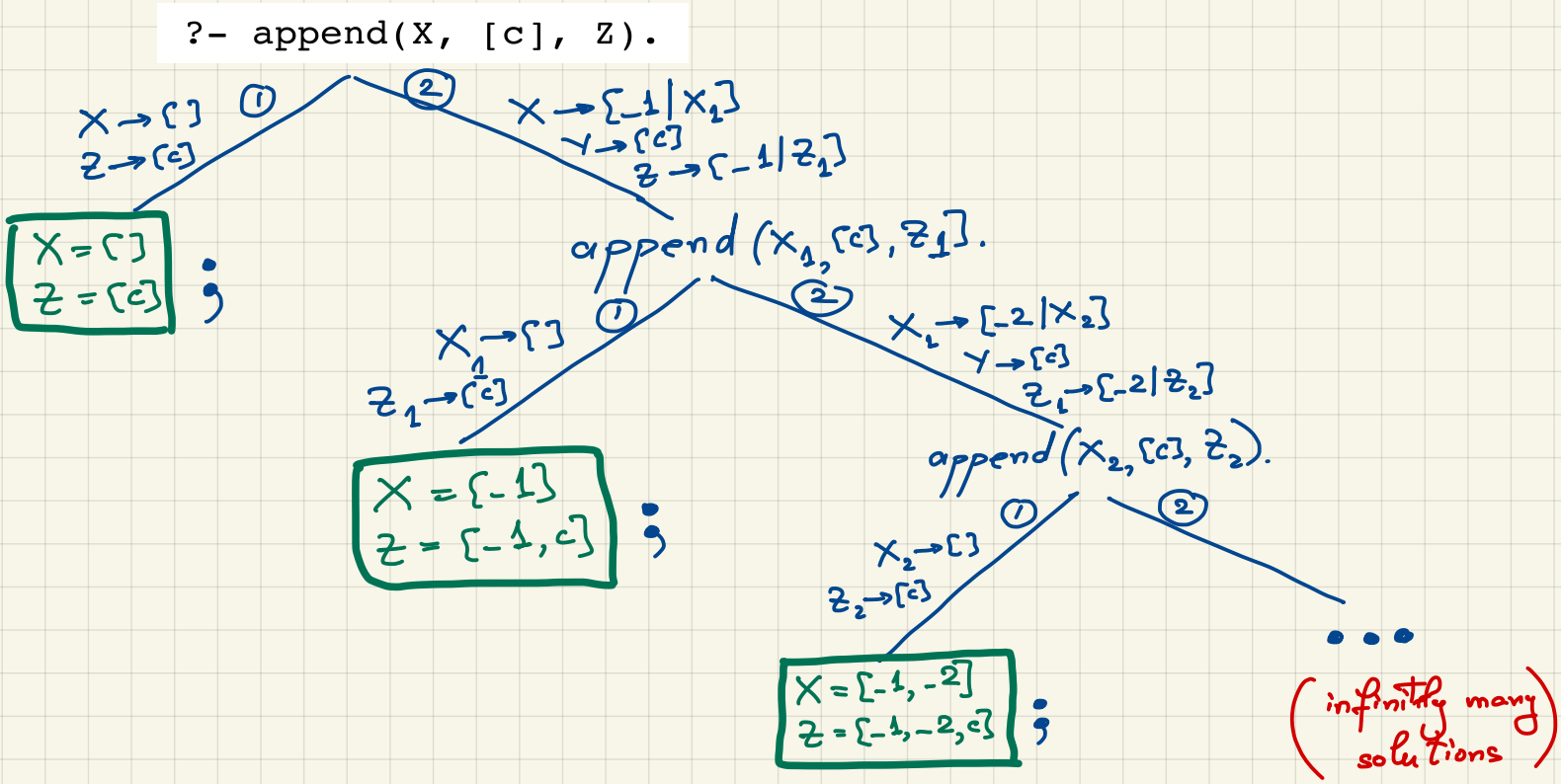


```
1. append([], Y, Y).
2. append([H|X], Y, [H|Z]) :- append(X, Y, Z).

3. prefix(P, L) :- append(P, _, L).
4. suffix(S, L) :- append(_, S, L).
```

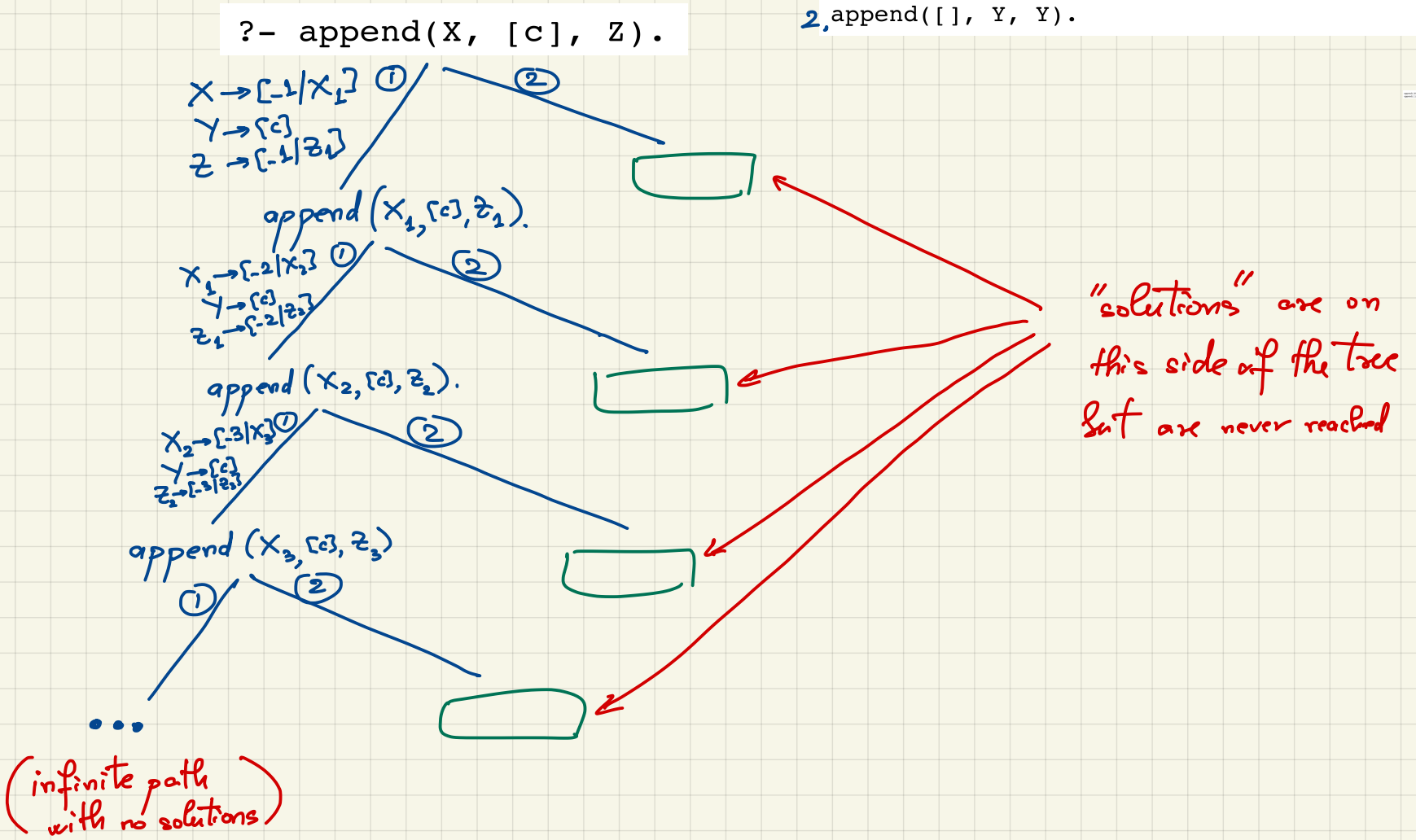
```
?- prefix(L, [a, b, c]).
```



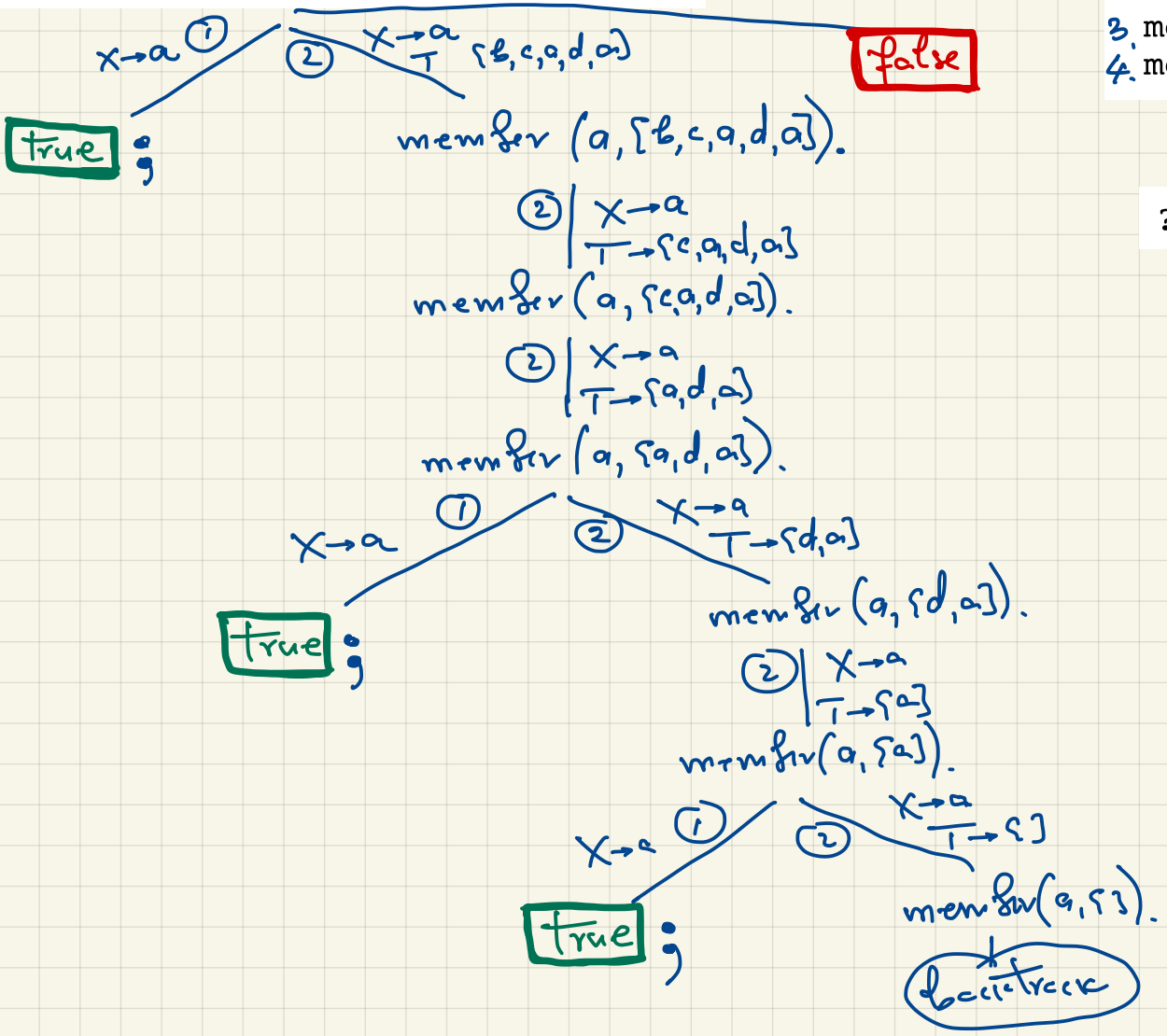
```
1. append([], Y, Y).
2. append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```



```
1. append([H|X], Y, [H|Z]) :- append(X, Y, Z).
2. append([], Y, Y).
```

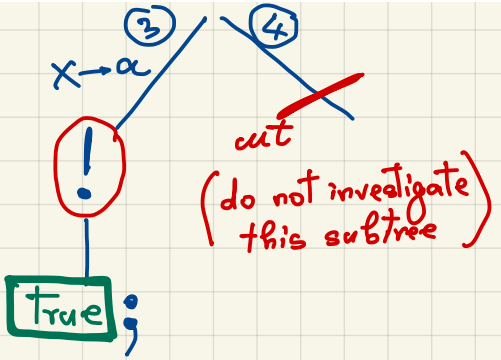


```
?- member(a, [a,b,c,a,d,a]).
```



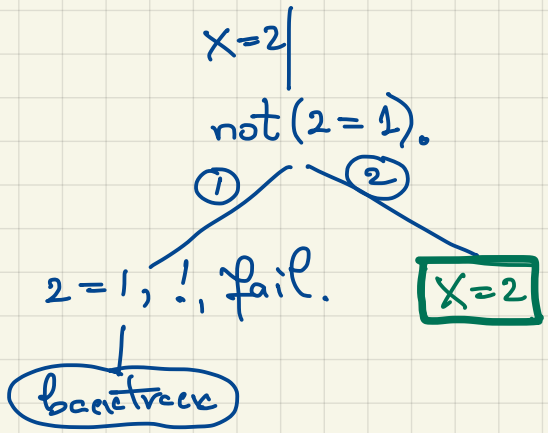

```
1. member(X, [X|_]).
2. member(X, [_|T]) :- member(X, T).

3. member1(X, [X|_]) :- !.
4. member1(X, [_|T]) :- member1(X, T).
```

```
?- member1(a, [a,b,c,a,d,a]).
```



```
1. not(X) :- X, !, fail.
2. not(_).
```

```
?- X=2, not(X=1).
```




```
?- not(X=1), X=2.
```