CS342b – winter 2006

PROGRAMMING LANGUAGES

Lucian Ilie

1 Introduction

- why study (concepts of) programming languages
- classification of languages
- programming domains
- evaluation criteria
- levels of programming languages
- compilation and interpretation

1.1 General questions

1.1.1 Why are there so many programming languages?

- evolution – we've learned better ways of doing things over time
- socio-economic factors – proprietary interests, commercial advantage
- orientation toward special purposes
- orientation toward special hardware
- diverse ideas about what is pleasant to use

1.1.2 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)
- easy to implement (BASIC, Forth)
- possible to compile to very good (fast/small) code (Fortran)
- backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)
- wide dissemination at minimal cost (Pascal, Turing, Java)

1.1.3 Why do we have programming languages? – what is a language for?

- way of thinking
  – way of expressing algorithms
  – from the user's point of view
- abstraction of virtual machine
  – way of specifying what you want the hardware to do without getting down into the bits
  – from the implementor's point of view

Knuth: "Computer Programming is the art of explaining to another human being what you want the computer to do."

1.1.4 Why study programming languages?

- help you choose a language

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

- make it easier to learn new languages

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

- help you make better use of whatever language you use

- understand obscure features
  – in C, unions, arrays and pointers, separate compilation, variable number of arguments, etc.
  – in Common Lisp, help you understand first-class functions/closures,

- understand implementation costs – choose between alternative ways of doing things, based on knowledge of what will be done underneath:
  – use simple arithmetic equalities (use x*x instead of x**2)
  – use C pointers or Pascal "with" statement to factor address calculations
  – avoid call by value with large data items in Pascal
  – avoid the use of call by name in Algol 60

- figure out how to do things in languages that don't support them explicitly
  – lack of suitable control structures in Fortran IV – use comments and programmer discipline for control structures
  – lack of recursion in Fortran 77, CSP, etc. – write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)
  – lack of named constants and enumerations in Fortran – use variables that are initialized once, then never changed
  – lack of modules in C and Pascal – use comments and programmer discipline

- prepare for further study in language design or implementation

1.2 Classification of languages

- **declarative** – what to do?
  - ▸ functional (Scheme, ML, pure Lisp, Haskell, FP)
    - – based on lambda calculus
    - – computational model based on recursive definition of functions
    - – a program = a function from inputs to outputs
  - ▸ logic, constraint-based (Prolog, VisiCalc, RPG)
    - – based on propositional logic
    - – computation is an attempt to find values that satisfy certain specified relationships using goal-directed search through a list of logical rules

- **imperative** – how to do?
  - ▸ von Neumann (Fortran, Pascal, Basic, C, ...)
    - – computing via side effects
    - – statements that influence subsequent computation by changing the values of memory
  - ▸ object-oriented (Smalltalk, Eiffel, C++, Java)
    - – computation = interactions among objects

1.3 Programming domains

- scientific applications
  - simple data structures, large computations (Fortran, Algol 60)
- bussines applications
  - elaborate reports, storing numbers, arithmetics (Cobol)
- artificial intelligence
  - symbolic computations (Lisp, Prolog)
- systems programming
  - execution efficiency (C)
- special purpose languages

1.4 Evaluation of programming languages

- programming
  - development - creation and testing of a program
  - maintenance - correction and changes after

- good program
  - correct
  - easy to read / understand
  - easy to modify

1.4.1 Criteria

- readability / writability
  - simplicity, orthogonality, control statements (goto), data types and structures (bool, struct), syntax, support for abstraction
- reliability – type checking, exception handling
- portability – standardization, system independence
- generality – wide range of applications
- cost – program development, maintenance, execution

1.4.2 Language design trade-offs

- compilation cost vs execution speed
    - optimization requires time
- reliability vs execution cost
    - ex: index range checking
- readability vs writability
- flexibility vs safety

1.5 Brief history of programming languages

- Zuse's Pankalkül - 1945
  - first high level programming language
  - never implemented
  - unpublished until 1972
- Minimal hardware programming: pseudocodes
  - late 1940's and early 1950's
  - machine code - tedious and error-prone
  - somewhat higher-level languages - interpreted
  - Short Code, UNIVAC "compiling" system
- FORTRAN (FORmula TRANslating system)
  - came with IBM 704
  - first language to be compiled
  - IBM 704 - floating-point instructions in hardware
    - end of interpretation
  - FORTRAN 0 - 1954
  - FORTRAN I - 1956 - already very efficient

- I/O, subroutines, **if** selection, **do** loop
- FORTRAN II - 1958
  - independently compiled subroutines
- FORTRAN III - never widely distributed
- FORTRAN IV - 1960-62
- FORTRAN 77 = standard FORTRAN IV
  - type declarations, subprograms as parameters
- FORTRAN 90 - ANSI, 1992 - latest version
  - new statements, recursiveness, modules, dynamical allocation of arrays, etc.

- Functional programming: LISP (LISt-Processing language)
  - interest in AI - mid 1950's
    - the need for computers able to process symbolic data in linked lists
  - IBM became interested in theorem proving
    - LISP - 1958, John McCarthy (MIT)
  - LISP dominated AI applications for 25 years
  - two descendants widely used now:

  - Scheme and Common Lisp

- ALGOL 60 (ALGOrithmic Language)
  - effort to design a universal language
  - GAMM and ACM against the domination of IBM
  - ALGOL 58 - generalized FORTRAN
    - added: the concept of data type, compound statement, etc.
  - ALGOL 60 - block structure, pass by value/name, recursion, etc.
  - became (for 20 years) the sole language for publishing algorithms
  - every imperative language since 1960 owes something to ALGOL 60
  - never achieved widespread use
  - BNF notation introduced - now is most commonly used technique for describing syntax

- COBOL - 1960
  - CBL (Common Bussines Language)
  - designed for bussines computing
  - mandated by U.S. Department of Defense
  - widespread use but little respect

- BASIC (Beginner's All-purpose Symbolic Instruction Code)

- designed for introducing programming concepts to non-science students
- most important aspect: designed with the idea that terminals would be the method of for accessing computer

- PL/I - 1965, IBM
  - developed for a broad spectrum of application areas
  - included the best of ALGOL 60, FORTRAN IV, and COBOL 60
  - too large to be widely used

- SIMULA 67
  - introduced by two Norwegians
  - little impact
  - important - introduced coroutines and classes

- Pascal - 1971
  - descendent of ALGOL 60 - designed by Wirth
  - designed for teaching programming

- C - 1972
  - designed by Dennis Ritchie
  - descendant of ALGOL 68 (but not only)

  - originally designed for system programming
  - contributed little to previous knowledge but is has been very successful
  - appropriate for many application areas
  - lack of complete type checking - liked and disliked

- PROLOG (PROgramming LOGic) - early 1970's
  - based on predicate calculus and resolution
  - attempt to replace imperative languages - not successful so far

- Ada (Augusta Ada Byron - the world's first programmer) - 1970's-80's
  - Department of Defense mandated
  - history's largest effort
  - included most of the concepts in the 1970's
  - too large

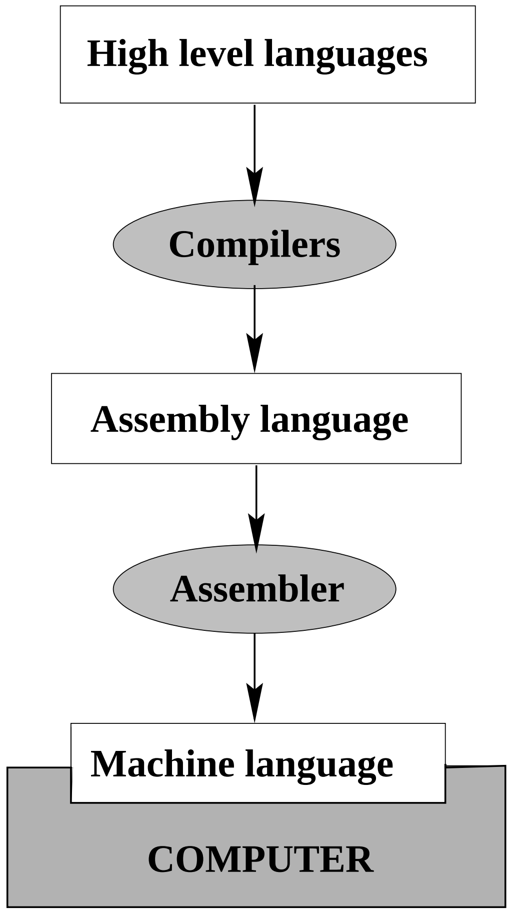
- Smalltalk
  - Alan Kay - first ideas - 1969; versions - 1972, 1980
  - object-oriented programming
  - all program units are objects which communicate with each other
  - promoted also the windowing system

- C++ - 1980's
  - Bjarne Stroustrup
  - combines imperative and object-oriented features
  - is very (and increasingly) popular
- Java - 1990's
  - provides much of the power and flexibility of C++
  - is smaller, simpler, and safer
  - suitable for web programming

1.6 Development of languages

- 1950's
  - machine language
- late 50's and early 60's
  - Fortran, Algol, Cobol
- 60's
  - Basic, Pascal
- late 60's and early 70's
  - abstract data type, concurrent programming
- 70's
  - C, object oriented programming, logic and functional programming
- 80's
  - C++, parallel programming
- 90's
  - Java, web programming

1.7 Levels of programming languages



- **machine language** - unintelligible

example

```
000000101011110010100
00000101111111001000
00000110011101010000
```

[Goldstine (1972): "While this enumeration is virtually unintelligible to a human, it is exactly what the machine understands."]

- **assembly language**
  - low level
  - names and symbols in place of the codes for machine

example

```
1 : $M[0] := 0$
2 : $read(M[1])$
3 : if $M[1] \geq 0$ then goto 5
4 : goto 7
5 : $M[3] := M[0] - M[1]$
6 : if $M[3] \geq 0$ then goto 16
7 : $writeln(M[1])$
⋮   ⋮
⋮   ⋮
```

- high level language

- readable familiar notations
- portability = machine independence
  (general purpose = wide range of problems)
- availability of program libraries
- consistency checks during implementation that can detect errors

[Ritchie (1973): "Although a comprehensive study of the space and time inflation due to the use of C might be interesting, it would be somewhat irrelevant, in that no matter what the results turned out to be, they would not cause us to start writing assembly language."]

1.7.1 Gotos vs structured stmts vs procedures vs classes

```
    read(x);
2:  if $x = 0$ then
       goto 8;
    writeln(x);
4:  read(next);
    if $next = x$ then
       goto 4;
    $x := next$;
    goto 2;
8:  ;
```

(a) goto

```
read(x);
while $x \neq 0$ do
begin
    writeln(x);
    repeat
        read(next)
    until $next \neq x$;
    $x := next$;
end;
```

(b) structured stmts

```
readentry;(e)
while not endentry(e) do
begin
    writeentry(e);
    repeat
        readentry(f)
    until not equalentry(e,f);
    copyentry(e,f);
end
```

(c) procedures

```
Entry e, f;
e.read();
while not e.end() do
begin
    e.write();
    do
        f.read();
    while $e = f$;
    $e := f$;
end
```

(d) classes

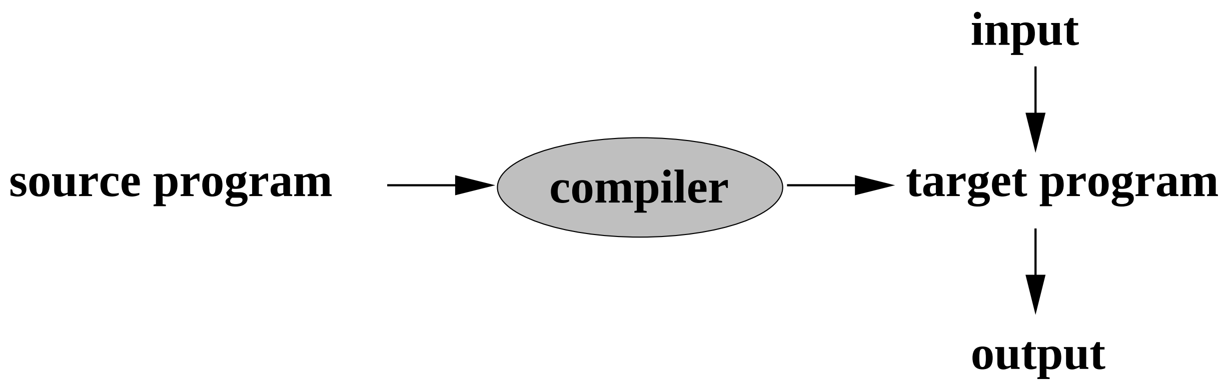
1.8 Compilation and interpretation

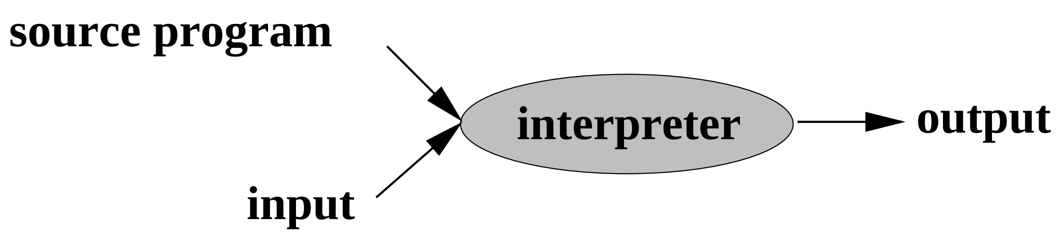
- compilation

- the compiler translates into a target language and then goes away
- the target program is the place of control at execution time

- interpretation

- the interpreter stays around at execution time
- it is the place of control during execution





Compilation

- programs - reside in memory and executed in CPU
- execution of machine code program:

fetch-execute cycle

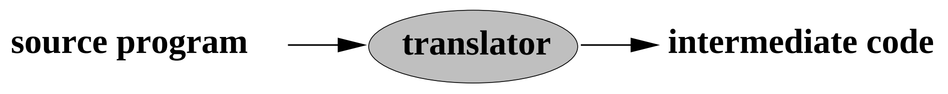
```
repeat forever
        fetch the instruction pointed by program counter
        increment program counter to point next instruction
        decode the instruction
        execute the instruction
end repeat
```

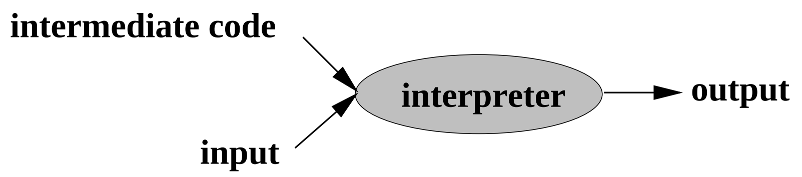
- von Neumann bottleneck - the connection between memory and CPU - instructions are executed faster than moved

Interpretation

- fetch-execute cycle with high-level statements (no translation)
- advantage - easy debugging
- disadvantage - execution 10 to 100 times slower

Hybrid systems - most systems





example

Java: intermediate code = byte code - portability
- some systems: byte code → machine code (faster execution)

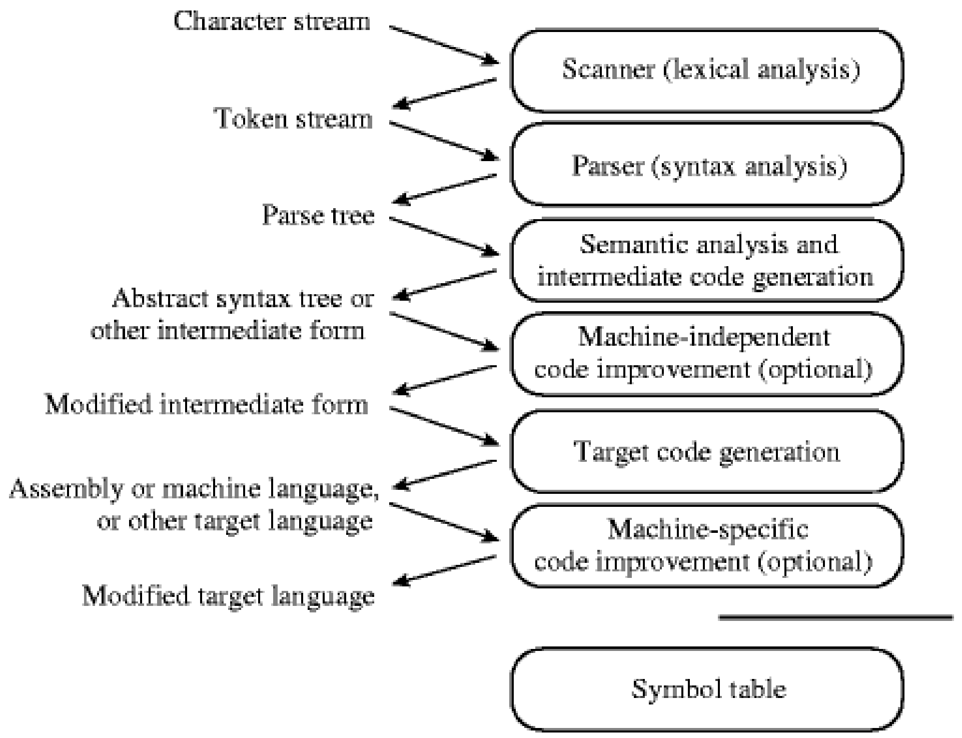
example

sometimes - both compiler and interpreter
- interpreter - develop and debug
- compiler - at the end to increase execution speed

1.8.1 Compilation phases

- **front end** (analyze source)
  - scanning
  - parsing
  - semantic analysis

- **back end** (build target)
  - intermediate code generation
  - code improvement
  - target code generation


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Assembly or machine language, or other target language
Machine-specific code improvement (optional)
Modified target language
Symbol table

2 Syntax

- lexical syntax
- regular grammars
- finite automata
- scanning
- concrete syntax
- context-free grammars
- BNF form
- parsing – top-down and bottom-up

- **syntax** - describes the form of programs
- **semantics** - describe the meaning of programs
  - more difficult
  - the state of the art: formal syntax and informal semantics

2.1 Lexical syntax

tokens - basic building blocks of programs

- the source program (stream of characters) is read from left to right and grouped into tokens = sequences of characters having a collective meaning

examples of tokens: keywords, identifiers, numbers, punctuation marks, etc.

example

the character sequence

$$b * b - 4 * a$$

is represented by the token sequence

$$\mathbf{name}_b * \mathbf{name}_b - \mathbf{number}_4 * \mathbf{name}_a$$

lexical syntax - specifies the way tokens should be written

2.1.1 Specifying lexical syntax

regular grammar – the tool for specifying lexical syntax

- a set of **nonterminals** (symbols, letters)
- a set of **terminals** (symbols, letters)
- a nonterminal chosen as the **starting symbol**
- a set of **productions** of the form:
  $A ::= aB$, or $A ::= a$ s.t. A, B are nonterminals and a is terminal

example 1

nonterminals: S, A
terminals: $0, 1$
starting symbol: S
productions: $S ::= 0S \mid 0A$ $\quad$ $' ::='$ means "can be"
$A ::= 1A \mid 1$ $\quad$ $' \mid '$ means "or"

derivation: $S \Rightarrow 0S \Rightarrow 00S \Rightarrow 000A \Rightarrow 0001A \Rightarrow 00011$

generated words = words derived (obtained) from the starting symbol by applying productions

- for example 1: the set of generated words (**language**) is

$$\{0^n 1^m \mid n, m \geq 1\} = \{\underbrace{00\ldots0}_{n}\underbrace{11\ldots1}_{m} \mid n, m \geq 1\}$$

example 2

nonterminals: **integer**
terminals: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
starting symbol: **integer**
productions: $\textbf{integer} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$\textbf{integer} ::= 0\ \textbf{integer} \mid 1\ \textbf{integer} \mid \cdots \mid 9\ \textbf{integer}$

derivation: $\textbf{integer} \Rightarrow 7\ \textbf{integer} \Rightarrow 73\ \textbf{integer} \Rightarrow 738$

example 3

nonterminals: **letter**, **digit**, **name**, **word**

terminals: $a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9$

starting symbol: **name**

productions:

$$\begin{aligned} \textbf{letter} &::= a \mid \cdots \mid z \mid A \mid \cdots \mid Z \\ \textbf{digit} &::= 0 \mid 1 \mid \cdots \mid 9 \\ \textbf{word} &::= \textbf{letter} \mid \textbf{digit} \\ &\quad \mid \textbf{letter word} \mid \textbf{digit word} \\ \textbf{name} &::= \textbf{letter word} \mid \textbf{letter} \end{aligned}$$

derivation:

$$\textbf{name} \Rightarrow \textbf{letter word} \Rightarrow \text{X}\ \textbf{word} \Rightarrow \text{X}\ \textbf{digit} \Rightarrow \text{X5}$$

example 4

C comments

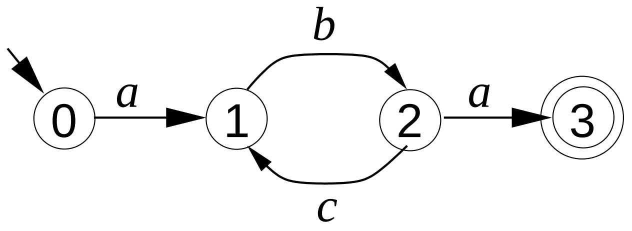
2.1.2 Recognizing lexical syntax

- this is **lexical analysis** or **scanning**
- the **scanners** are finite automata - machine counterpart of regular grammars
  regular grammars - generate words
  finite automata - accept words

finite automaton:

- an **alphabet**
- a set of states
- a state chosen as the **initial state**
- a subset of **final states**
- a **transition mapping** which maps pairs (state, letter) to states

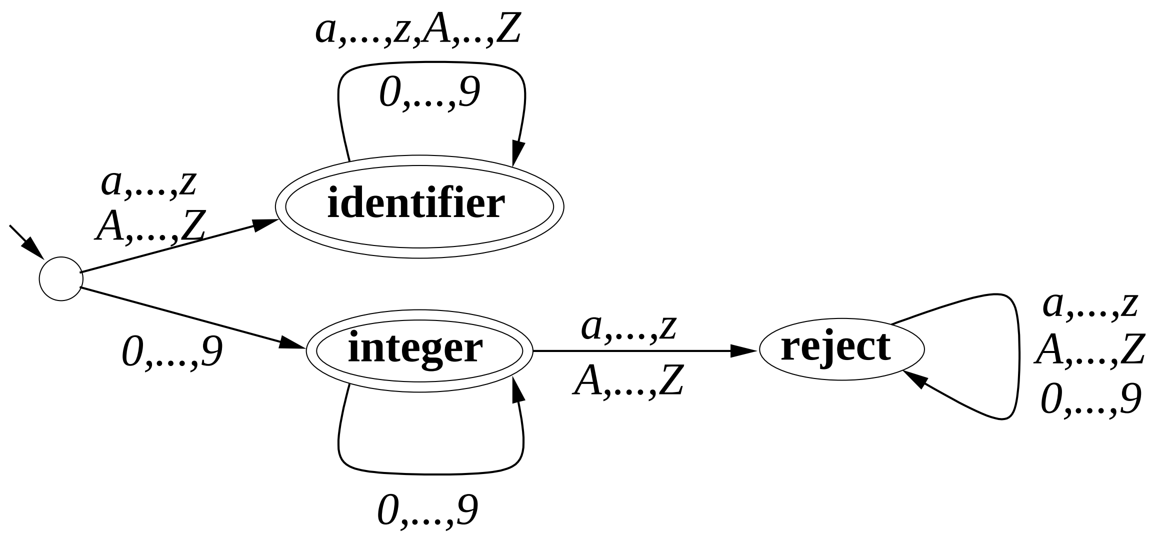
example 1


$$\begin{tikzcd} \arrow[r] & 0 \arrow[r, "a"] & 1 \arrow[r, bend left, "b"] & 2 \arrow[l, bend left, "c"] \arrow[r, "a"] & \boxed{3} \end{tikzcd}$$

accepted words (language) = words that take the automaton from the initial state to a final one

- the automaton in example 1 accepts the language: $ab(cb)^*a$

example 2



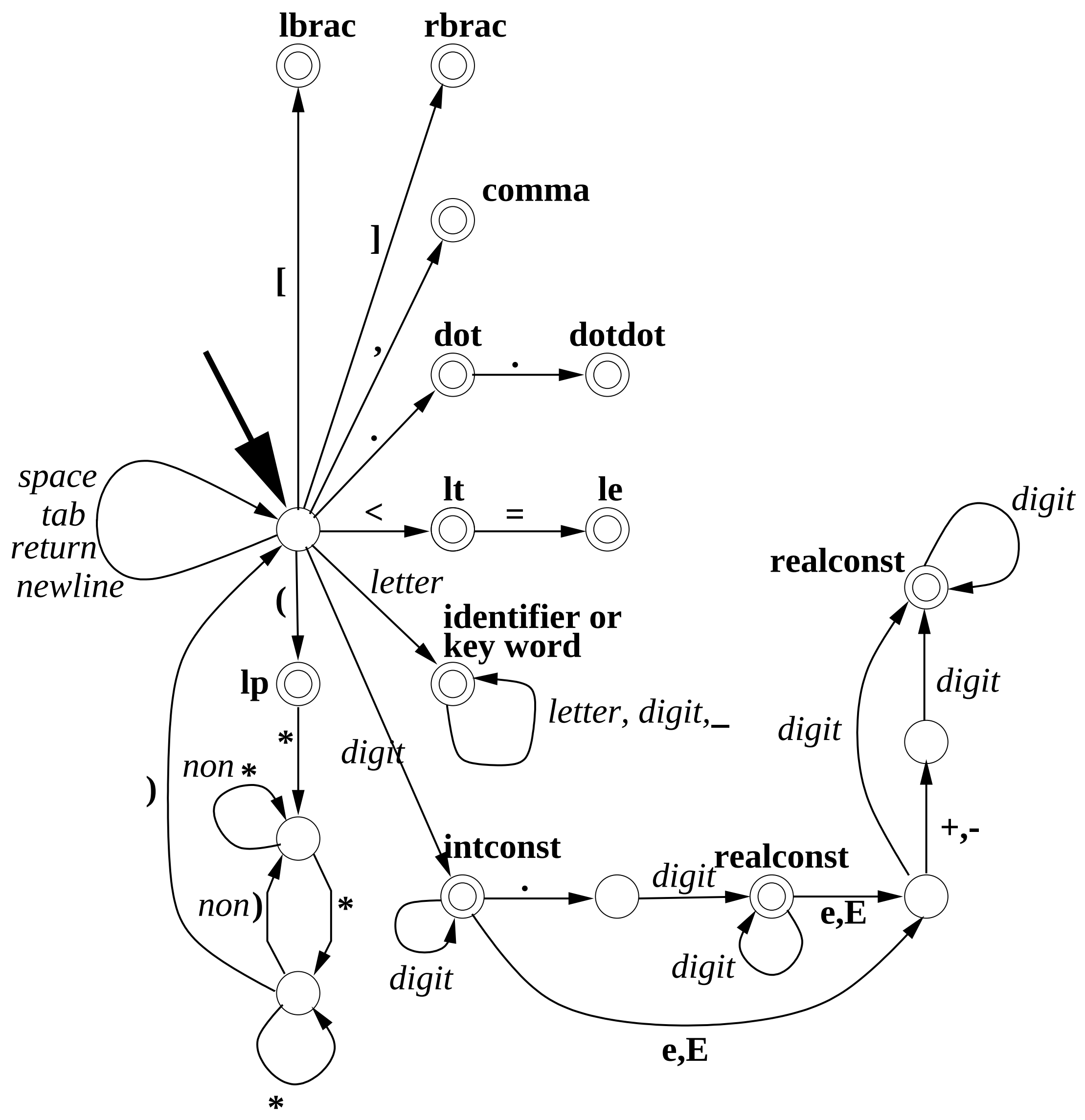
accepts: identifiers and integers

scanning

regular grammar - specifies the right strings
finite automaton (the scanner) - verifies the correctness (of tokens)
regular grammars and finite automata are algorithmically equivalent

example of a tool for scanners: **lex** - takes a regular grammar and produces an equivalent finite automaton

example 3 – Part of a Pascal scanner (tokens classified by final states)



2.1.3 Scanner implementation

- when invoked, the scanner returns the longest possible token
- removes comments (nested ones require special attention!)
- ad hoc scanners – written by hand
  – (usually) using nested case statements
- built automatically – by a scanner generator
  – (usually) using tables

▶ in the ad-hoc scanner
  – handling keywords
  – the dot-dot problem: `3.14`, `3..5`, `3.foo`

- arbitrary long look-ahead
  `DO 5 I = 1,25` – a for loop
  `DO 5 I = 1.25` – an assignment
  `DO 5,I = 1,25` – alternate syntax for loop, F77

▶ in the table-driven scanner
  – white spaces – scanner repeats main loop until finding meaningful token

example – a nested case scanner

```
state := start
loop
    case state of
        start :
            erase text of current token
            case input_char of
            ' ', '\t', '\n', '\r' : no_op
            '[' : state := got_lbrac
            ']' : state := got_rbrac
            ',' : state := got_comma
            . . .
            '(' : state := saw_lparen
            '.' : state := saw_dot
            '<' : state := saw_lthan
            . . .
            'a'..'z', 'A'..'Z' :
                state := in_ident
            '0'..'9' : state := in_int
            . . .
            else error
        . . .
        saw_lparen: case input_char of
            '*' : state := in_comment
            else return lparen
        in_comment: case input_char of
            '*' : state := leaving_comment
            else no_op
        leaving_comment: case input_char of
            ')' : state := start
            else state := in_comment
        . . .
        saw_dot : case input_char of
            '.' : state := got_dotdot
            else return dot
        . . .
        saw_lthan : case input_char of
            '=' : state := got_le
            else return lt
        . . .
        in_ident : case input_char of
            'a'..'z', 'A'..'Z', '0'..'9', '_' : no_op
            else
                look up accumulated token
                    in keyword table
                if found, return keyword
                else return identifier
        . . .
        in_int : case input_char of
            '0'..'9' : no_op
            '.' :
                peek at character beyond input_char;
                    if '0'..'9', state := saw_real_dot
                    else
                        unread peeked-at character
                        return intconst
            'a'..'z', 'A'..'Z', '_' : error
            else return intconst
        . . .
        saw_real_dot : . . .
        . . .
        got_lbrac : return lbrac
        got_rbrac : return rbrac
        got_comma : return comma
        got_dotdot : return dotdot
        got_le : return le
        . . .
    append input_char to text of current token
    read new input_char
```

example – table-driven scanner

```
state = 1..number of states
action_rec = record
      action : (move, recognize, error)
      new_state : state
      token_found : token

scan_tab : array [char, state] of action_rec
keyword_tab : set of record
      k_image : string
      k_token : token
-- these two tables are created by a scanner generator tool

tok : token
image : string
cur_state : state
cur_char : char

state := start_state
image := null
repeat
      loop
            read cur_char
            case scan_tab[cur_char, cur_state].action
                  move:
                        cur_state := scan_tab[cur_char, cur_state].new_state
                  recognize:
                        tok := scan_tab[cur_char, cur_state].token_found
                        exit inner loop
                  error:
                        -- print error message and recover; probably start over
            append cur_char to image
      -- end inner loop
until tok not in [white_space, comment]
look image up in keyword_tab and replace tok with appropriate keyword if found
return (tok, image)
```

- lexical errors

- rare (most character sequences do correspond to some token
- recovery
  - throw away the current (invalid) token
  - skip forward until new possible beginning of token
  - restart scanner
  - count on the error recovery mechanism of the parser

- sometimes useful to give more information
  - most tokens are the same: `;`, **`while`**
  - a name might be useful for a more precise error from parser
    - "`foo` has not been declared"

2.2 Concrete syntax

- describes the written representation of a language; the form of its expressions, statements, and program units
  - described using context-free grammars

context-free grammar: four parts

- a set of **nonterminals** (symbols, letters)
- a set of **terminals** (symbols, letters)
- a nonterminal chosen as the **starting symbol**
- a set of **productions** of the form:

$A ::= \alpha$, A - nonterminal
α - strings of terminals and nonterminals (possibly empty)

example 1

nonterminals: S, A, B
terminals: $a, \overline{a}, b, \overline{b}$
starting symbol: S
productions:

$$\begin{aligned} S &::= AB \\ A &::= aA\overline{a} \mid a\overline{a} \\ B &::= bB\overline{b} \mid b\overline{b} \end{aligned}$$

derivation: $S \Rightarrow AB \Rightarrow aA\overline{a}B \Rightarrow aa\overline{a}\overline{a}B \Rightarrow aa\overline{a}\overline{a}b\overline{b}$

language: $\{a^n\overline{a}^n b^m\overline{b}^m \mid n, m \geq 1\}$

example 2

nonterminals: S
terminals: `(`, `)`
starting symbol: S
productions: $S ::= SS \mid \texttt{(}\ S\ \texttt{)} \mid \texttt{(}\ \texttt{)}$

derivation:
$S \Rightarrow \texttt{(S)} \Rightarrow \texttt{(SS)} \Rightarrow \texttt{((S)S)} \Rightarrow \texttt{((())S)} \Rightarrow \texttt{((())())}$

example 3

replace in example 2:

`(`	by	`begin`
`)`	by	`end`

note: none of the languages in examples 1-3 can be generated by regular grammars (finite automata) because of the finite memory

2.2.1 Specifying concrete syntax

Backus-Naur Form (BNF)

- nonterminals enclosed between $'\langle'$ and $'\rangle'$
- empty string: $\langle \textit{empty} \rangle$

example 1

$\langle \textit{digit-sequence} \rangle ::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{digit-sequence} \rangle$
$\langle \textit{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

example 2

example 1 plus the following productions:

$$\begin{aligned}\langle \textit{real-number} \rangle \ ::= \ & \langle \textit{digit-sequence} \rangle . \langle \textit{digit-sequence} \rangle \\ \mid \ & . \langle \textit{digit-sequence} \rangle\end{aligned}$$

example 3

example 1 plus the following productions:

$\langle integer\text{-}part\rangle ::= \langle digit\rangle \mid \langle integer\text{-}part\rangle\langle digit\rangle$

$\langle fraction\rangle ::= \langle digit\rangle \mid \langle digit\rangle\langle fraction\rangle$

$\langle real\text{-}number\rangle ::= \langle integer\text{-}part\rangle.\langle fraction\rangle$

example 4

$$
\begin{aligned}
\langle expr\rangle \;::=\; & \langle expr\rangle + \langle expr\rangle \mid \langle expr\rangle - \langle expr\rangle \\
\mid\; & \langle expr\rangle * \langle expr\rangle \mid \langle expr\rangle/\langle expr\rangle \\
\mid\; & (\langle expr\rangle) \mid \langle number\rangle \mid \langle name\rangle
\end{aligned}
$$

Extended Backus-Naur Form (EBNF)

- the same power of expressing
- add to readability and writability
- new metasymbols:
  - { and } - reprezent zero or more repetitions
  - [and] - reprezent an optional component
  - (and) are used for grouping

example 1

$\langle integer \rangle ::= (+ \mid -)\langle digit\text{-}sequence \rangle \mid \langle digit\text{-}sequence \rangle$

(or $\langle integer \rangle ::= [+ \mid -]\langle digit\text{-}sequence \rangle$)

$\langle digit\text{-}sequence \rangle ::= \langle digit \rangle \{\langle digit \rangle\}$

example 2

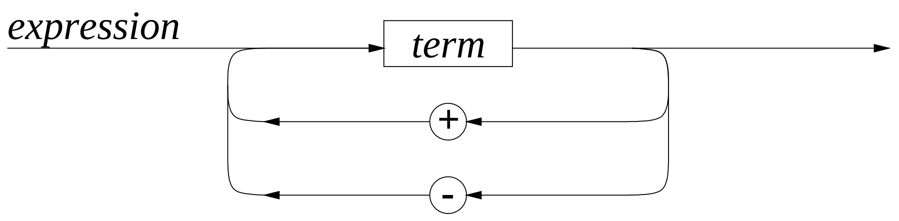
$\langle expression \rangle ::= \langle term \rangle \{(+ \mid -)\langle term \rangle\}$

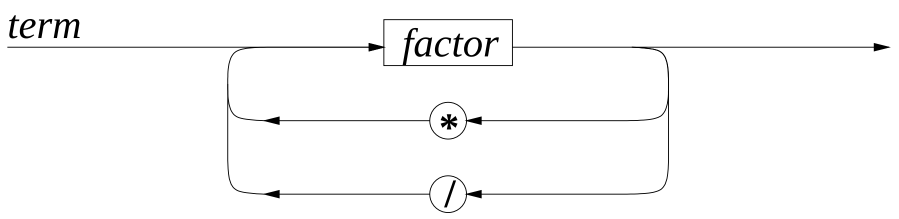
$\langle term \rangle ::= \langle factor \rangle \{(* \mid /)\langle factor \rangle\}$

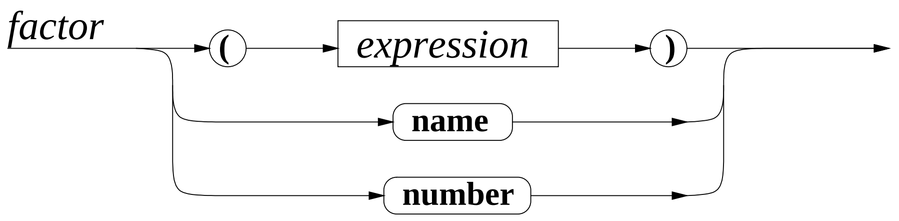
$\langle factor \rangle ::= '('\langle expression \rangle')' \mid \langle name \rangle \mid \langle number \rangle$

syntax graphs (charts)

- the same power of expressing
- visual appeal







Parse trees

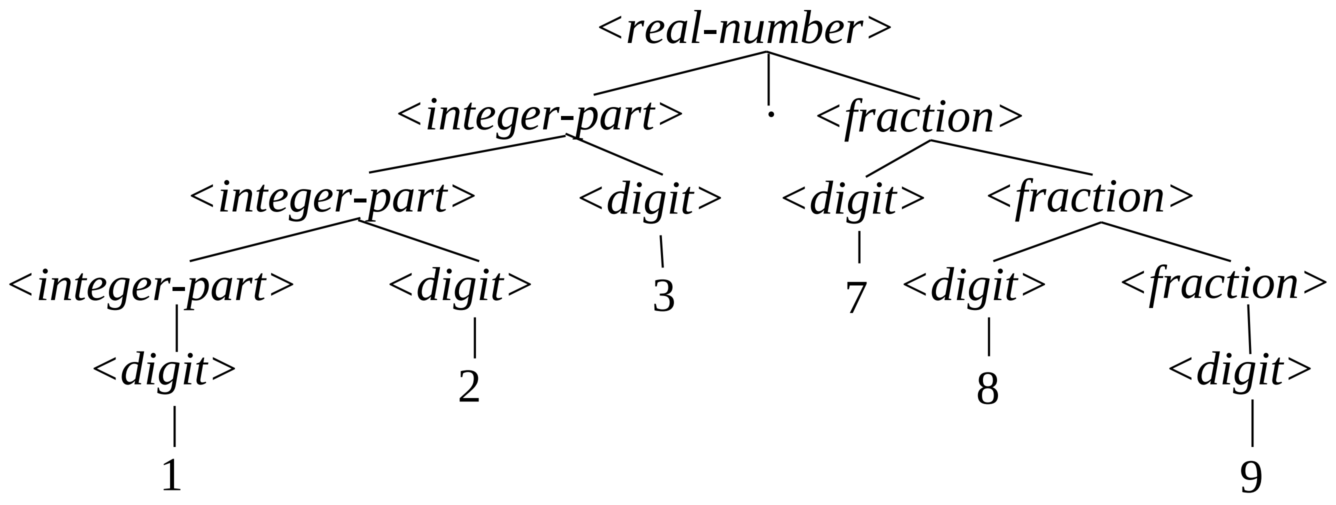
- a way to describe a derivation of a word in a context-free grammar

example 1 : consider the grammar:

$\langle digit\rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$\langle integer\text{-}part\rangle ::= \langle digit\rangle \mid \langle integer\text{-}part\rangle\langle digit\rangle$
$\langle fraction\rangle ::= \langle digit\rangle \mid \langle digit\rangle\langle fraction\rangle$
$\langle real\text{-}number\rangle ::= \langle integer\text{-}part\rangle.\langle fraction\rangle$

the word 123.789 has a derivation as below:
$\langle real\text{-}number\rangle \Rightarrow \langle integer\text{-}part\rangle.\langle fraction\rangle$
$\Rightarrow^* \langle integer\text{-}part\rangle\langle digit\rangle.\langle digit\rangle\langle fraction\rangle$
$\Rightarrow^* \langle integer\text{-}part\rangle\langle digit\rangle\langle digit\rangle.\langle digit\rangle\langle digit\rangle\langle fraction\rangle$
$\Rightarrow^* \langle digit\rangle\langle digit\rangle\langle digit\rangle.\langle digit\rangle\langle digit\rangle\langle digit\rangle \Rightarrow^* 123.789$

the corresponding parse tree is:



example 2

grammar:

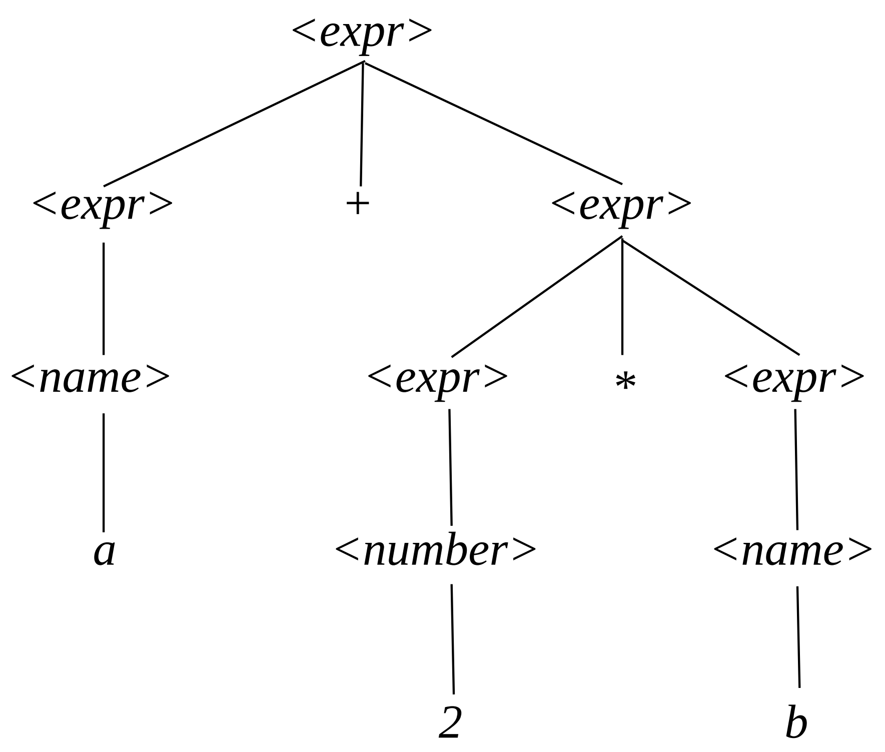
$$\begin{aligned}\langle expr\rangle \ ::= \ & \langle expr\rangle + \langle expr\rangle \mid \langle expr\rangle - \langle expr\rangle \\ \mid \ & \langle expr\rangle * \langle expr\rangle \mid \langle expr\rangle / \langle expr\rangle \\ \mid \ & (\langle expr\rangle) \mid \langle number\rangle \mid \langle name\rangle\end{aligned}$$

word: $a + 2 * b$

a derivation:

$$\langle expr\rangle \Rightarrow \langle expr\rangle + \langle expr\rangle \Rightarrow \langle expr\rangle + \langle expr\rangle * \langle expr\rangle$$
$$\Rightarrow^* \langle name\rangle + \langle number\rangle * \langle name\rangle \Rightarrow^* a + 2 * b$$

the corresponding parse tree:



2.2.2 Recognizing concrete syntax

- this is **syntax analysis** or **parsing**
  - the **parser** takes the stream of tokens from lexical analysis
  - builds a parse tree according to the rules of the underlying grammar

Scanning versus Parsing

scanning - verifies the correctness of each token (locally)
  example $i\#f$, $30a$ - wrong

parsing - verifies the correctness of the program as a whole (sequence of tokens; globally)
  example

$a := b\ c\ d$
if if else then

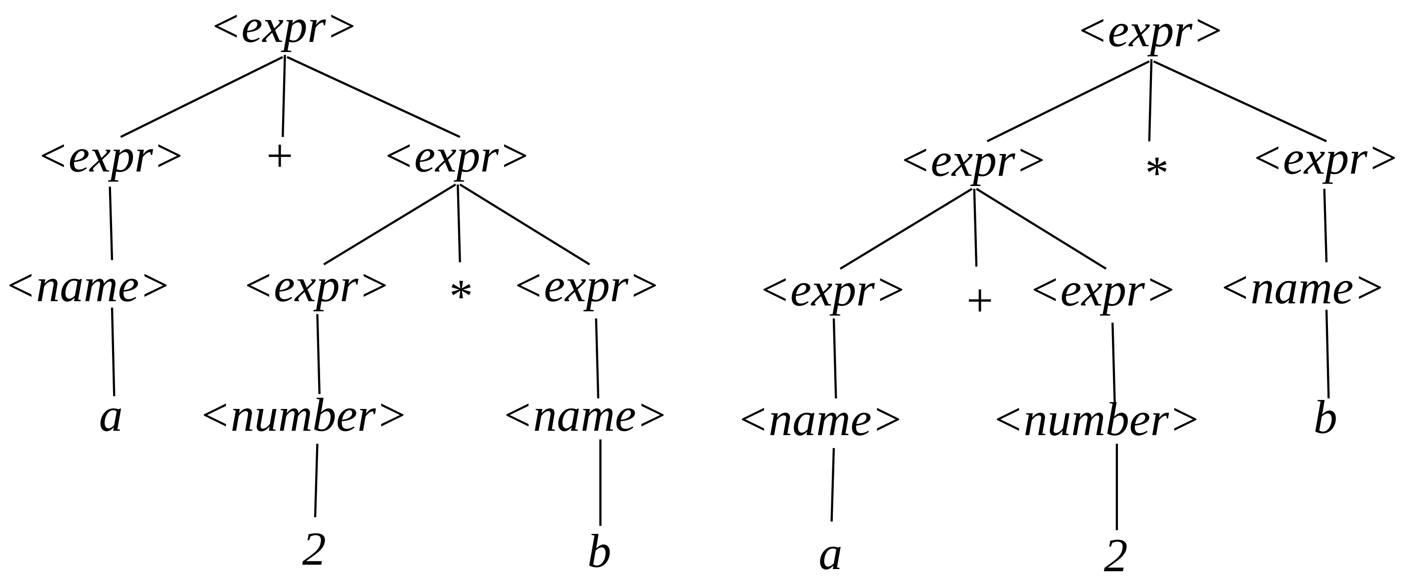
are wrong but correct from the scanner

2.2.3 Ambiguity

(syntactically) ambiguous grammar = a grammar for which a generated word has (at least) two parse trees (i.e., it can be generated in at least two different ways)

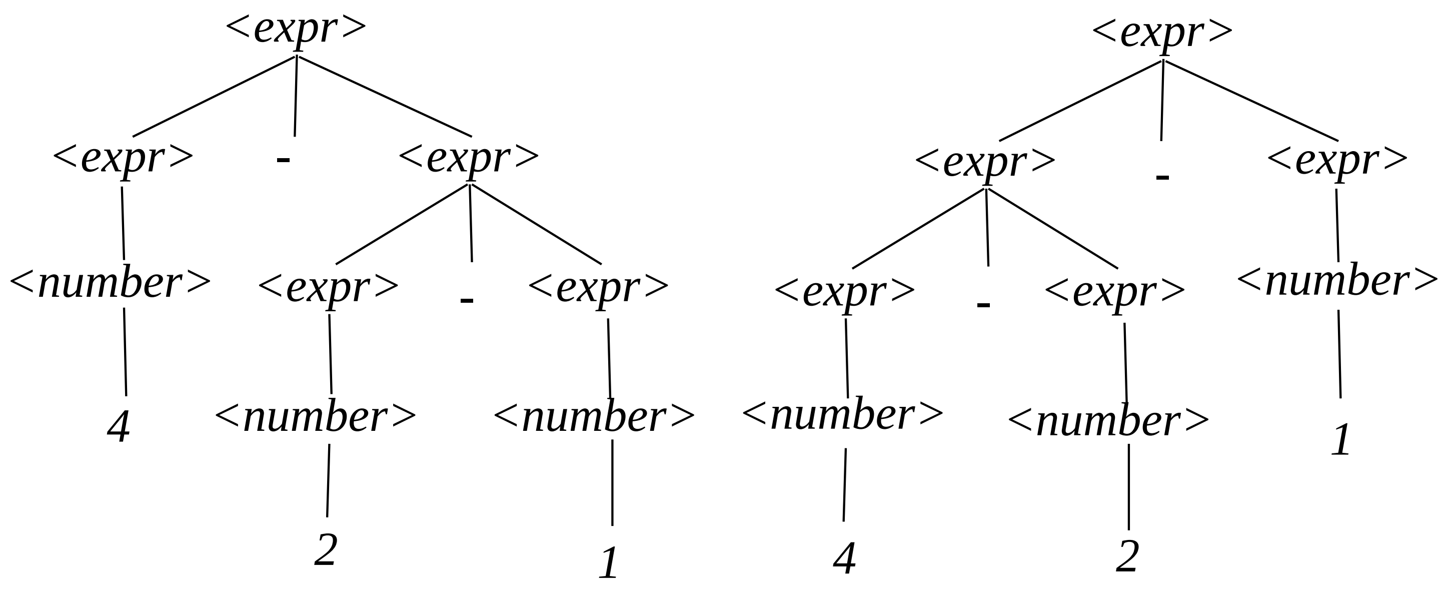
example 1

the word $a + 2 * b$ has two parse trees (precedence not respected)



example 2

the word $4 - 2 - 1$ has two parse trees (associativity not respected)



Handling associativity

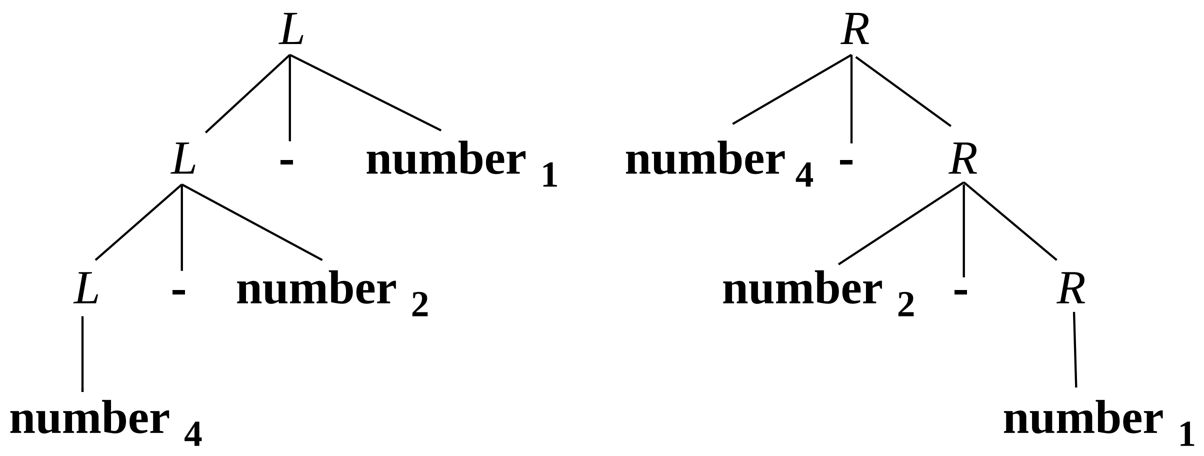
$'-'$ is left associative
$'+'$ is both left and right associative
- a grammar suitable for left associative operators is:

$$L ::= L + \textbf{number} \mid L - \textbf{number} \mid \textbf{number}$$

- a grammar suitable for right associative operators is:

$$R ::= \textbf{number} + R \mid \textbf{number} - R \mid \textbf{number}$$

- the word $4 - 2 - 1$ can be generated by any of the two grammars
  - the parse trees are:



good 4-2-1=1 bad 4-2-1=3

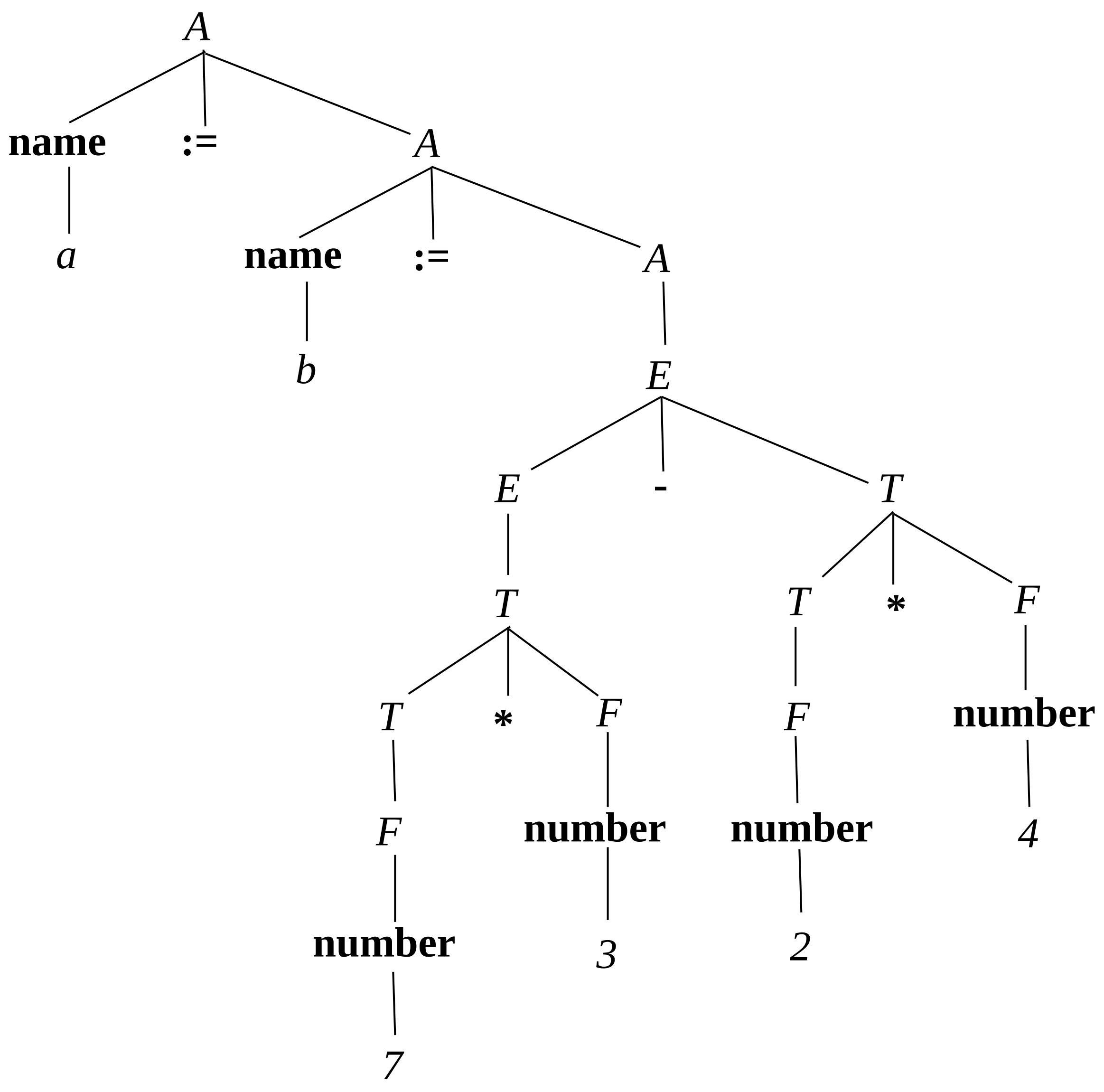
Handling associativity and precedence

$'*'$ $'/'$	left associative	$\uparrow$	
$'+'$ $'-'$	left associative	$\uparrow$	(increasing precedence)
$':='$	right associative	$\uparrow$	

- a grammar suitable for these:

$$\begin{array}{llccc}
A & ::= & \textbf{name} := A & \mid & E \\
E & ::= & E + T & \mid & E - T & \mid & T \\
T & ::= & T * F & \mid & T/F & \mid & F \\
F & ::= & (E) & \mid & \textbf{name} & \mid & \textbf{number}
\end{array}$$

idea - a new nonterminal is needed for any precedence level parse tree for the word: $a := b := 7 * 3 - 2 * 4$



T

Dangling-else ambiguity

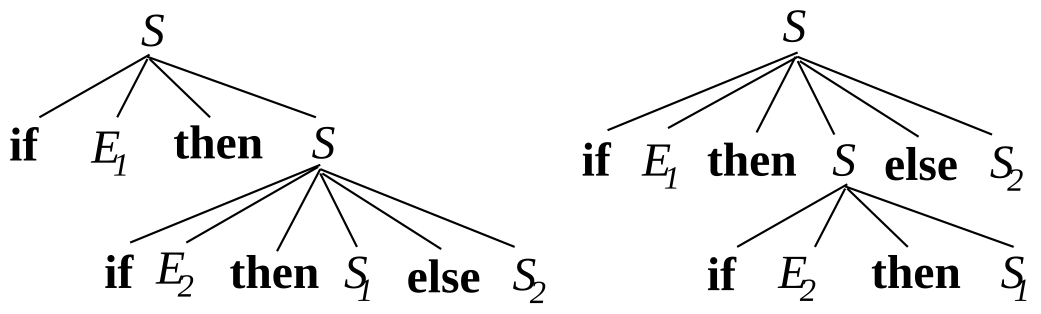
(ambiguous) grammar:

S := **if** E **then** S
S := **if** E **then** S **else** S

word with two parse trees:

if E_1 **then** **if** E_2 **then** S_1 **else** S_2

the two parse trees:

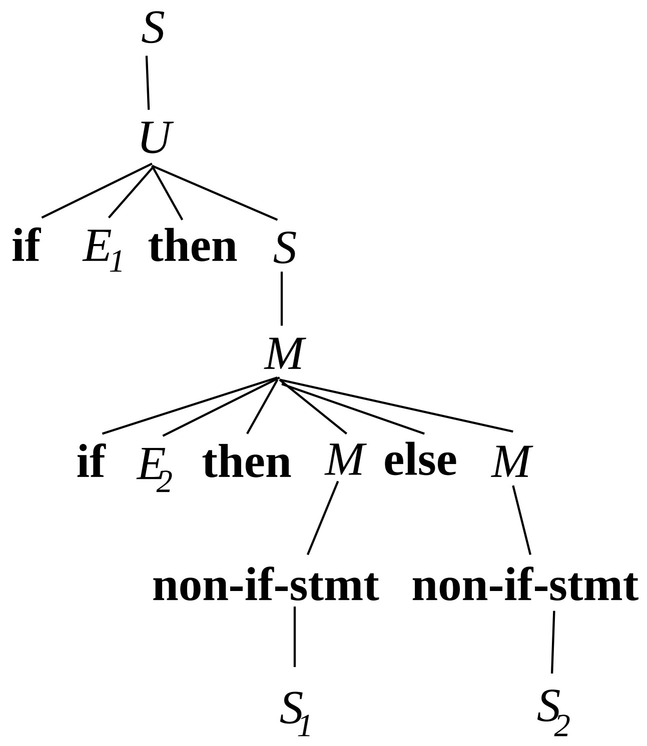


unambiguous grammar

- the idea – any **else** matched with the nearest previous unmatched **then**

$$
\begin{aligned}
S &:= M \mid U \\
M &:= \textbf{if } E \textbf{ then } M \textbf{ else } M \\
&\quad \mid \textbf{non-if-stmt} \\
U &:= \textbf{if } E \textbf{ then } S \\
&\quad \mid \textbf{if } E \textbf{ then } M \textbf{ else } U
\end{aligned}
$$

parse tree for: **if** E_1 **then** **if** E_2 **then** S_1 **else** S_2



2.2.4 Top-down and bottom-up parsing

- any context-free grammar - parser running in $\mathcal{O}(n^3)$ – too slow
- LL-grammars and LR-grammars - linear-time parsers

top-down (predictive) parser:

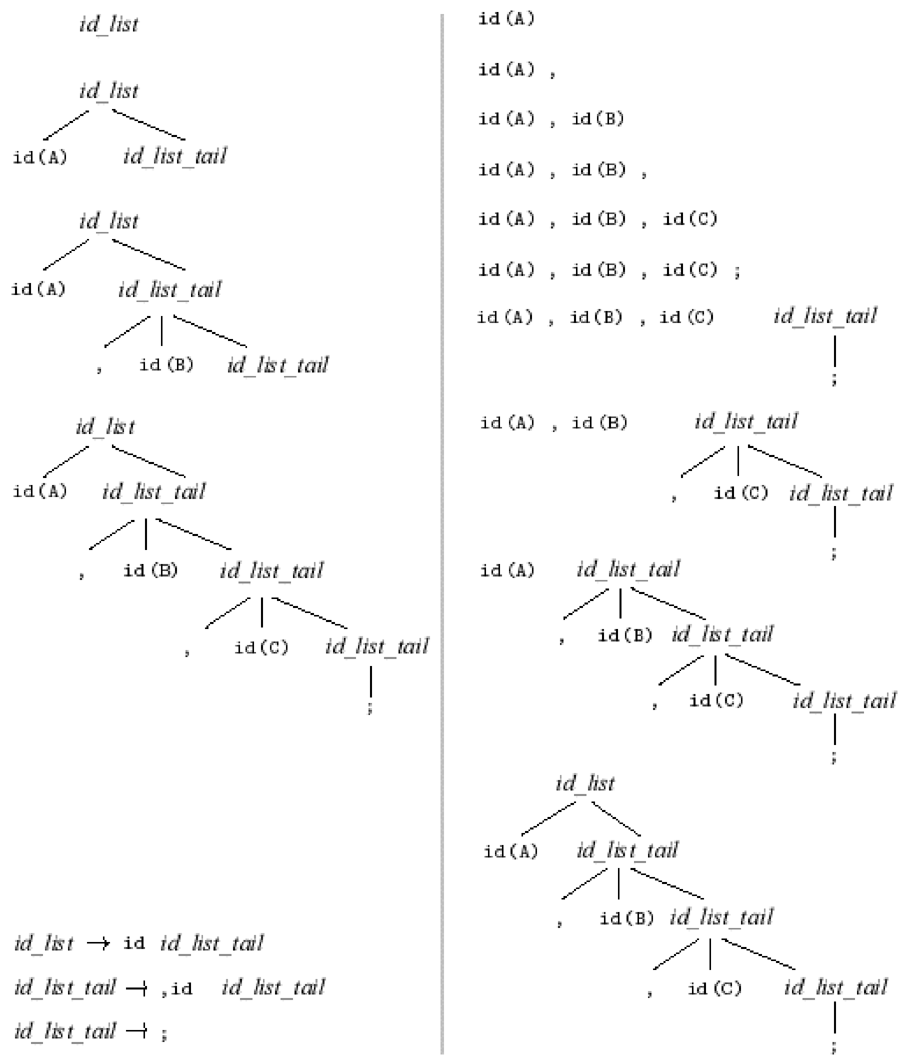
- constructs a parse tree from the root down
- predicts at each step which production to used based on next token
- LL-grammars - Left-to-right Leftmost derivation
- discovers a leftmost derivation
- LL(1) – one look-ahead token

bottom-up (shift-reduce) parser:

- constructs a parse tree from the leaves up
- maintains a forest of subtrees of the parse tree
- joins subtrees (from right) when recognizing a right-hand side of a production
- LR-grammars - Left-to-right Rightmost derivation
- discovers a rightmost derivation
- LR(1) – one look-ahead token
- larger class of parsers and grammar structure more intuitive

example – top-down versus bottom-up parsing (word `A,B,C;`)

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

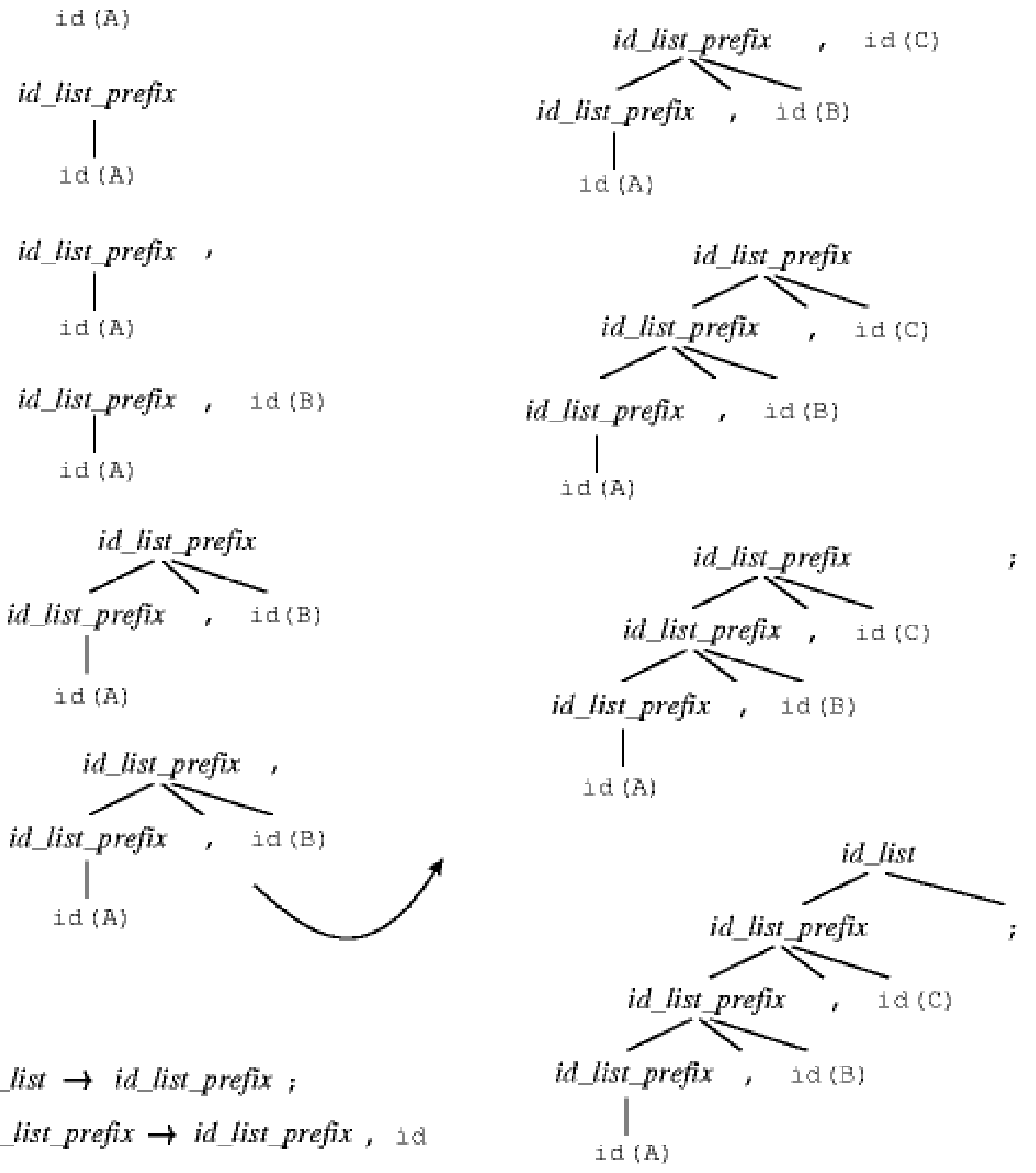


- problem with the previous grammar for bottom-up parsing
  – all tokens are shifted before any reduction is done
  – for a large program, it might run out of memory

example

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

- this grammar cannot be parsed top-down
  – when we see `id` we don't know which production to use
- it works better for bottom-up
  – it avoids too much shifting



id_list → *id_list_prefix* ;

id_list_prefix → *id_list_prefix* , id

→ id

2.2.5 Recursive descent parser

- written by hand for simple languages
- has a subroutine for every nonterminal

example

$program \longrightarrow stmt\_list$ `$$`

$stmt\_list \longrightarrow stmt\ stmt\_list \mid \epsilon$

$stmt \longrightarrow$ `id := ` $expr \mid$ `read id` $\mid$ `write` $expr$

$expr \longrightarrow term\ term\_tail$

$term\_tail \longrightarrow add\_op\ term\ term\_tail \mid \epsilon$

$term \longrightarrow factor\ fact\_tail$

$fact\_tail \longrightarrow mult\_op\ factor\ fact\_tail \mid \epsilon$

$factor \longrightarrow$ `(` $expr$ `)` $\mid$ `id` $\mid$ `literal`

$add\_op \longrightarrow$ `+` $\mid$ `-`

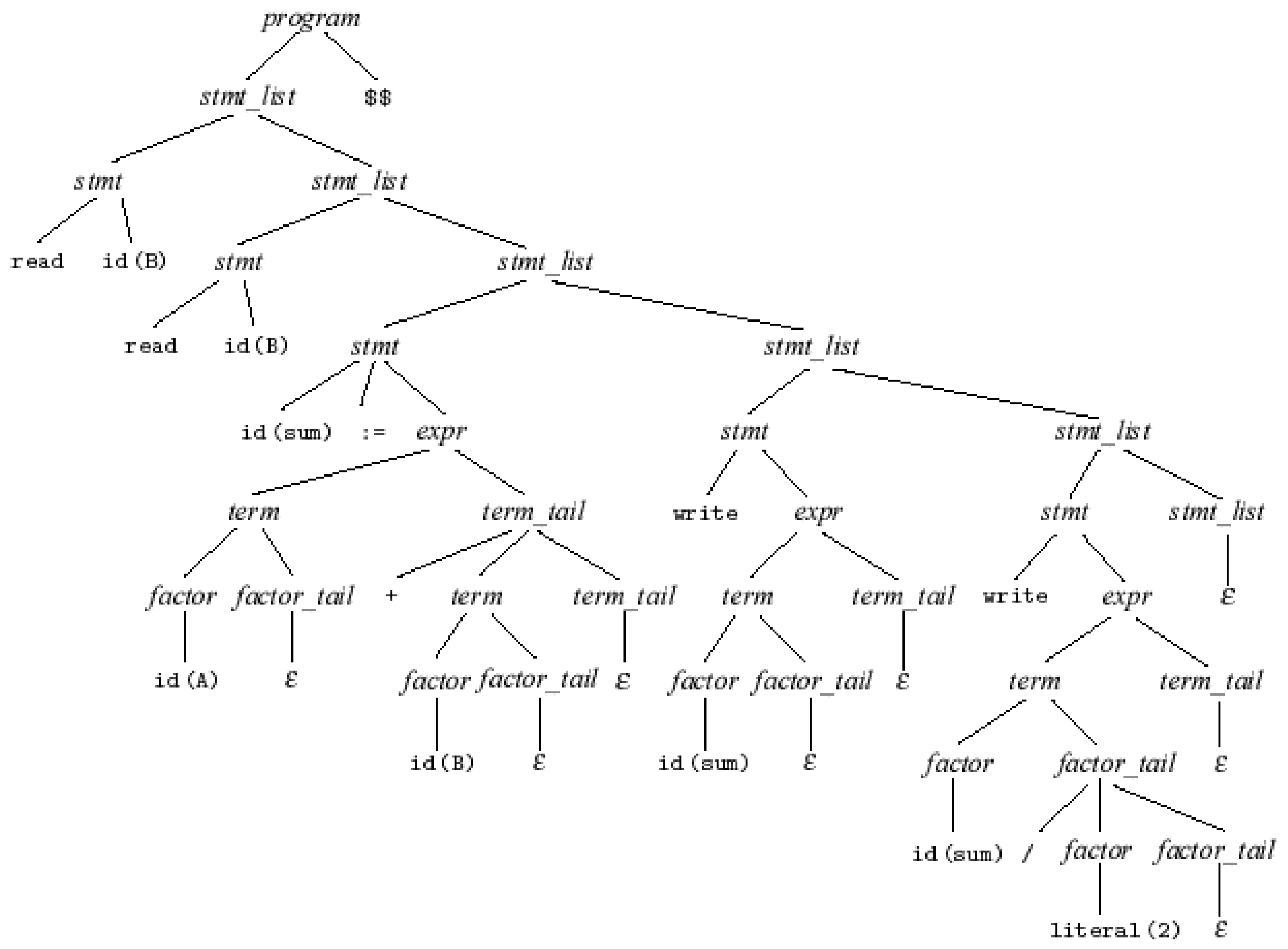
$mult\_op \longrightarrow$ `*` $\mid$ `/`

- the parser

```
procedure match (expected)
    if input_token = expected
        consume input_token
    else error

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

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip          –– epsilon production
        otherwise error

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

procedure expr
    case input_token of
        id, literal, ( : term; term_tail
        otherwise error

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

procedure term
    case input_token of
        id, literal, ( : factor; factor_tail
        otherwise error

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

procedure factor
    case input_token of
        id : match (id)
        literal : match (literal)
        ( : match ((); expr; match ())
        otherwise error

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

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

- a parse tree for

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



▶ recursive descent parsing
- start with `program`
- see `read` – call `stmt_list` then attempt to match `$$`
- call `stmt` and `stmt_list`
- `stmt_list` appears in the right-hand side of a `stmt_list` production – this gives the name of the procedure
- continue like this to find a left-to-right depth traversal of the parse tree

▶ error handling
- build the parse tree (or syntax tree – does not include the nonterminals)
- `match` should save information about tokens in the leaves

▶ `case` arm labelling
- each arm represents one production
- the labels (tokens) – predict the production
- `X` predicts a production if
  - (i) the right-hand side produces something starting with `X`
  - (ii) the right-hand side dissapears and `X` may begin the yield of what comes next

2.2.6 Syntax errors

- naive approach for errors – print error message and halt
- we want to find as many errors as possible
- modify the parser to accept further tokens
- disable back end of compiler

▸ **panic mode**

- define safe symbols
- when error, delete tokens until a safe symbol
- back parser out to a context in which that symbol might appear
- too drastic (might delete too much; if deleting important starters, might give spurious cascading errors)

example

```
IF a b THEN x;
ELSE y;
END;
```

- when discovering the error at `b`, the parser might skip forward to semicolon and miss `THEN`
- will give errors at `ELSE` and `END`

▸ phrase-level recovery

- different sets of "safe symbols" in different contexts
  - e.g., end of current expression, statement, or declaration.
- Wirth, based on FIRST and FOLLOW

```
procedure foo
  if (input_token $\notin$ FIRST$(foo)$ and $\epsilon \notin$ FIRST$(foo)$)
    announce_error --print message for the user
    repeat
      delete_token
    until input_token $\in$ (FIRST$(foo) \cup$ FOLLOW$(foo) \cup \{\$\$\}$)
  case input_token of
    ...
    ... --valid starting tokens
    ...
    otherwise return --error or $foo \to \epsilon$
```

- problem – immediate error detection
  - when $foo \to \epsilon$, the algorithm tends to predict one or more epsilon productions, when it should announce the error right away
  - this is because FOLLOW is context independent

▸ context-sensitive look-ahead

- use context dependent FOLLOW sets
- explicitly passed as a parameter to each nonterminal subroutine
- Wirth

```
procedure check_for_error(first_set, follow_set
  if input_token $\notin$ first_set
    announce_error
    repeat
      delete_token
    until input_token $\in$ first_set $\cup$ follow_set $\cup$ starter_set $\cup$ $\{\$\$\}$

procedure expr(follow_set)
  check_for_errors(FIRST($expr$), follow_set)
  case input_token of
    ...
    ··· --valid starting tokens
    ...
    otherwise return
```

example

- consider the expression

```
Y := (A * X X*X) + (B * X*X) + (C * X)
```

- obvious error (for humans): forgotten `*`

- context independent follow sets gives many errors
- it transforms the input into

```
Y := (A * X)
X := X
B := X*X
C := X
```

- context dependent follow sets gives only one error
- it translates the input into (the much closer)

```
Y := (A * X) + (B * X*X) + (C * X)
```

▸ exception-based recovery

- exception-handling mechanism – in many modern languages (Ada, Modula-3, C++, Java, ML)
- do not implement recovery for every nonterminal
- choose a small set of contexts to which we back out if error
  - e.g., nearest expression, statement
- attach an exception handler to the blocks where we want recovery

2.2.7 Table-driven top-down parsing

- mirrors the structure of a recursive descent parser
- it uses a stack containing the list of symbols it expects to see

```
repeat
    if terminal at top of stack then
        if it matches the token from scanner then
            remove token from input
            pop the stack
        else
            initiate error recovery
            (get back in a state from where it can continue)
    if nonterminal at top of stack then
        if possible to predict a production then
            predict_production_table(nonterminal, next_token)
            pop the left-hand-side symbol of production off of stack
            push the right-hand-side symbols in reverse order
        else
            initiate error recovery
            (get back in a state from where it can continue)
until input is empty
```

example

- grammar (calculator language)

$program \rightarrow stmt\_list$ `$$`
$stmt\_list \rightarrow stmt\ stmt\_list \mid \epsilon$
$stmt \rightarrow$ `id := ` $expr \mid$ `read id` $\mid$ `write` $expr$
$expr \rightarrow term\ term\_tail$
$term\_tail \rightarrow add\_op\ term\ term\_tail \mid \epsilon$
$term \rightarrow factor\ fact\_tail$
$fact\_tail \rightarrow mult\_op\ factor\ fact\_tail \mid \epsilon$
$factor \rightarrow$ `(` $expr$ `)` $\mid$ `id` $\mid$ `literal`
$add\ op \rightarrow$ `+` $\mid$ `-`
$mult\_op \rightarrow$ `*` $\mid$ `/`

- program (sum and average) – table-driven parse on next page

```
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 $$	write sum ...	match id
term_tail stmt_list $$	write sum write ...	predict *factor_tail* ⟶ ε
stmt_list $$	write sum write ...	predict *term_tail* ⟶ ε
stmt stmt_list $$	write sum write ...	predict *stmt_list* ⟶ *stmt stmt_list*
write *expr stmt_list* $$	write sum write ...	predict *stmt* ⟶ write *expr*
expr stmt_list $$	sum write sum / 2	match write
term term_tail stmt_list $$	sum write sum / 2	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list $$	sum write sum / 2	predict *term* ⟶ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	sum write sum / 2	predict *factor* ⟶ id
factor_tail term_tail stmt_list $$	sum write sum / 2	match id
term_tail stmt_list $$	write sum / 2	predict *factor_tail* ⟶ ε
stmt_list $$	write sum / 2	predict *term_tail* ⟶ ε
stmt stmt_list $$	write sum / 2	predict *stmt_list* ⟶ *stmt stmt_list*
write *expr stmt_list* $$	write sum / 2	predict *stmt* ⟶ write *expr*
expr stmt_list $$	sum / 2	match write
term term_tail stmt_list $$	sum / 2	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list $$	sum / 2	predict *term* ⟶ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	sum / 2	predict *factor* ⟶ id
factor_tail term_tail stmt_list $$	/ 2	match id
mult_op factor factor_tail term_tail stmt_list $$	/ 2	predict *factor_tail* ⟶ *mult_op factor factor_tail*
/ *factor factor_tail term_tail stmt_list* $$	/ 2	predict *mult_op* ⟶ /
factor factor_tail term_tail stmt_list $$	2	match /
literal *factor_tail term_tail stmt_list* $$	2	predict *factor* ⟶ literal
factor_tail term_tail stmt_list $$		match literal
term_tail stmt_list $$		predict *factor_tail* ⟶ ε
stmt_list $$		predict *term_tail* ⟶ ε
$$		predict *stmt_list* ⟶ ε

- predict sets

- the predict set of a production $A \rightarrow \alpha$ is

$$\text{FIRST}(\alpha) - \{\epsilon\} \text{ plus } \text{FOLLOW}(\alpha) \text{ if } \alpha \Rightarrow^* \epsilon$$

- formally

$$\begin{aligned}
\text{FIRST}(\alpha) = &\ \{a \mid \alpha \Rightarrow^* a\beta\} \\
&\cup\ (\text{ if } \alpha \Rightarrow^* \epsilon \text{ then } \{\epsilon\} \text{ else } \emptyset) \\
\text{FOLLOW}(A) = &\ \{a \mid S \Rightarrow^+ \alpha A a \beta\} \\
&\cup\ (\text{ if } S \Rightarrow^* \alpha A \text{ then } \{\epsilon\} \text{ else } \emptyset) \\
\text{PREDICT}(A \rightarrow \alpha) = &\ (\text{FIRST}(\alpha) - \{\epsilon\}) \\
&\cup\ (\text{ if } \alpha \Rightarrow^* \epsilon \text{ then } \text{FOLLOW}(A) \text{ else } \emptyset)
\end{aligned}$$

- algorithm to compute them on next page

```
First sets for all symbols:
    for all terminals a, FIRST(a) := {a}
    for all non-terminals $X$, FIRST($X$) := $\emptyset$
    for all productions $X \longrightarrow \epsilon$, add $\epsilon$ to FIRST($X$)
    repeat
        <outer> for all productions $X \longrightarrow Y_1\ Y_2 \ldots Y_k$,
            <inner> for $i$ in $1\,..\,k$
                add (FIRST($Y_i$) $\setminus \{\epsilon\}$) to FIRST($X$)
                if $\epsilon \notin$ FIRST($Y_i$) (yet)
                    continue outer loop
            add $\epsilon$ to FIRST($X$)
    until no further progress
```

```
First set subroutine for string $X_1\ X_2 \ldots X_n$, similar to the inner loop above:
    return_value := $\emptyset$
    for $i$ in $1\,..\,n$
        add (FIRST($X_i$) $\setminus \{\epsilon\}$) to return_value
        if $\epsilon \notin$ FIRST($X_i$) (yet)
            return
    add $\epsilon$ to return_value
```

```
Follow sets for all symbols:
    FOLLOW($$) := $\{\epsilon\}$
    FOLLOW($S$) := $\{\epsilon\}$, where $S$ is the start symbol
    for all other symbols $X$, FOLLOW($X$) := $\emptyset$
    repeat
        for all productions $A \longrightarrow \alpha\ B\ \beta$,
            add (FIRST($\beta$) $\setminus \{\epsilon\}$) to FOLLOW($B$)
        for all productions $A \longrightarrow \alpha\ B$
                or $A \longrightarrow \alpha\ B\ \beta$, where $\epsilon \in$ FIRST($\beta$),
            add FOLLOW($A$) to FOLLOW($B$)
    until no further progress
```

```
Predict sets for all productions:
    for all productions $A \longrightarrow \alpha$
        PREDICT(A $\longrightarrow \alpha$) := (FIRST($\alpha$) $\setminus \{\epsilon\}$)
            $\cup$ (if $\alpha \Longrightarrow^* \epsilon$ then FOLLOW($A$) else $\emptyset$)
```

- FIRST, FOLLOW, and PREDICT sets for the calculator grammar on page 77

FIRST

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

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

FOLLOW

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

PREDICT

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

- parse table

top-of-stack	current input token											
non-terminal	`id`	`literal`	`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	–

- predict-predict conflict – some token belongs to the PREDICT set of more than one production with the same left-hand side

- grammar not $LL(1)$

- $LL(1)$ **grammars** – two common obstacles

- ▶ left recursion

example – a grammar which has left recursion

$id\_list \longrightarrow id\_list\_prefix$ `;`
$id\_list\_prefix \longrightarrow id\_list\_prefix$ `, id`
$\longrightarrow$ `id`

example – after removing left recursion

$id\_list \longrightarrow$ `id` id_list_tail
$id\_list\_tail \longrightarrow$ `, id` id_list_tail
$\longrightarrow$ `;`

- common prefixes

example – a grammar with common prefixes

$$\begin{aligned} \mathit{stmt} &\longrightarrow \texttt{id} \ \texttt{:=} \ \mathit{expr} \\ &\longrightarrow \texttt{id} \ \texttt{(} \ \mathit{argument\_list} \ \texttt{)} \end{aligned}$$

example – after removing common prefixes (by left factoring)

$$\begin{aligned} &\mathit{stmt} \longrightarrow \texttt{id} \ \mathit{stmt\_list\_tail} \\ &\mathit{stmt\_list\_tail} \longrightarrow \texttt{:=} \ \mathit{expr} \mid \texttt{(} \ \mathit{argument\_list} \ \texttt{)} \end{aligned}$$

- no $LL(1)$ grammar for Pascal `if-then-else`
- grammar on page 61 can be parsed bottom-up, not top-down
- solution – different syntax

2.2.8 Bottom-up parsing

- almost always table driven
- stack – keeps the roots of the trees
- when accepting a token from scanner – *shift* it into the stack
- when recognizing a right-hand side (*handle*), *reduce* – replace, in the stack, right-hand side by the symbol in the left-hand side
    - stack for top-down – expected symbols
    - stack for bottom-up – already seen symbols
- finds a (reversed) rightmost derivation

example – the bottom-up parse on page 64 corresponds to the derivation

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

- about right-hand sides

- most important issue in bottom-up parsing – how to recognize right-hand sides
- solution – we need to know in which productions we are and where
- we mark this by a $\bullet$ in the right place in each production
- a production with a $\bullet$ is called *LR item*

- characteristic finite state machine

- recognizes *viable prefixes* – prefixes of right sentential forms that do not continue past the right end of the rightmost handle
- each state contains a list of LR items
  - basis – original items obtained by shift/reduce
  - closure – additional items given by nonterminals at the right of $\bullet$

example

- basis: $\mathit{program} \longrightarrow \bullet\ \mathit{stmt\_list}$ $$

- closure: $\mathit{stmt\_list} \longrightarrow \bullet\ \mathit{stmt\_list}\ \mathit{stmt}$

  $\mathit{stmt\_list} \longrightarrow \bullet\ \mathit{stmt}$

  $\mathit{stmt} \longrightarrow \bullet$ `id := ` expr

  $\mathit{stmt} \longrightarrow \bullet$ `read id`

  $\mathit{stmt} \longrightarrow \bullet$ `write` expr

example - grammar (for the calculator language)

$program \longrightarrow stmt\_list$ `$$`

$stmt\_list \longrightarrow stmt\_list\ stmt \mid stmt$

$stmt \longrightarrow$ `id := ` $expr \mid$ `read id` $\mid$ `write` $expr$

$expr \longrightarrow term \mid expr\ add\_op\ term$

$term \longrightarrow factor \mid term\ mult\_op\ factor$

$factor \longrightarrow$ `(` $expr$ `)` $\mid$ `id` $\mid$ `literal`

$add\_op \longrightarrow$ `+` $\mid$ `-`

$mult\_op \longrightarrow$ `*` $\mid$ `/`

- better than previous one
  - left-recursive on three nonterminals (good for bottom-up)

- program

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

- shift-reduce conflict

- xxxx • xxxx – one item with • in the middle (*shift*)
- xxxxxxxxx • – another with • at the end (*reduce*)

- parsers handle conflict differently
- e.g., in state 6, $\text{FIRST}(add\_op) \cap \text{FOLLOW}(stmt) = \emptyset$

	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 `$$` shift and reduce (pop 2 states, push *program* on input)
	stmt_list ⟶ *stmt_list* . *stmt*	on *stmt* shift and reduce (pop 2 states, push *stmt_list* on input)
	stmt ⟶ . `id` `:=` *expr*	on `id` shift and goto 3
	stmt ⟶ . `read` `id`	on `read` shift and goto 1
	stmt ⟶ . `write` *expr*	on `write` shift and goto 4
3:	*stmt* ⟶ `id` . `:=` *expr*	on `:=` shift and goto 5
4:	*stmt* ⟶ `write` . *expr*	on *expr* shift and goto 6
	expr ⟶ . *term*	on *term* shift and goto 7
	expr ⟶ . *expr add_op term*	
	term ⟶ . *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ . *term mult_op factor*	
	factor ⟶ . `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ . `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ . `literal`	on `literal` shift and reduce (pop 1 state, push *factor* on input)
5:	*stmt* ⟶ `id` `:=` . *expr*	on *expr* shift and goto 9
	expr ⟶ . *term*	on *term* shift and goto 7
	expr ⟶ . *expr add_op term*	
	term ⟶ . *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ . *term mult_op factor*	
	factor ⟶ . `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ . `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ . `literal`	on `literal` shift and reduce (pop 1 state, push *factor* on input)
6:	*stmt* ⟶ `write` *expr* .	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce (pop 2 states, push *stmt* on input)
	stmt ⟶ *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* •	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce
	term ⟶ *term* • *mult_op factor*	(pop 1 state, push *expr* on input)
		on *mult_op* shift and goto 11
	mult_op ⟶ • `*`	on `*` shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ • `/`	on `/` shift and reduce (pop 1 state, push *mult_op* on input)
8:	*factor* ⟶ `(` • *expr* `)`	on *expr* shift and goto 12
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `literal`	on `literal` shift and reduce (pop 1 state, push *factor* on input)
9:	*stmt* ⟶ `id` `:=` *expr* •	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce
	expr ⟶ *expr* • *add_op term*	(pop 3 states, push *stmt* on input)
		on *add_op* shift and goto 10
	add_op ⟶ • `+`	on `+` shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • `-`	on `-` shift and reduce (pop 1 state, push *add_op* on input)
10:	*expr* ⟶ *expr add_op* • *term*	on *term* shift and goto 13
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `literal`	on `literal` 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 ⟶ • `literal`	on `literal` shift and reduce (pop 1 state, push *factor* on input)
12:	*factor* ⟶ `(` *expr* • `)`	on `)` shift and reduce (pop 3 states, push *factor* on input)
	expr ⟶ *expr* • *add_op term*	on *add_op* shift and goto 10
	add_op ⟶ • `+`	on `+` shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • `-`	on `-` shift and reduce (pop 1 state, push *add_op* on input)
13:	*expr* ⟶ *expr add_op term* •	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce
	term ⟶ *term* • *mult_op factor*	(pop 3 states, push *expr* on input)
		on *mult_op* shift and goto 11
	mult_op ⟶ • `*`	on `*` shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ • `/`	on `/` shift and reduce (pop 1 state, push *mult_op* on input)

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

3 Semantics

- semantic rules
- attribute grammars
- attribute flow
- action routines
- space management

3.1 What is semantics?

- syntax – the form
- semantics – the meaning

3.1.1 Semantic rules

▶ static
  – enforced by the compiler at compile time

▶ dynamic
  – checked at run time, using the code generated by the compiler
  – e.g., division by zero, out-of-bound index

- computability theory says: **ensuring correct semantics is impossible**
- that is, there is no algorithm to predict everything for arbitrary programs

– there will inevitably be cases in which an error will always occur, but the compiler cannot tell – will delay the error message until run time

– there will inevitably be cases in which an error will never occur, but the compiler cannot tell – will generate code for unnecessary run time checks

3.1.2 Role of semantic analyzer

- programming languages vary dramatically in the choice of semantic rules
  - C allows operands of many types intermixed in expressions
  - Ada does not

  - C – no dynamic checks (except coming free with hardware)
  - Java – as many checks as possible

- dynamic checks that can be disabled
  - enabled during testing
  - disable for production use – for speed

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”

- attributed semantics – meaning of programs using attribute grammars
- denotational semantics – machine independent
- axiomatic semantics – proving theorems about behavior of programs

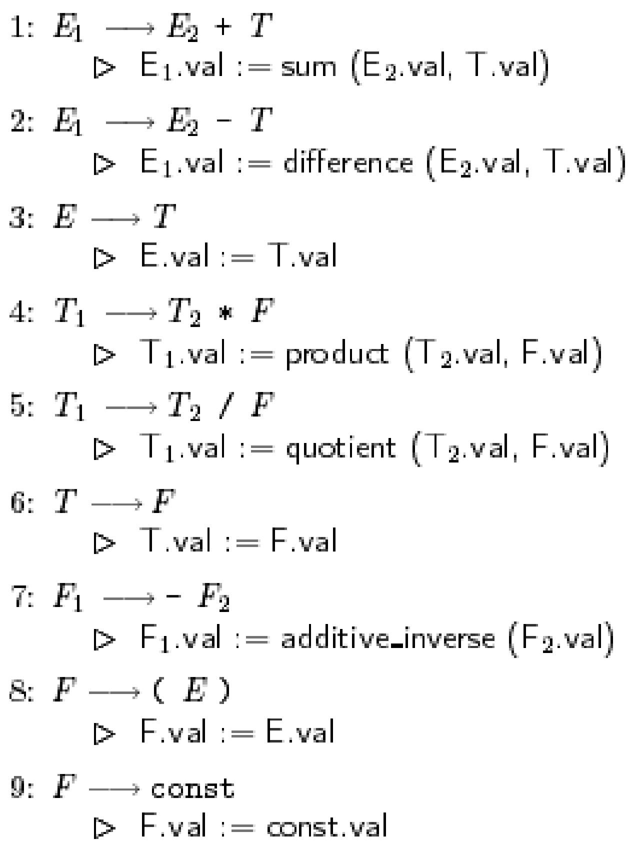
3.2 Attribute grammars

- **attributes** – associated with symbols of a grammar
  - give meaning to syntactic constructs

<u>example</u> – left - grammar for expressions; right - with attributes and rules

$$
\begin{array}{ll}
E \rightarrow E + T & T \rightarrow F \\
E \rightarrow E + T & F \rightarrow - F \\
E \rightarrow T & F \rightarrow (E) \\
T \rightarrow T * F & F \rightarrow \mathit{const} \\
T \rightarrow T / F &
\end{array}
$$

- rules
  - copy rules
  - using semantic functions; arguments are attributes of symbols in the same production; the result must be assigned to an attribute of a symbol in the production



1: $E_1 \longrightarrow E_2 + T$
 ▷ E_1.val := sum (E_2.val, T.val)

2: $E_1 \longrightarrow E_2 - T$
 ▷ E_1.val := difference (E_2.val, T.val)

3: $E \longrightarrow T$
 ▷ E.val := T.val

4: $T_1 \longrightarrow T_2 * F$
 ▷ T_1.val := product (T_2.val, F.val)

5: $T_1 \longrightarrow T_2 / F$
 ▷ T_1.val := quotient (T_2.val, F.val)

6: $T \longrightarrow F$
 ▷ T.val := F.val

7: $F_1 \longrightarrow - F_2$
 ▷ F_1.val := additive_inverse (F_2.val)

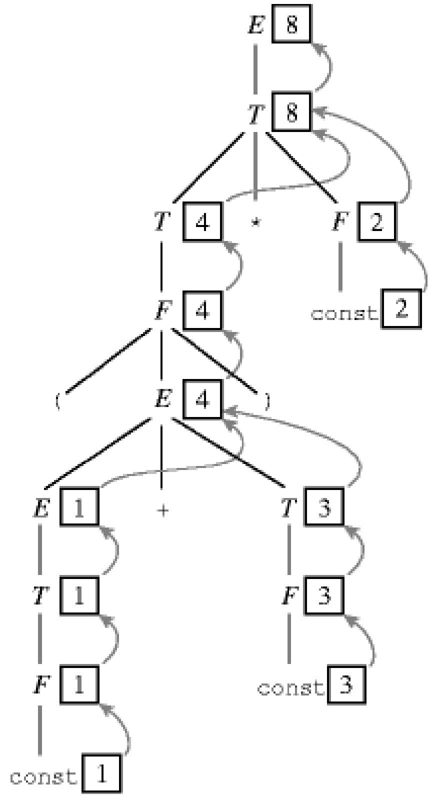
8: $F \longrightarrow (E)$
 ▷ F.val := E.val

9: $F \longrightarrow$ const
 ▷ F.val := const.val

3.3 Attribute flow

- **attribute flow** – moving information between attributes
- **annotation** of parse tree – evaluation of attributes

example – annotation of a parse tree for `(1+3)*2`



- **synthesized** attributes – whose value is computed only in productions in which their symbol appears on the left-hand side (information going bottom-up)
- **S-attributed** grammar – if all attributes are synthesized
  - the most general class of attribute grammars that can be evaluated on-the-fly during an LR-parse
- **inherited** attributes – whose value is computed in productions in which their symbol appears on the right-hand side (information can come from above or from the side)
- **L-attributed** grammar – if all attributes can be evaluated by visiting the nodes of the parse tree in a single, left-to-right depth-first traversal
  - the most general class of attribute grammars that can be evaluated on-the-fly during an LL-parse (it is a proper superset of S-attributed grammars)
  - formally, a grammar is L-attributed if:

    (i) each synthesized attribute of a left-hand side symbol depends only on its own inherited attributes or on attributes (synthesized of inherited) of the symbols in the right-hand side

    (ii) each inherited attribute of a right-hand side symbol depends only on inherited attributes of the left-hand side symbol or on attributes (synthesized of inherited) of the symbols to its left in the right-hand side

example – attribute grammar for expressions, based on LL(1) cfg

1: $E \longrightarrow T\ TT$
$\rhd$ TT.st := T.val $\rhd$ E.val := TT.val

2: $TT_1 \longrightarrow +\ T\ TT_2$
$\rhd$ TT_2.st := TT_1.st + T.val $\rhd$ TT_1.val := TT_2.val

3: $TT_1 \longrightarrow -\ T\ TT_1$
$\rhd$ TT_2.st := TT_1.st − T.val $\rhd$ TT_1.val := TT_2.val

4: $TT \longrightarrow \epsilon$
$\rhd$ TT.val := TT.st

5: $T \longrightarrow F\ FT$
$\rhd$ FT.st := F.val $\rhd$ T.val := FT.val

6: $FT_1 \longrightarrow *\ F\ FT_2$
$\rhd$ FT_2.st := FT_1.st × F.val $\rhd$ FT_1.val := FT_2.val

7: $FT_1 \longrightarrow /\ F\ FT_2$
$\rhd$ FT_2.st := FT_1.st ÷ F.val $\rhd$ FT_1.val := FT_2.val

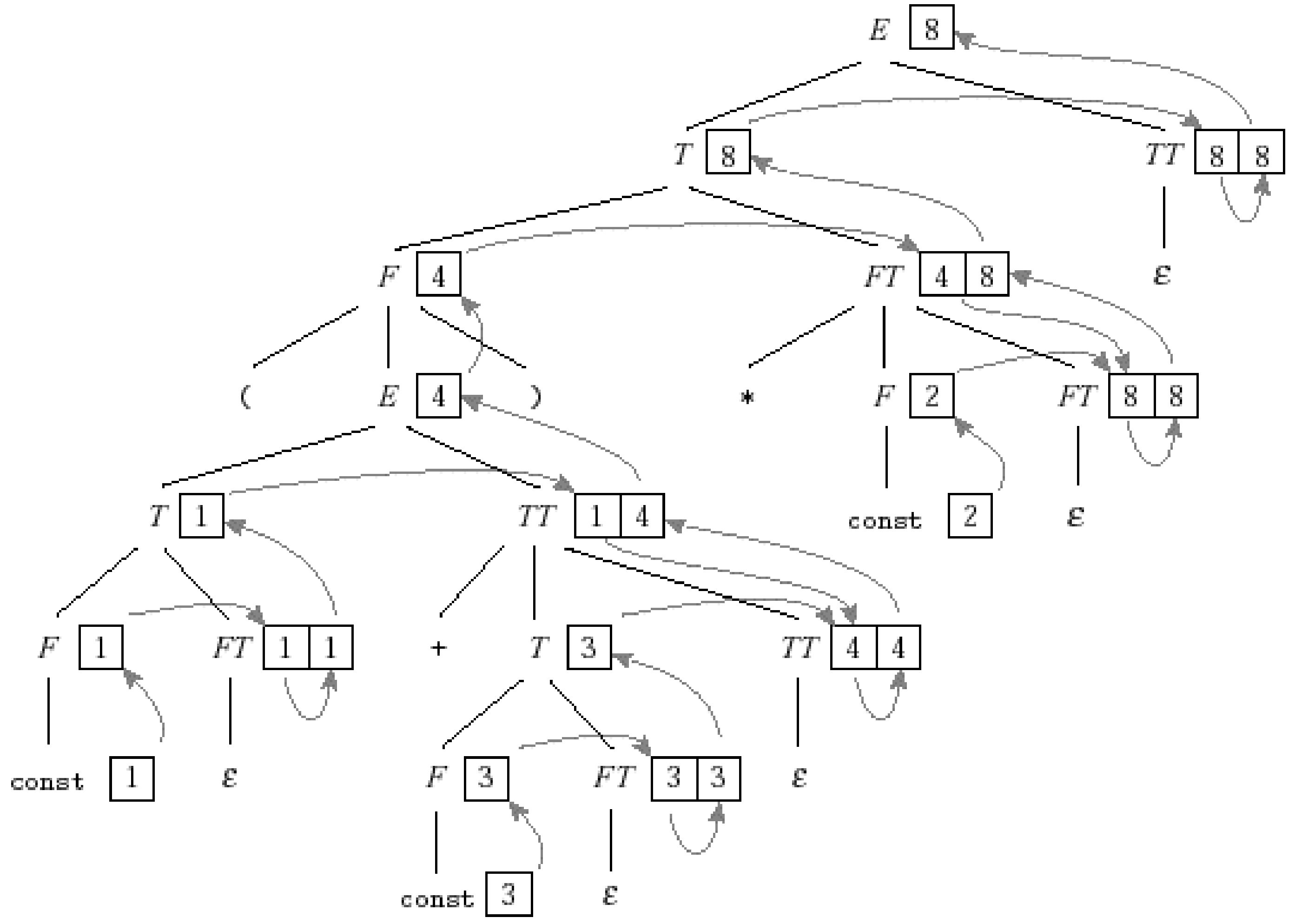
8: $FT \longrightarrow \epsilon$
$\rhd$ FT.val := FT.st

9: $F_1 \longrightarrow -\ F_2$
$\rhd$ F_1.val := − F_2.val

10: $F \longrightarrow (\ E\ )$
$\rhd$ F.val := E.val

11: $F \longrightarrow$ const
$\rhd$ F.val := const.val

example – annotation of a top-down parse tree for `(1+3)*2` using the attribute grammar on the previous page



3.3.1 Syntax tree construction

- one-pass compiler – interleaves semantic analysis and code generation with parsing
  - the syntax tree might not be needed
- when semantic analysis is not interleaved with parsing, we can use attributes to construct the syntax tree

example – bottom-up attribute grammar to construct a syntax tree

$E_1 \longrightarrow E_2$ + T

▷ $\mathsf{E_1.ptr := make\_bin\_op\ ("+", E_2.ptr, T.ptr)}$

$E_1 \longrightarrow E_2$ - T

▷ $\mathsf{E_1.ptr := make\_bin\_op\ ("-", E_2.ptr, T.ptr)}$

$E \longrightarrow T$

▷ $\mathsf{E.ptr := T.ptr}$

$T_1 \longrightarrow T_2$ * F

▷ $\mathsf{T_1.ptr := make\_bin\_op\ ("\times", T_2.ptr, F.ptr)}$

$T_1 \longrightarrow T_2$ / F

▷ $\mathsf{T_1.ptr := make\_bin\_op\ ("\div", T_2.ptr, F.ptr)}$

$T \longrightarrow F$

▷ $\mathsf{T.ptr := F.ptr}$

$F_1 \longrightarrow$ - F_2

▷ $\mathsf{F_1.ptr := make\_un\_op\ ("{}^{+}/_{-}", F_2.ptr)}$

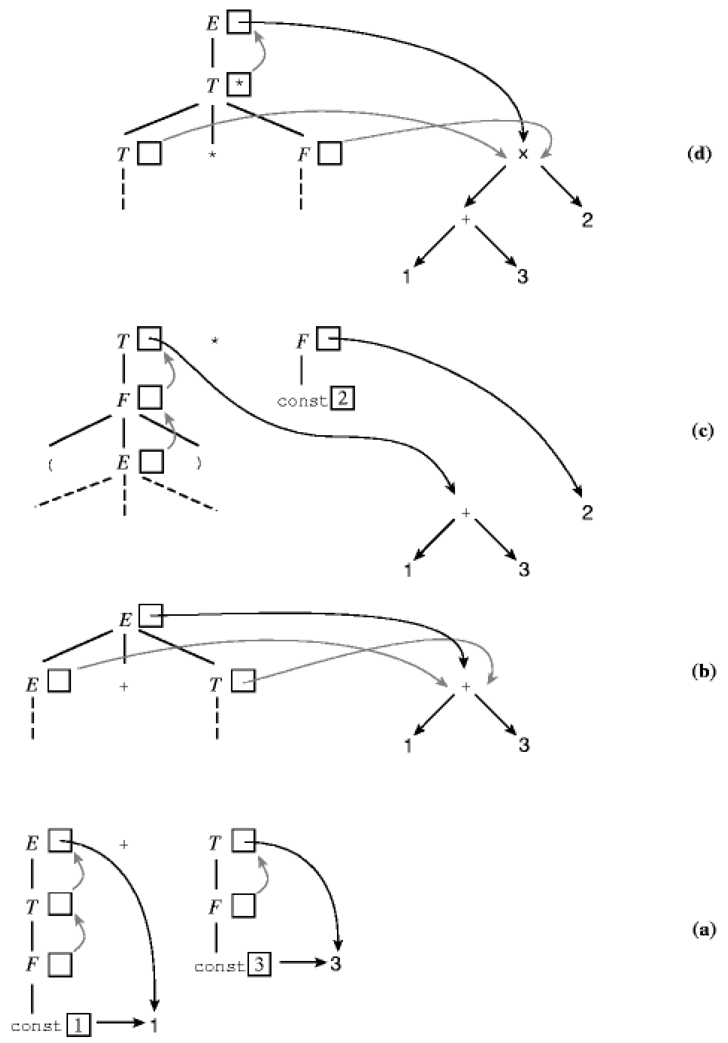
$F \longrightarrow$ (E)

▷ $\mathsf{F.ptr := E.ptr}$

$F \longrightarrow$ const

▷ $\mathsf{F.ptr := make\_leaf\ (const.val)}$

example – construction of a syntax tree via annotation of a bottom-up parse tree, using the grammar on previous page



example – top-down attribute grammar to construct a syntax tree

$E \longrightarrow T\ TT$
  ▷ TT.st := T.ptr   ▷ E.ptr := TT.ptr

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

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

$TT \longrightarrow \epsilon$
  ▷ TT.ptr := TT.st

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

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

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

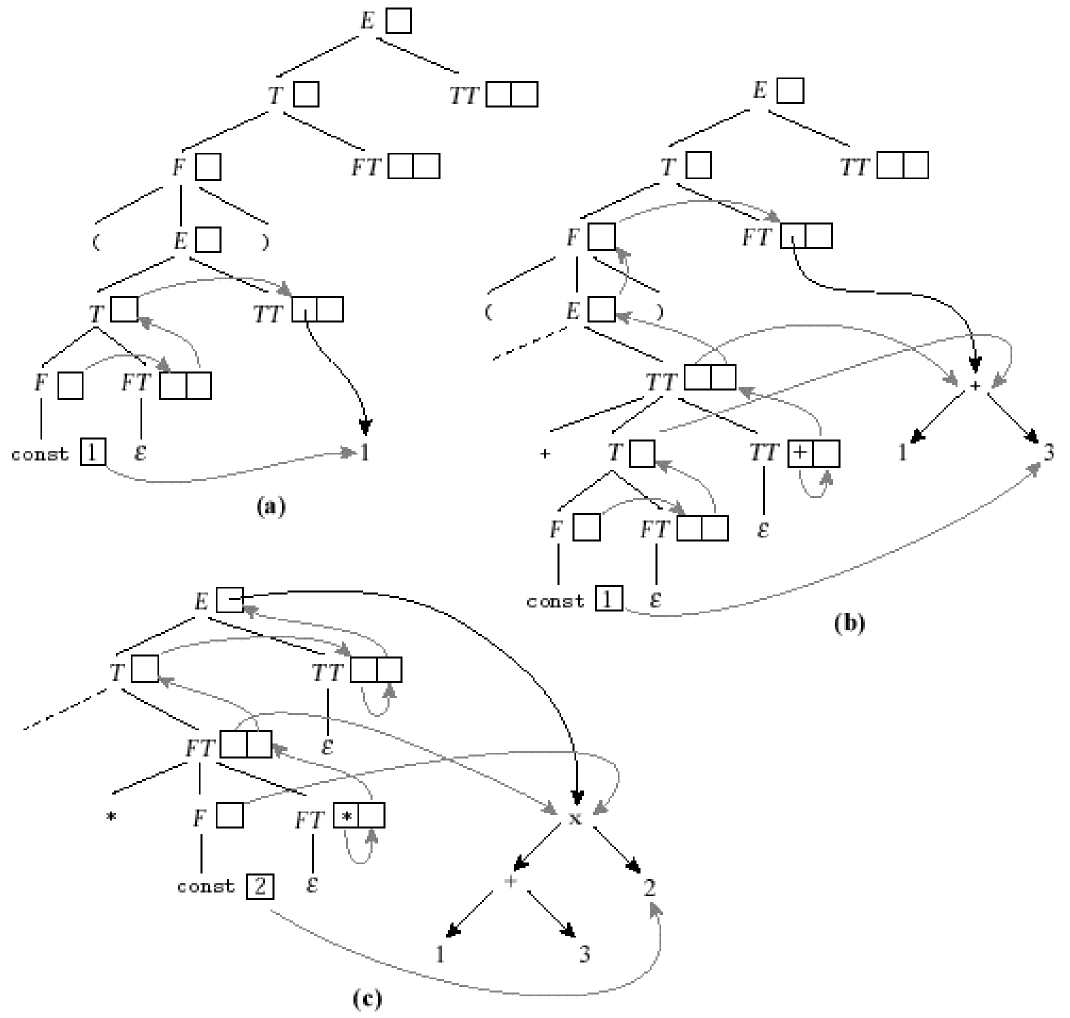
$FT \longrightarrow \epsilon$
  ▷ FT.ptr := FT.st

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

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

$F \longrightarrow \mathtt{const}$
  ▷ F.ptr := make_leaf (const.val)

example – construction of a syntax tree via annotation of a top-down parse tree, using the grammar on previous page



3.4 Action routines

- translation scheme – an algorithm that invokes the attribute rules of a parse tree in an order that respects its attribute flow

- action routines – ad hoc translation scheme that is interleaved with parsing

- there can be also automatic tools to construct a semantic analyzer for a given attribute grammar

- most compilers use ad-hoc hand-written translation schemes

- an *action routine* is a semantic function that the programmer instructs the compiler to execute at a particular point in the parse

▶ for LL parsers

- action routines can appear everywhere within a right-hand side
- it will be called as soon as the symbols before it were matched
- parser pushes, among terminals and nonterminals, pointers to action routines

example – LL(1) cfg with action routines to build a syntax tree

$$
\begin{array}{lll}
E & \longrightarrow & T \; \{\, \mathsf{TT.st := T.ptr} \,\} \; TT \; \{\, \mathsf{E.ptr := TT.ptr} \,\} \\
TT_1 & \longrightarrow & + \; T \; \{\, \mathsf{TT_2.st := make\_bin\_op\,(\text{"}+\text{"},\, TT_1.st,\, T.ptr)} \,\} \; TT_2 \; \{\, \mathsf{TT_1.ptr := TT_2.ptr} \,\} \\
TT_1 & \longrightarrow & - \; T \; \{\, \mathsf{TT_2.st := make\_bin\_op\,(\text{"}-\text{"},\, TT_1.st,\, T.ptr)} \,\} \; TT_2 \; \{\, \mathsf{TT_1.ptr := TT_2.ptr} \,\} \\
TT & \longrightarrow & \epsilon \; \{\, \mathsf{TT.ptr := TT.st} \,\} \\
T & \longrightarrow & F \; \{\, \mathsf{FT.st := F.ptr} \,\} \; FT \; \{\, \mathsf{T.ptr := FT.ptr} \,\} \\
FT_1 & \longrightarrow & * \; F \; \{\, \mathsf{FT_2.st := make\_bin\_op\,(\text{"}\times\text{"},\, FT_1.st,\, F.ptr)} \,\} \; FT_2 \; \{\, \mathsf{FT_1.ptr := FT_2.ptr} \,\} \\
FT_1 & \longrightarrow & / \; F \; \{\, \mathsf{FT_2.st := make\_bin\_op\,(\text{"}\div\text{"},\, FT_1.st,\, F.ptr)} \,\} \; FT_2 \; \{\, \mathsf{FT_1.ptr := FT_2.ptr} \,\} \\
FT & \longrightarrow & \epsilon \; \{\, \mathsf{FT.ptr := FT.st} \,\} \\
F_1 & \longrightarrow & - \; F_2 \; \{\, \mathsf{F_1.ptr := make\_un\_op\,(\text{"}+/\_\text{"},\, F_2.ptr)} \,\} \\
F & \longrightarrow & (\; E \;) \; \{\, \mathsf{F.ptr := E.ptr} \,\} \\
F & \longrightarrow & \mathtt{const} \; \{\, \mathsf{F.ptr := make\_leaf\,(const.ptr)} \,\}
\end{array}
$$

▶ **for LR parsers**

- action routines can appear only after the point at which the production can be identified unambiguously
- if the flow is bottom-up, then execution at the end of right-end sides is enough

example – see pages 95 and 101

3.5 Space management for attributes

- we need space for the attributes
- if the parse tree is explicitly built then we can store the attributes in the nodes
- if the parse tree is not built, we need to store the attributes of the symbols which we have seen/predicted but not yet finished parsing

3.5.1 In bottom-up parsing

▶ **for S-attributed grammars**

- maintain an **attribute stack** that mirrors the parse stack
- pseudoconstructs for evaluation
  - `$$` for left-hand side
  - `$1`, `$2`, ... for right-hand side
- e.g., `$$`.val = `$1`.val + `$3`.val

example - parse/attribute stack for `(1+3)*2` using grammar on page 95

1:	(
2:	(1
3:	(F_1
4:	(T_1
5:	(E_1
6:	(E_1 +
7:	(E_1 + 3
8:	(E_1 + F_3
9:	(E_1 + T_3
10:	(E_4
11:	(E_4)
12:	F_4
13:	T_4
14:	T_4 *
15:	T_4 * 2
16:	T_4 * F_2
17:	T_8
18:	E_8

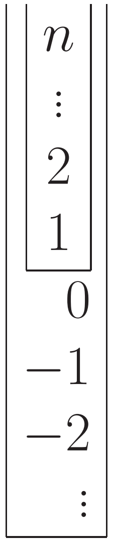
▸ for inherited attributes

- consider the grammar below
  - for variable declaration in C or Fortran

$dec \longrightarrow type\ id\_list$
$id\_list \longrightarrow$ `id`
$id\_list \longrightarrow id\_list$ `, id`

- *type* has attribute *tp* – synthesized
- we want to pass *tp* to *id_list* as inherited – to enter full information about variables in the symbol table
- computing the attributes using action routines

$dec \longrightarrow type\ id\_list$
$id\_list \longrightarrow$ `id` { declare_id(`$1`.name, `$0`.tp) }
$id\_list \longrightarrow id\_list$ `, id` { declare_id(`$3`.name, `$0`.tp) }



- idea: inherited attributes are "faked" by accessing the synthesized attributes of symbols known to lie below the current production in the stack (n = length of right-hand side)

- semantic hooks

- consider the grammar below – for statements in Perl
  - context-dependent meaning of an expression

example

$$\begin{aligned} stmt &\rightarrow \texttt{id := } expr && \text{– type of } \texttt{id} \text{ expected} \\ &\rightarrow \ldots \\ &\rightarrow \texttt{if } expr \texttt{ then } stmt && \text{– type Boolean expected} \\ expr &\rightarrow \ldots \end{aligned}$$

- in the production for *expr* the context is unknown
- idea: add semantic hooks to the grammar

example (continued)

$$\begin{aligned} stmt &\rightarrow \texttt{id := } A\ expr \\ &\rightarrow \ldots \\ &\rightarrow \texttt{if } B\ expr \texttt{ then } stmt \\ A &\rightarrow \epsilon\ \{\ \texttt{\$\$}.\text{tp} := \texttt{\$-1}.\text{tp}\ \} \\ B &\rightarrow \epsilon\ \{\ \texttt{\$\$}.\text{tp} := \text{Boolean}\ \} \\ expr &\rightarrow \ldots \{\texttt{if } \texttt{\$0}.\text{tp} = \text{Boolean } \texttt{then} \ldots\} \end{aligned}$$

3.5.2 In top-down parsing

▶ **for recursive descent parsers** (hand written)
- synthesized attributes – return parameters
- inherited attributes – passed as parameters

▶ **for table driven parsers** (automatically generated)
- automatic management
- use an attribute stack – it contains all symbols in all productions on the path from the root to the symbol at the top of the stack

example – LL(1) grammar for constant expressions with action routines

$$
\begin{array}{lll}
E & \longrightarrow & T \; \{ \text{TT.st} := \text{T.val} \}^{1} \; TT \; \{ \text{E.val} := \text{TT.val} \}^{2} \\
TT_1 & \longrightarrow & \texttt{+} \; T \; \{ \text{TT}_2\text{.st} := \text{TT}_1\text{.st} + \text{T.val} \}^{3} \; TT_2 \; \{ \text{TT}_1\text{.val} := \text{TT}_2\text{.val} \}^{4} \\
TT_1 & \longrightarrow & \texttt{-} \; T \; \{ \text{TT}_2\text{.st} := \text{TT}_1\text{.st} - \text{T.val} \}^{5} \; TT_2 \; \{ \text{TT}_1\text{.val} := \text{TT}_2\text{.val} \}^{6} \\
TT & \longrightarrow & \epsilon \; \{ \text{TT.val} := \text{TT.st} \}^{7} \\
T & \longrightarrow & F \; \{ \text{FT.st} := \text{F.val} \}^{8} \; FT \; \{ \text{T.val} := \text{FT.val} \}^{9} \\
FT_1 & \longrightarrow & \texttt{*} \; F \; \{ \text{FT}_2\text{.st} := \text{FT}_1\text{.st} \times \text{F.val} \}^{10} \; FT_2 \; \{ \text{FT}_1\text{.val} := \text{FT}_2\text{.val} \}^{11} \\
FT_1 & \longrightarrow & \texttt{/} \; F \; \{ \text{FT}_2\text{.st} := \text{FT}_1\text{.st} \div \text{F.val} \}^{12} \; FT_2 \; \{ \text{FT}_1\text{.val} := \text{FT}_2\text{.val} \}^{13} \\
FT & \longrightarrow & \epsilon \; \{ \text{FT.val} := \text{FT.st} \}^{14} \\
F_1 & \longrightarrow & \texttt{-} \; F_2 \; \{ \text{F}_1\text{.val} := - \text{F}_2\text{.val} \}^{15} \\
F & \longrightarrow & \texttt{(} \; E \; \texttt{)} \; \{ \text{F.val} := \text{E.val} \}^{16} \\
F & \longrightarrow & \texttt{const} \; \{ \text{F.val} := \text{C.val} \}^{17}
\end{array}
$$

example

- trace of the parse stack (left) and attribute stack (right) for `(1+3)*2` using the grammar on previous page

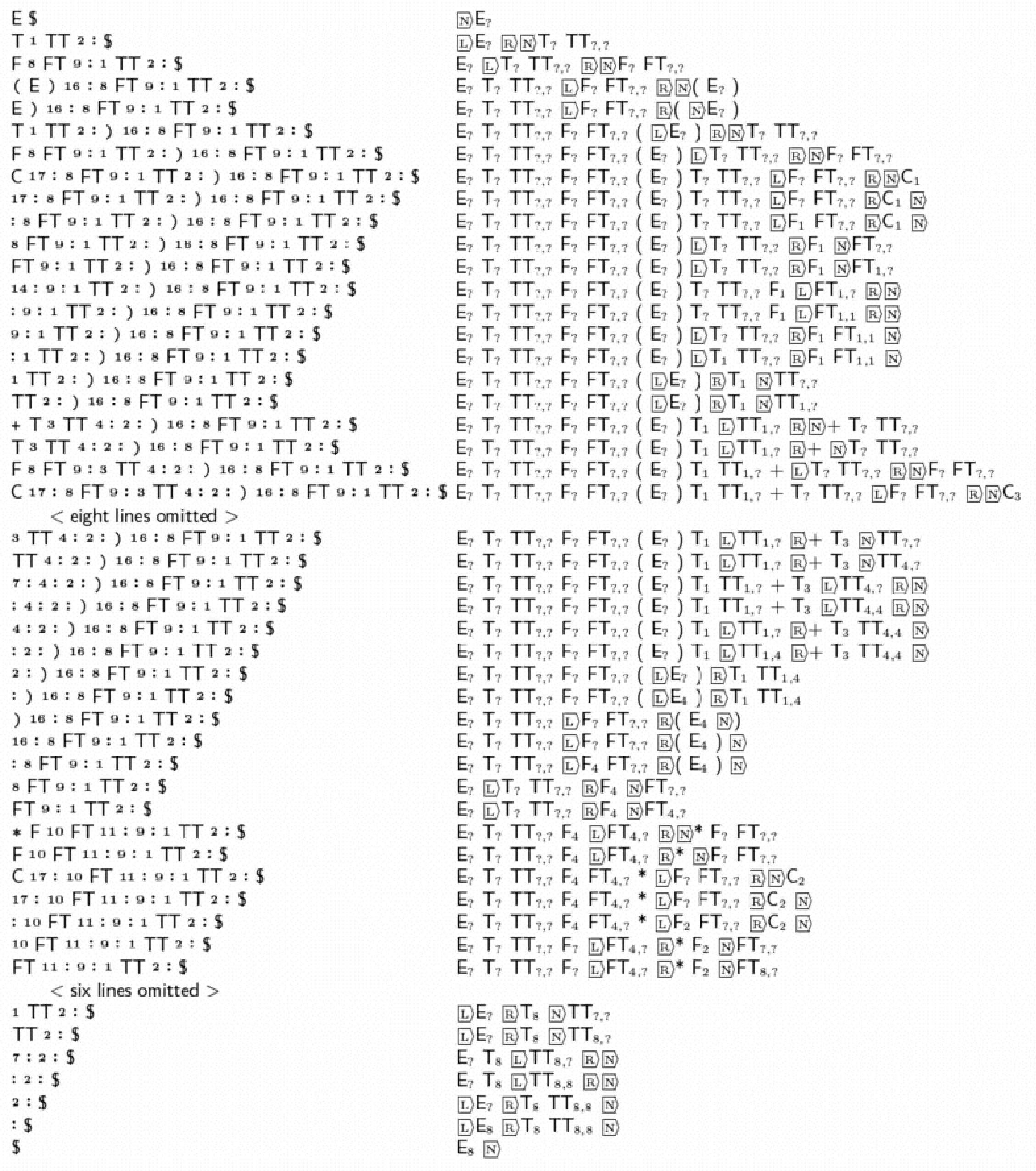
$\boxed{L}\rangle$ – attributes of left-hand side symbol of current production

$\boxed{R}\rangle$ – first symbol of the right-hand side of the production

$\boxed{L}\rangle$ and $\boxed{R}\rangle$ together allow the action routines to find the attributes of the symbols of the current production

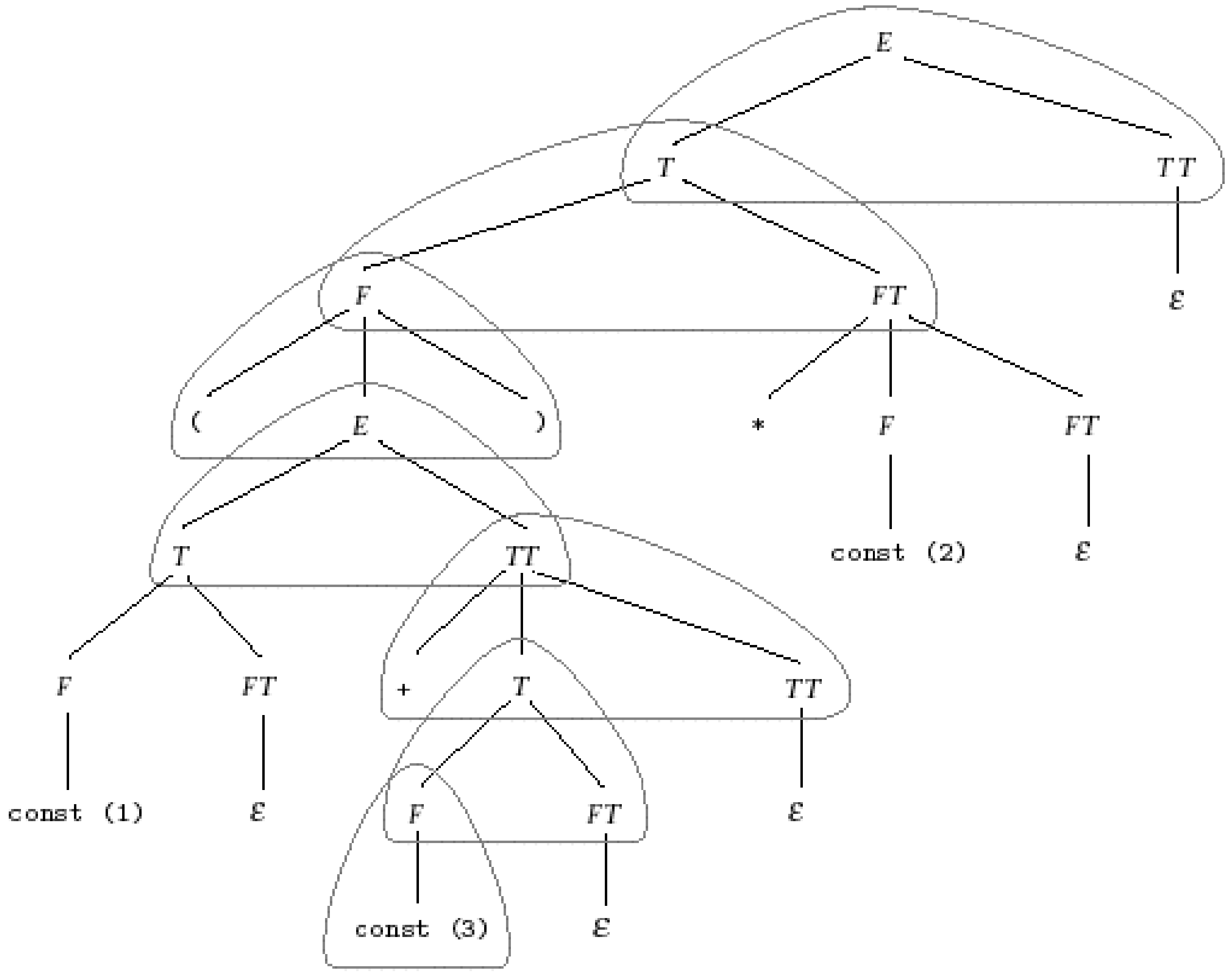
$\boxed{N}\rangle$ – first symbol within the right-hand side that has not yet been completely parsed

$\boxed{N}\rangle$ allows the parser to correctly update $\boxed{L}\rangle$ when a production is predicted



`E $`	$\boxed{N}\rangle E_?$
`T 1 TT 2 : $`	$\boxed{L}\rangle E_?\ \boxed{R}\rangle\boxed{N}\rangle T_?\ TT_{?,?}$
`F 8 FT 9 : 1 TT 2 : $`	$E_?\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle\boxed{N}\rangle F_?\ FT_{?,?}$
`( E ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle\boxed{N}\rangle (\ E_?\ )$
`E ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle (\ \boxed{N}\rangle E_?\ )$
`T 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\rangle E_?\ )\ \boxed{R}\rangle\boxed{N}\rangle T_?\ TT_{?,?}$
`F 8 FT 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle\boxed{N}\rangle F_?\ FT_{?,?}$
`C 17 : 8 FT 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle\boxed{N}\rangle C_1$
`17 : 8 FT 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle C_1\ \boxed{N}\rangle$
`: 8 FT 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_1\ FT_{?,?}\ \boxed{R}\rangle C_1\ \boxed{N}\rangle$
`8 FT 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle F_1\ \boxed{N}\rangle FT_{?,?}$
`FT 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle F_1\ \boxed{N}\rangle FT_{1,?}$
`14 : 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ F_1\ \boxed{L}\rangle FT_{1,?}\ \boxed{R}\rangle\boxed{N}\rangle$
`: 9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ F_1\ \boxed{L}\rangle FT_{1,1}\ \boxed{R}\rangle\boxed{N}\rangle$
`9 : 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle F_1\ FT_{1,1}\ \boxed{N}\rangle$
`: 1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\rangle T_1\ TT_{?,?}\ \boxed{R}\rangle F_1\ FT_{1,1}\ \boxed{N}\rangle$
`1 TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\rangle E_?\ )\ \boxed{R}\rangle T_1\ \boxed{N}\rangle TT_{?,?}$
`TT 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\rangle E_?\ )\ \boxed{R}\rangle T_1\ \boxed{N}\rangle TT_{1,?}$
`+ T 3 TT 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\rangle TT_{1,?}\ \boxed{R}\rangle\boxed{N}\rangle +\ T_?\ TT_{?,?}$
`T 3 TT 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\rangle TT_{1,?}\ \boxed{R}\rangle +\ \boxed{N}\rangle T_?\ TT_{?,?}$
`F 8 FT 9 : 3 TT 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle\boxed{N}\rangle F_?\ FT_{?,?}$
`C 17 : 8 FT 9 : 3 TT 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle\boxed{N}\rangle C_3$
< eight lines omitted >	
`3 TT 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\rangle TT_{1,?}\ \boxed{R}\rangle +\ T_3\ \boxed{N}\rangle TT_{?,?}$
`TT 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\rangle TT_{1,?}\ \boxed{R}\rangle +\ T_3\ \boxed{N}\rangle TT_{4,?}$
`7 : 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ T_3\ \boxed{L}\rangle TT_{4,?}\ \boxed{R}\rangle\boxed{N}\rangle$
`: 4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ T_3\ \boxed{L}\rangle TT_{4,4}\ \boxed{R}\rangle\boxed{N}\rangle$
`4 : 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\rangle TT_{1,?}\ \boxed{R}\rangle +\ T_3\ TT_{4,4}\ \boxed{N}\rangle$
`: 2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\rangle TT_{1,4}\ \boxed{R}\rangle +\ T_3\ TT_{4,4}\ \boxed{N}\rangle$
`2 : ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\rangle E_?\ )\ \boxed{R}\rangle T_1\ TT_{1,4}$
`: ) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\rangle E_4\ )\ \boxed{R}\rangle T_1\ TT_{1,4}$
`) 16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle (\ E_4\ \boxed{N}\rangle)$
`16 : 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle (\ E_4\ )\ \boxed{N}\rangle$
`: 8 FT 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\rangle F_4\ FT_{?,?}\ \boxed{R}\rangle (\ E_4\ )\ \boxed{N}\rangle$
`8 FT 9 : 1 TT 2 : $`	$E_?\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle F_4\ \boxed{N}\rangle FT_{?,?}$
`FT 9 : 1 TT 2 : $`	$E_?\ \boxed{L}\rangle T_?\ TT_{?,?}\ \boxed{R}\rangle F_4\ \boxed{N}\rangle FT_{4,?}$
`* F 10 FT 11 : 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_4\ \boxed{L}\rangle FT_{4,?}\ \boxed{R}\rangle\boxed{N}\rangle *\ F_?\ FT_{?,?}$
`F 10 FT 11 : 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_4\ \boxed{L}\rangle FT_{4,?}\ \boxed{R}\rangle *\ \boxed{N}\rangle F_?\ FT_{?,?}$
`C 17 : 10 FT 11 : 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_4\ FT_{4,?}\ *\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle\boxed{N}\rangle C_2$
`17 : 10 FT 11 : 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_4\ FT_{4,?}\ *\ \boxed{L}\rangle F_?\ FT_{?,?}\ \boxed{R}\rangle C_2\ \boxed{N}\rangle$
`: 10 FT 11 : 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_4\ FT_{4,?}\ *\ \boxed{L}\rangle F_2\ FT_{?,?}\ \boxed{R}\rangle C_2\ \boxed{N}\rangle$
`10 FT 11 : 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ \boxed{L}\rangle FT_{4,?}\ \boxed{R}\rangle *\ F_2\ \boxed{N}\rangle FT_{?,?}$
`FT 11 : 9 : 1 TT 2 : $`	$E_?\ T_?\ TT_{?,?}\ F_?\ \boxed{L}\rangle FT_{4,?}\ \boxed{R}\rangle *\ F_2\ \boxed{N}\rangle FT_{8,?}$
< six lines omitted >	
`1 TT 2 : $`	$\boxed{L}\rangle E_?\ \boxed{R}\rangle T_8\ \boxed{N}\rangle TT_{?,?}$
`TT 2 : $`	$\boxed{L}\rangle E_?\ \boxed{R}\rangle T_8\ \boxed{N}\rangle TT_{8,?}$
`7 : 2 : $`	$E_?\ T_8\ \boxed{L}\rangle TT_{8,?}\ \boxed{R}\rangle\boxed{N}\rangle$
`: 2 : $`	$E_?\ T_8\ \boxed{L}\rangle TT_{8,8}\ \boxed{R}\rangle\boxed{N}\rangle$
`2 : $`	$\boxed{L}\rangle E_?\ \boxed{R}\rangle T_8\ TT_{8,8}\ \boxed{N}\rangle$
`: $`	$\boxed{L}\rangle E_8\ \boxed{R}\rangle T_8\ TT_{8,8}\ \boxed{N}\rangle$
`$`	$E_8\ \boxed{N}\rangle$
]

example (continued) – productions with symbols currently in the attribute stack at the point we are about to parse 3



- ad-hoc management

- problem with automatic management – too much copying
- manage an ad hoc semantic stack within the action routines

example

$$
\begin{array}{lll}
E & \longrightarrow & T\ TT \\
TT & \longrightarrow & +\ T\ \{\ \text{bin\_op}\ (\text{"+"})\ \}\ TT \\
TT & \longrightarrow & -\ T\ \{\ \text{bin\_op}\ (\text{"−"})\ \}\ TT \\
TT & \longrightarrow & \epsilon \\
T & \longrightarrow & F\ FT \\
FT & \longrightarrow & *\ F\ \{\ \text{bin\_op}\ (\text{"}\times\text{"})\ \}\ FT \\
FT & \longrightarrow & /\ F\ \{\ \text{bin\_op}\ (\text{"}\div\text{"})\ \}\ FT \\
FT & \longrightarrow & \epsilon \\
F & \longrightarrow & -\ F\ \{\ \text{un\_op}\ (\text{"}^{+}/_{-}\text{"})\ \} \\
F & \longrightarrow & (\ E\ ) \\
F & \longrightarrow & \texttt{const}\ \{\ \text{push\_leaf}\ (\text{cur\_tok.val})\ \}
\end{array}
$$

- the semantic stack contains pointers to syntax tree nodes
- advantages - fewer rules (and therefore faster)
  - elimination of inherited attributes
- disadvantages - compiler writer must remember what is in the stack (push, pop appropriately)
  - the automatic approach is more regular and easier to maintain

<u>**example**</u> – another advantage of ad-hoc semantic stack

- it allows action routines to push or pop unlimited number of records
- automatic management – number of records that can be seen is limited by the number of symbols in the current production

- automatic management

$dec \longrightarrow id\_list$ `:` $type$ { declare_vars(id_list.chain, type.tp) }
$id\_list \longrightarrow$ `id` id_list_tail
{ id_list.chain := append(id.name, id_list_tail.chain) }
$id\_list\_tail \longrightarrow$ `,` id_list { id_list_tail.chain := id_list.chain }
$\longrightarrow \epsilon$ { id_list_tail.chain := nil }

- ad-hoc management – eliminate the linked lists

```
dec → { push(marker) }
        id_list : type
        { pop(tp)
          pop(name)
          while name <> marker
            declare_var(name,tp)
            pop(name) }
id_list → id { push(cur_tok.name) } id_list_tail
id_list_tail → , id_list
             → ε
```

3.6 Annotating a syntax tree

- attribute grammars can be used to annotate syntax trees
  - the compiler constructs the syntax tree
  - the semantic analysis and intermediate code generation will use the syntax tree as base

example – bottom-up grammar for calculator language with types

program ⟶ *stmt_list* $$

stmt_list ⟶ *stmt_list decl* | *stmt_list stmt* | ϵ

decl ⟶ `int id` | `real id`

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

expr ⟶ *term* | *expr add_op term*

term ⟶ *factor* | *term mult_op factor*

factor ⟶ `(` *expr* `)` | `id` | `int_const` | `real_const` | `float (` *expr* `)` | `trunc (` *expr* `)`

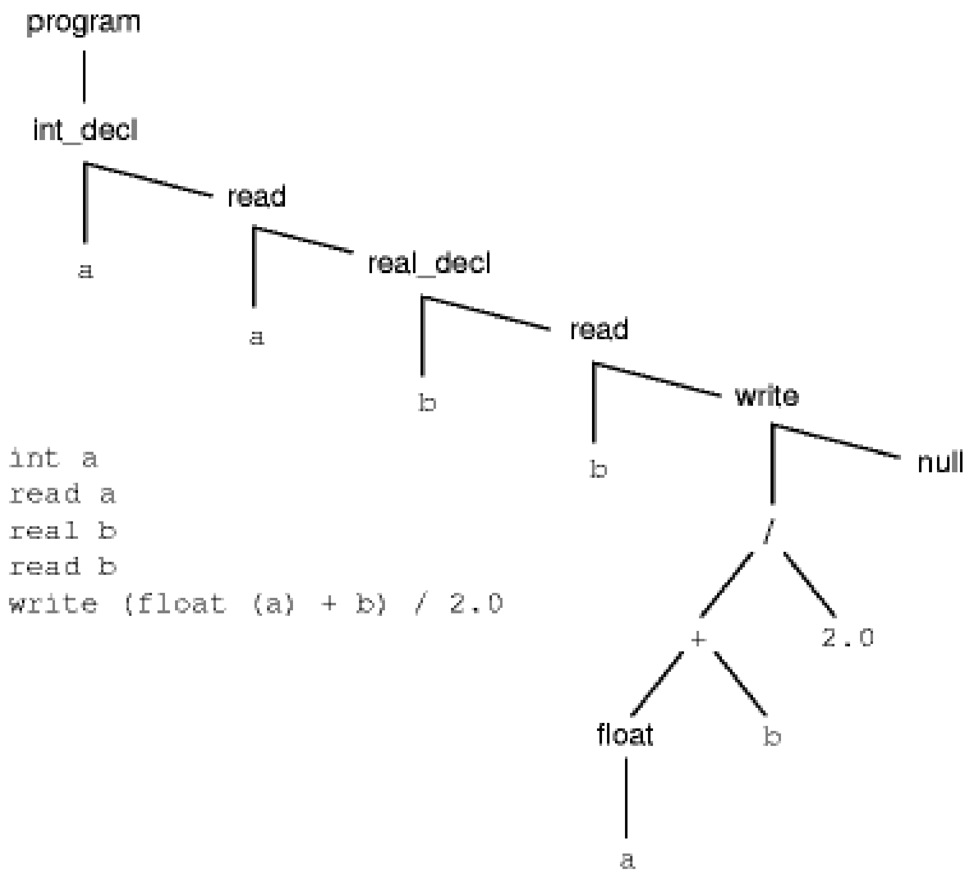
add_op ⟶ `+` | `-`

mult_op ⟶ `*` | `/`

- the intended semantics requires that every identifier be declared before it is used and types not be mixed in operations

example – continued

- assuming we added semantic functions or action routines to the grammar to construct a syntax tree, a concrete example would look as below



```
int a
read a
real b
read b
write (float (a) + b) / 2.0
```

3.6.1 Tree grammars

- tree grammars describe the structure of syntax trees (as context-free grammars describe possible structure of parse trees)
- semantic rules attached to the productions of a tree grammar can be used to define the attribute flow of a syntax tree (as done with context-free grammars for parse trees)

example – fragment of tree grammar needed to handle the above program

$program \longrightarrow item$

$int\_decl : item \longrightarrow \texttt{id}\ item$

$read : item \longrightarrow \texttt{id}\ item$

$real\_decl : item \longrightarrow \texttt{id}\ item$

$\texttt{write} : item \longrightarrow expr\ item$

$\texttt{null} : item \longrightarrow \epsilon$

$\texttt{/} : expr \longrightarrow expr\ expr$

$\texttt{+} : expr \longrightarrow expr\ expr$

$\texttt{float} : expr \longrightarrow expr$

$\texttt{id} : expr \longrightarrow \epsilon$

$real\_const : expr \longrightarrow \epsilon$

$A : B$ on the left-hand side means A is one kind of B and may appear whenever B is expected in the right-hand side

- next two pages – complete tree attribute grammar for the calculator language with types

```
program ⟶ item
    ▷ item.symtab := nil                      ▷ item.errors_in := nil
    ▷ program.errors := item.errors_out
int_decl : item_1 ⟶ id item_2
    ▷ declare_name (id, item_1, item_2, int)    ▷ item_1.errors_out := item_2.errors_out
real_decl : item_1 ⟶ id item_2
    ▷ declare_name (id, item_1, item_2, real)   ▷ item_1.errors_out := item_2.errors_out
read : item_1 ⟶ id item_2
    ▷ item_2.symtab := item_1.symtab
    ▷ if <id.name, ?> ∈ item_1.symtab
          item_2.errors_in := item_1.errors_in
      else
          item_2.errors_in := item_1.errors_in + [id.name "undefined at" id.location]
    ▷ item_1.errors_out := item_2.errors_out
write : item_1 ⟶ expr item_2
    ▷ expr.symtab := item_1.symtab             ▷ item_2.symtab := item_1.symtab
    ▷ item_2.errors_in := item_1.errors_in + expr.errors
    ▷ item_1.errors_out := item_2.errors_out
':=' : item_1 ⟶ id expr item_2
    ▷ expr.symtab := item_1.symtab             ▷ item_2.symtab := item_1.symtab
    ▷ if <id.name, A> ∈ item_1.symtab            -- for some type A
          if A <> error and expr.type <> error and A <> expr.type
              item_2.errors_in := item_1.errors_in + ["type clash at" item_1.location]
          else
              item_2.errors_in := item_1.errors_in
      else
          item_2.errors_in := item_1.errors_in + [id.name "undefined at" id.location]
    ▷ item_1.errors_out := item_2.errors_out
null : item ⟶ ε
    ▷ item.errors_out := item.errors_in
id : expr ⟶ ε
    ▷ if <id.name, A> ∈ expr.symtab              -- for some type A
          expr.errors := nil
          expr.type := A
      else
          expr.errors := [id.name "undefined at" id.location]
          expr.type := error
int_const : expr ⟶ ε
    ▷ expr.type := int
real_const : expr ⟶ ε
    ▷ expr.type := real
```

```
'+' : $expr_1 \longrightarrow expr_2\ expr_3$
      ▷ expr₂.symtab := expr₁.symtab      ▷ expr₃.symtab := expr₁.symtab
      ▷ check_types (expr₁, expr₂, expr₃)
'−' : $expr_1 \longrightarrow expr_2\ expr_3$
      ▷ expr₂.symtab := expr₁.symtab      ▷ expr₃.symtab := expr₁.symtab
      ▷ check_types (expr₁, expr₂, expr₃)
'×' : $expr_1 \longrightarrow expr_2\ expr_3$
      ▷ expr₂.symtab := expr₁.symtab      ▷ expr₃.symtab := expr₁.symtab
      ▷ check_types (expr₁, expr₂, expr₃)
'÷' : $expr_1 \longrightarrow expr_2\ expr_3$
      ▷ expr₂.symtab := expr₁.symtab      ▷ expr₃.symtab := expr₁.symtab
      ▷ check_types (expr₁, expr₂, expr₃)
float : $expr_1 \longrightarrow expr_2$
      ▷ expr₂.symtab := expr₁.symtab
      ▷ convert_type (expr₂, expr₁, int, real, "float of non-int")
trunc : $expr_1 \longrightarrow expr_2$
      ▷ expr₂.symtab := expr₁.symtab
      ▷ convert_type (expr₂, expr₁, real, int, "trunc of non-real")

macro declare_name (id, cur_item, next_item : syntax_tree_node; t : type)
    if <id.name, ?> ∈ cur_item.symtab
        next_item.errors_in := cur_item.errors_in + ["redefinition of" id.name "at" cur_item.location]
        next_item.symtab := cur_item.symtab − <id.name, ?> + <id.name, error>
    else
        next_item.errors_in := cur_item.errors_in
        next_item.symtab := cur_item.symtab + <id.name, t>
macro check_types (result, operand1, operand2)
    if operand1.type = error or operand2.type = error
        result.type := error
        result.errors := operand1.errors + operand2.errors
    else if operand1.type <> operand2.type
        result.type := error
        result.errors := operand1.errors + operand2.errors + ["type clash at" result.location]
    else
        result.type := operand1.type
        result.errors := operand1.errors + operand2.errors
macro convert_type (old_expr, new_expr : syntax_tree_node; from_t, to_t : type; msg : string)
    if old_expr.type = from_t or old_expr.type = error
        new_expr.errors :− old_expr.errors
        new_expr.type := to_t
    else
        new_expr.errors := old_expr.errors + [msg "at" old_expr.location]
        new_expr.type := error
```

4 Names

- binding time
- lifetime
- storage management
- scope rules
  - static
  - dynamic
- binding referencing environments
  - subroutine closure
  - first- and second-class subroutines

4.1 Binding time

- **binding** – association between two things
  - e.g., a name and the thing it names
- **binding time** – the time at which a binding is created

- **static** – before run time
  - early binding times – greater efficiency
  - e.g., compilation

- **dynamic** – at run time
  - late binding times – greater flexibility
  - e.g., implementation

4.2 Lifetimes

- distinguish between names and the objects they refer to
- key events
  - creation of objects
  - creation of bindings
  - references to variables, subroutines, types, etc.
    - all use bindings
  - deactivation/reactivation of bindings
  - destruction of bindings
  - destruction of objects
- **binding's lifetime** – the period of time between creation and destruction of binding
- **objects's lifetime** – the period of time between creation and destruction of object

- binding's lifetime and object's lifetime are not the same !!!

example - longer object's lifetime: a variable passes to a subroutine by reference

– the binding between the parameter name and the variable has a shorter lifetime than the variable itself

example - longer binding's lifetime (usually a program bug): a variable created by Pascal's `new` is passed by reference and the deallocated before the subroutine returns

- dangling reference – a binding to an object that is no longer live

4.3 Storage management

- storage allocation – managing object's space
- three types
  - *static* – objects are given an absolute address that is retained throughout the program's execution
  - *stack* – objects are allocated in last-in, first-out order, usually in conjunction with subroutine calls and returns
  - *heap* – objects may be allocated/deallocated at arbitrary times

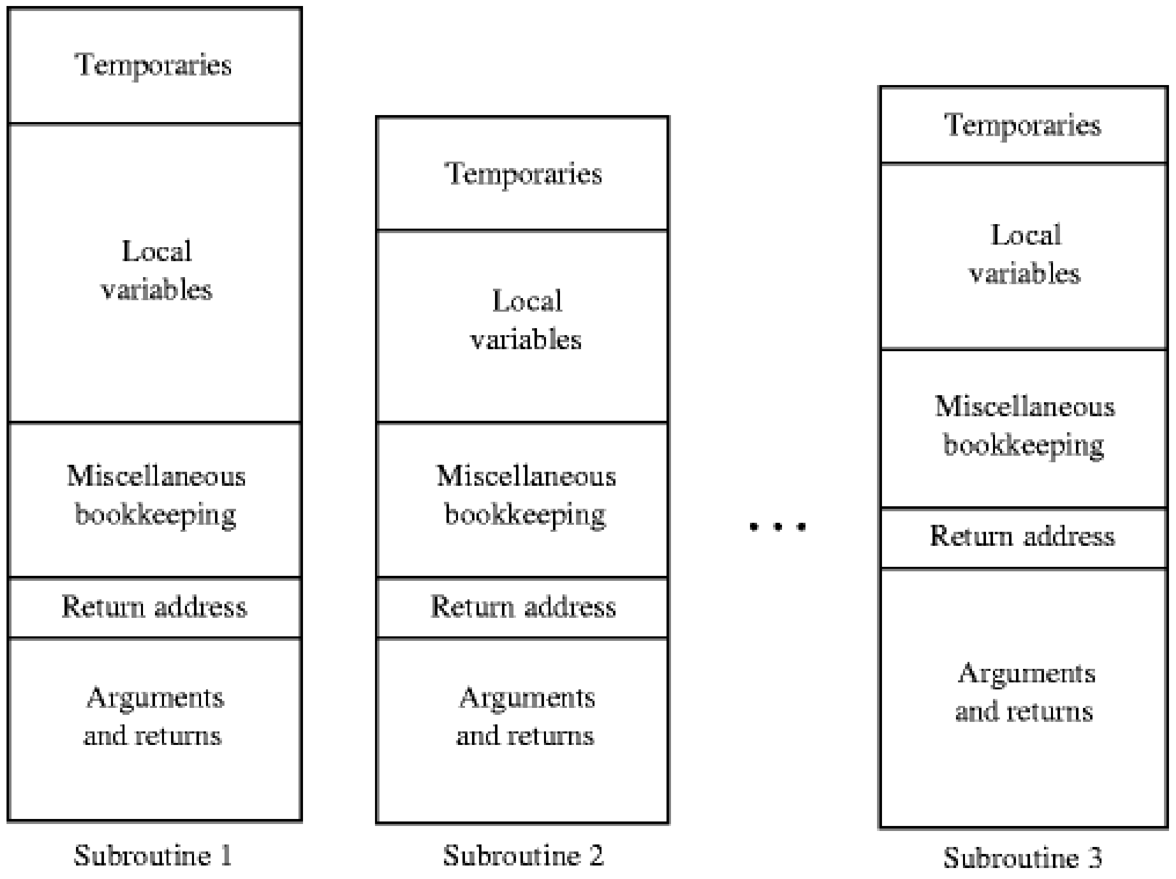
4.3.1 Static allocation

- global variables
- instructions
- local variables which retain value between calls (e.g., `static`)
- constants
- tables: for debugging, dynamic-type checking, garbage collection, exception handling, etc.

- statically allocated objects whose value should not change (e.g., instructions, constants, some tables) are often allocated in protected read-only memory

- limitations
  - does not allow recursion
  - data structures cannot be created dynamically

example - static allocation for subroutines in a language without recursion



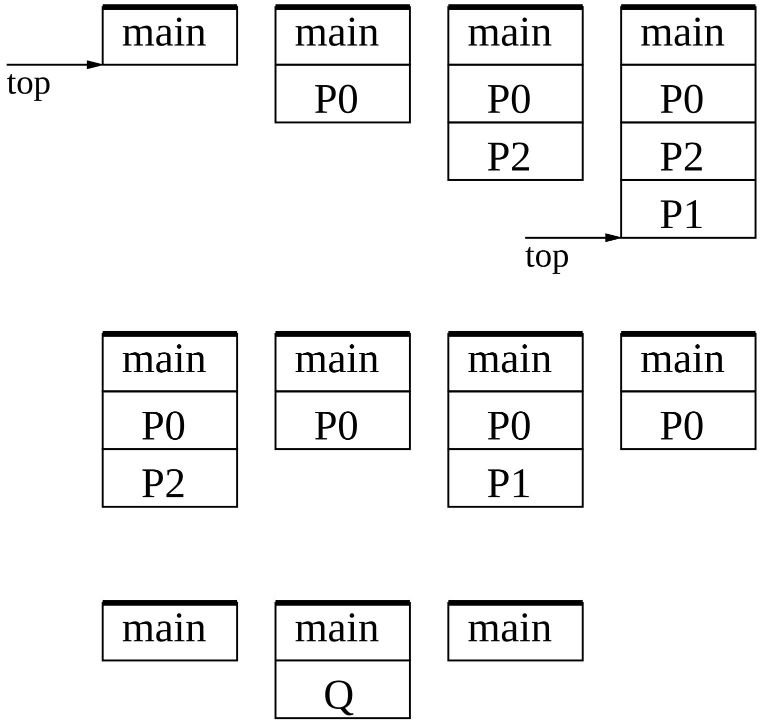
- in a language without recursion, all variables can be statically allocated

4.3.2 Stack-based allocation

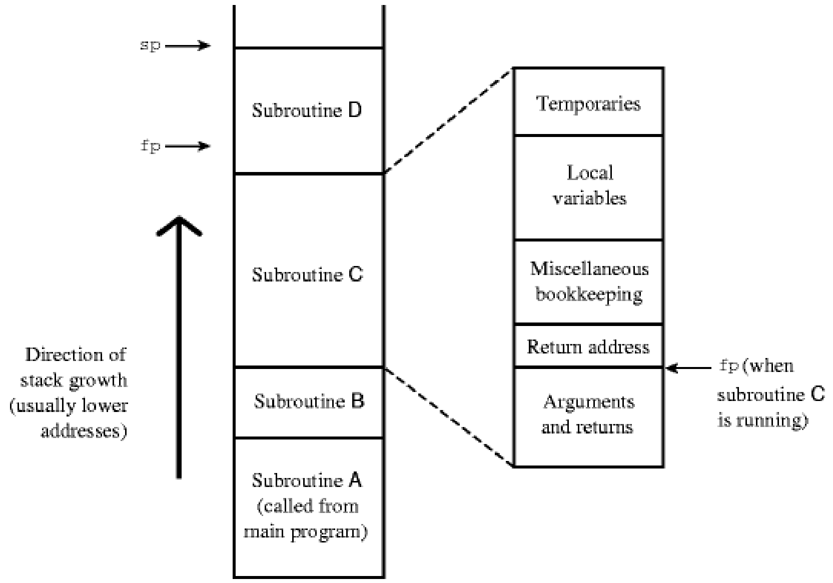
- if recursion is permitted, stack allocation is needed – the number of instances of a variable that may exist at the same time is conceptually unbounded
- the natural nesting of subroutine calls makes it easy to allocate space for locals on a stack

example

```
program main;
begin
|..........
|  procedure P0;
|  begin
|  |   .......
|  |  procedure P1;
|  |    begin.....end
|  |  procedure P2;
|  |    begin P1;...end
|  |  P2; P1;
|  |   ..........
|  end
|  procedure Q;
|    begin....end
|  P0; Q;
end
```



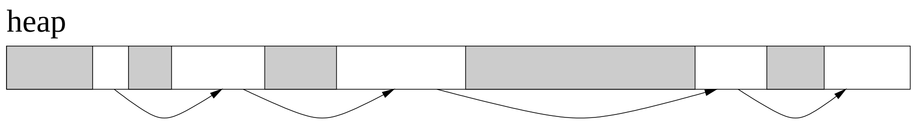
- each instance of a subroutine at run time has its own frame
  - **activation record**
  - return values, local variables, temporaries, bookkeeping information (e.g., return address – *dynamic link*, saved registers)


sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine A (called from main program)
Direction of stack growth (usually lower addresses)
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
Arguments and returns
fp (when subroutine C is running)

- maintenance of stack
  - by the *calling sequence* – code at the beginning and at the end of subroutines
  - the location of the stack frame cannot be predicted but the offsets (w.r.t. the *frame pointer*) of objects within a frame can

4.3.3 Heap-based allocation

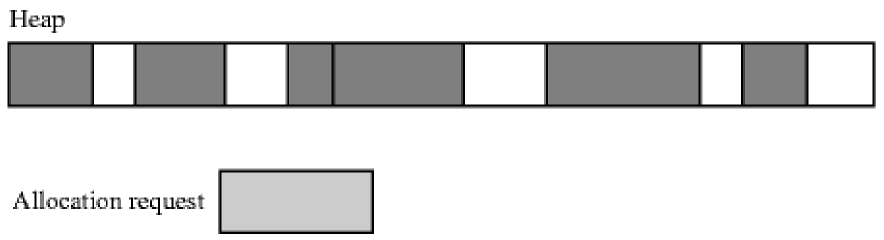
- **heap** – region of storage where blocks can be (de)allocated at any time
- heap allocation required for
  - dynamically allocated data structures
  - dynamically resized objects
- managing space in a heap
  - space vs speed



  - single linked list of unused blocks – the **free list**

- internal fragmentation – when the block allocated from heap is bigger than the object; the extra space is unused

- external fragmentation – when the used blocks are scattered in such a way that the unused space is composed of multiple blocks
  - there may be a lot of free space but no one piece large enough for the request



- allocation algorithms

▶ **first fit**

- select the first block which is large enough

▶ **best fit**

- select the smallest block which is large enough

- when allocating, if the chosen block is much bigger, then it can be split in two and the unneeded portion returned to the free list
- when deallocating, if two physically adjacent blocks are free we make one block of those

▶ **pools**

- separate free lists for blocks of different size
- idea is to reduce the linear allocation time to constant
- **buddy system**
  - block sizes – powers of 2
  - if no block of size 2^k, then take one of size 2^{k+1} and split into two
- **Fibonacci heap**
  - block sizes – Fibonacci numbers

4.4 Scope rules

- scope of a binding – the textual region of the program in which the binding is active

- *static (lexical)* – determined statically, at compile time

    - e.g., when entering a subroutine a new scope is created; local bindings are created and global ones might be deactivated (hidden)

- *dynamic* – bindings depend on execution

- other definitions

- *scope* – a program region in which no bindings change

- *elaboration* – the process by which declarations become active when control first enters a scope

- *referencing environment* – the set of active bindings at any given point in a program's execution

    - determined by *scope rules* and *binding rules*

4.4.1 Static scope

- bindings – determined at compile time
  - the current binding for a given name is the one encountered most recently in a top-to-bottom scan of the program
- used by most modern programming languages

- closest nested scope rule – a name is known in the scope in which it is declared and in each internally nested scopes, unless *hidden* by another declaration of the same name
- global scope – at the level of the program
- outermost scope – surrounds the global scope
  - for built-in objects, e.g., I/O routines, trigonometric functions, built-in types, etc.

- finding a binding – search from innermost to outermost; the first found is good

example

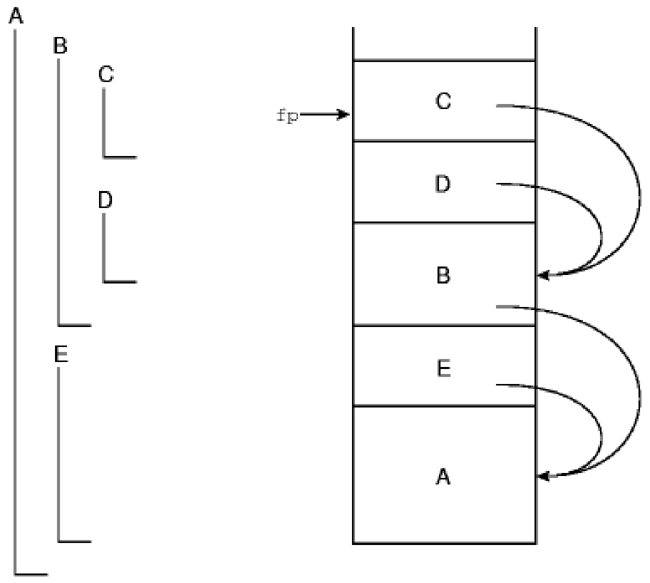
- F1 can call P2
- P4 can call F1
- P2 cannot call F1
- F1's X makes a *hole* in the scope of P1's X
  - scope resolution operator can help access the outer meaning (in C '::')

```
procedure P1 (A1 : T1);
var X : real;
    ...
    procedure P2 (A2 : T2);
        ...
        procedure P3 (A3 : T3);
        ...
        begin
            ...       (* body of P3 *)
        end;
        ...
    begin
        ...           (* body of P2 *)
    end;
    ...
    procedure P4 (A4 : T4);
        ...
        function F1 (A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...       (* body of F1 *)
        end;
        ...
    begin
        ...           (* body of P4 *)
    end;
    ...
begin
    ...               (* body of P1 *)
end
```

- static links

- used to find objects in lexically surrounding scopes at run time
- a static link points to the frame of the most recent activation of the lexically surrounding subroutine
    - for outermost level - nil
- to find an object declared in a scope j lexical nesting levels outward, we have to dereference the static chain j times
- different from *dynamic links*

example



4.4.2 Dynamic scope

- bindings – determined at run time

  - the current binding for a given name is the one encountered most recently during execution and not yet destroyed by returning from its scope

  - cannot be determined at compile time

  - many semantic checks must be deferred until run time (such languages tend to be interpreted rather than compiled)

example - static vs dynamic scope

- static scope – prints 1
- dynamic scope – prints 1 or 2, depending on the value read at run time

```
a : integer          -- global declaration

procedure first
      a := 1

procedure second
      a : integer          -- local declaration
      first ()

a := 2
if read_integer () > 0
      second ()
else
      first ()
write_integer (a)
```

- advantage
  - very easy for an interpreter to look up the meaning of a name – keep a stack of declarations
- disadvantages
  - very high run-time cost
  - programs are harder to understand
    - no program fragment that makes use of non-local names is guaranteed a predictable referencing environment

example - of a problem with dynamic scoping

- `max_score` accidentally redefined
- global is `integer`; local is `real` – the error might be detected by dynamic semantic checks
- if local were `integer`, then it would be very difficult to detect the error

```
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 percentage seen so far
    ...
    foreach student in class
        student.percent := scaled_score (student.points)
        if student.percent > max_score
            max_score := student.percent
```

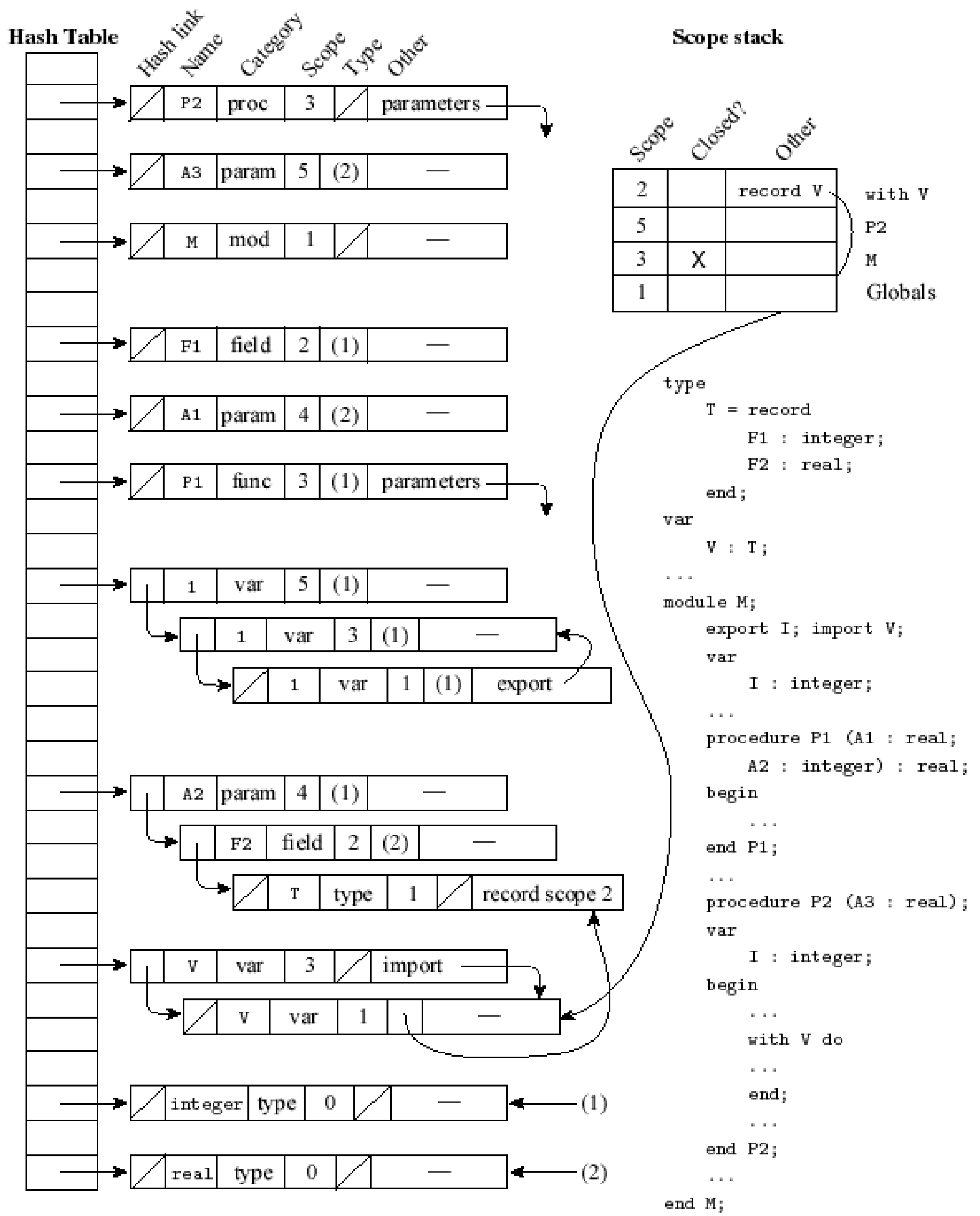
4.4.3 Symbol tables

- data structure needed to keep track of the names in statically scoped programs
- maps names to information the compiler knows about them
- basic operations
  - `insert` – place a new name-to-object binding into the table
  - `lookup` – retrieve (nondestructively) the information for a given name

- LeBlanc-Cook symbol table

- dictionary data structure with `enter_scope`, `leave_scope` operations for visibility
- each scope, as encountered, is assigned a serial number
  - outermost – 0; global – 1
- *hash table* – contains all names with category (variable, constant type, procedure, field name, parameter, etc.), scope number, type (a pointer to another symbol table entry), category-specific fields
- *scope stack* – contains, in order, the scope that comprise the current referencing environment; the semantic analyzer pushes/pops the stack when enters/leaves a scope
  - each entry – scope number, whether closed or not, further information

example



```
type
    T = record
        F1 : integer;
        F2 : real;
    end;
var
    V : T;
...
module M;
    export I; import V;
    var
        I : integer;
    ...
    procedure P1 (A1 : real;
        A2 : integer) : real;
    begin
        ...
    end P1;
    ...
    procedure P2 (A3 : real);
    var
        I : integer;
    begin
        ...
        with V do
        ...
        end;
        ...
    end P2;
    ...
end M;
```

- lookup

- match the name
- look into stack to see if the scope is visible
- look no further than top-most closed scope

```
procedure lookup (name)
    pervasive := best := nil
    apply hash function to name to find appropriate chain
    foreach entry e on chain
        if e.name = name        -- not something else with same hash value
            if e.scope = 0
                pervasive := e
            else
                foreach scope s on scope stack, top first
                    if s.scope = e.scope
                        best := e        -- closer instance
                    elsif best <> nil and then s.scope = best.scope
                        exit inner loop        -- won't find better
                    if s.closed
                        exit inner loop        -- can't see farther
    if best <> nil
        while best is an import or export entry
            best := best.real_entry
        return best
    elsif pervasive <> nil
        return pervasive
    else
        return nil        -- name not found
```

4.4.4 Association lists, central reference tables

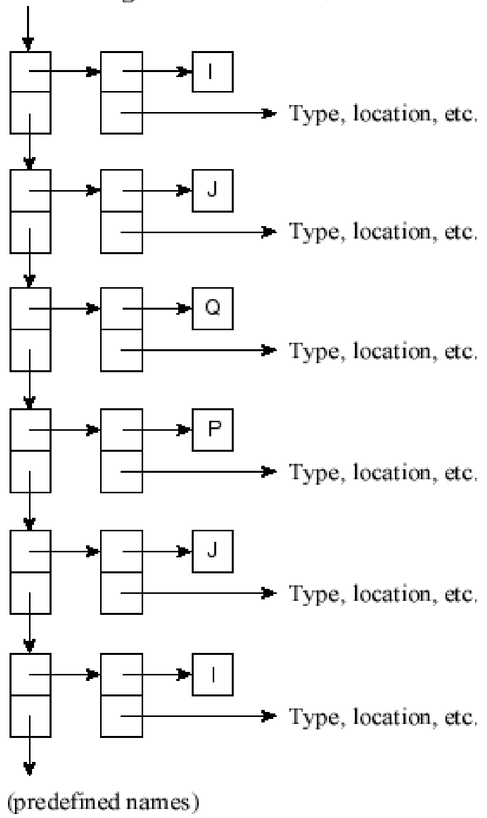
- data structure for keeping track of the names in dynamically scoped programs
- the interpreter must perform at run time the operations corresponding to `insert`, `lookup`, `enter_scope`, `leave_scope`

▶ with **association lists** (**A-lists**)
- a list of name/value pairs – functions like a stack
    - when execution enters a scope at run time, the interpreter pushes bindings for names declared in that scope onto the front of the A-list
    - when execution leaves a scope, these bindings are removed
- lookup – search from the front of the list until finding an appropriate binding
- problem – slow

▶ with **central reference tables**
- stack of entries for each different name; most recent occurrence at beginning
- update is somewhat slower
    - when control enters a new scope at run time, entries must be pushed at the beginning of each list whose name is redeclared
    - when control leaves a scope, these entries must be popped
- lookup is much faster

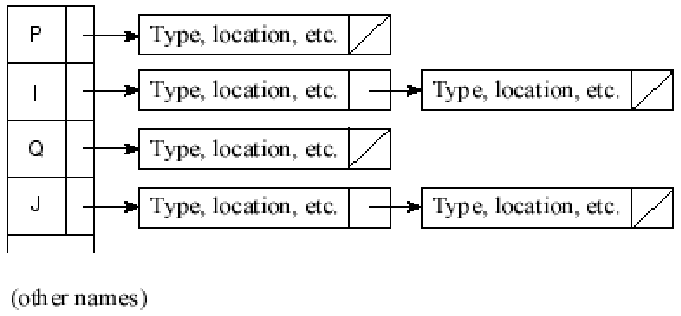
example

Referencing environment A-list



```
I, J : integer

procedure P (I : integer)

      . . .

Procedure Q
      J : integer
      . . .
      P (J)
      . . .

––    main program
. . .
Q
```

Central reference table



4.5 Binding of reference environment

problem: in a language that allows one to create a *reference to a subroutine*, e.g., by passing it as a parameter, when are the scope rules applied to to such a subroutine?
(i) when the reference is first created
(ii) when the routine is finally called

(i) **deep binding** – early binding of the referencing environment
- bind the referencing environment at the time the subroutine is first passed as parameter and then restore that environment when the routine is called
- is the default in languages with static scoping

(ii) **shallow binding** – late binding of the referencing environment
- bind the referencing environment when the subroutine is called
- is the default in languages with dynamic scoping

example

- deep binding is appropriate for `older_than`
- shallow binding is appropriate for `print_person`

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

function older_than (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 non-local variable line_length to format
    –– data in columns.
    . . .

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, print_person)
```

4.5.1 Subroutine closures

- closure

- used to implement deep binding

- consists of an explicit representation of the referencing environment in which the subroutine would execute if called at the present time and a reference to the subroutine

- with association lists

- the referencing environment in a closure can be represented by a beginning of A-list pointer

- when a subroutine is called through a closure, the main pointer to the A-list is replaced by the saved pointer; restored when the subroutine returns

with central reference tables

- save the environment by copying the main array and the first entries of the lists for the interesting names

- when the subroutine is called through a closure, these entries can be pushed onto the beginnings of the appropriate lists

- deep binding with static scoping
  - binding time of referencing environments matters in languages with static scoping because a running program may have more than one instance of an object that is declared in a recursive subroutine; a closure captures the current instance (when created), which will be used when calling the subroutine, ignoring subsequently created instances
  - matters only when accessing objects whcih are neither local nor global
    - local – created when the subroutine started running
    - global – there is only one instance
  - irrelevant for C (no nested subroutines) or Modula-2 (only outermost subroutines can be passed) or in languages that do not permit subroutines to be passed at all

  - implementation
    - to represent a closure for a subroutine, we save a pointer to its code together with the static link that would be used if called now
    - when called, we temporarily restore the saved static link

- shallow binding with static scoping - doesn't make much sense

example – deep vs shallow binding with static scoping

- with deep binding, it prints 1
- with shallow binding, it prints 2

```
program BindingExample (input, output);

procedure A (I : integer; procedure P);

    procedure B;
    begin
        writeln (I);
    end;

begin (* A *)
    if I > 1 then
        P
    else
        A (2, B);
end;

procedure C; begin end;

begin (* main *)
    A (1, C);
end.
```

5 Control flow

- expressions
  - evaluation
  - ordering
  - short-circuit evaluation
- sequencing
  - structural programming
- selection
  - short-circuit conditions
  - `case` statements
- iteration
- recursion
  - versus iteration
  - tail recursion elimination

- control flow – ordering in program execution

- language mechanisms used to specify ordering
  - sequencing – statements executed in the order they appear
  - selection
  - iteration
  - procedural abstraction
  - recursion
  - concurrency – program fragments executed at the same time
- imperative languages
  - sequencing is central
  - iteration is very important
- functional languages
  - sequencing has a minor role
  - recursion very heavily used
- logic languages
  - control flow completely hidden
  - rules are specified and the implementation finds the order

5.1 Expressions

5.1.1 Expression evaluation

- **infix** – $E_1 \odot E_2$ - used for binary operators
- **prefix** – $\odot E_1 E_2 \ldots E_k$
  - used for unary operators and functions
  - used for all functions in Lisp: `(* (+ 1 3) 2)`
- **postfix** – $E_1 E_2 \ldots E_k \odot$
  - used in Postscript, Forth
  - post-increment/decrement operators in C – `i++ i--`
- **mixfix** (**multiword infix**) – `a = b != 0 ?  a/b :  0`

- <u>ambiguity</u> – in infix expressions without parentheses
- parentheses reduced by precedence and associativity
- **precedence** – certain operators group more tightly than others
- **associativity** – operators of the same precedence group to the right or left
  - most operators – left associative
  - exponentiation, assignment – right associative

  `4**3**2 = 4**(3**2)`   `a = b = a+c` is `a = (b = a+c)`

5.1.2 Ordering within expressions

- the order in which the operands are evaluated
- important for two reasons

▸ **side effects** – any modification by a programming construct other than returning a value

  - imperative programming – computing by side effects
  - pure functional programming – no side effects

`x + f(x)` – if `f` modifies `x` order is important

`f(a,g(b),c)` – if `g(b)` modifies `a` or `c` order is important

▸ **code improvement**

`a*b+f(c)` – better call `f` before evaluating `a*b` because otherwise `a*b` has to be saved in a register which `f` cannot use

- some languages impose a certain order (Java - left-to-right)
- some others leave it to the compiler to chose for a faster code
- some rearrange expressions (within mathematical rules) for faster code:

  – `a = b+c; d = c+e+b` is rearranged as `a = b+c; d = b+c+e` and have code generated as `a = b+c; d = a+e` (dangerous because of limited precision)

5.1.3 Short-circuit evaluation

- avoid evaluating all subexpressions of a boolean expression once the result is known

example `(a<b) and (b<c)`

- if `a>=b` there is no need to evaluate `(b<c)`

▶ can save time

example `if (unlikely_cond && expensive_f()) ...`

▶ can change the semantics of boolean expressions

example - the same code which works in C but not in Pascal

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

p := my_list;
while (p <> nil) and (p^.key <> val) do (*ouch!*)
  p := p^.next;
```

- when the subexpressions have important side effects, short-circuit is not wanted

example

```
function tally (word : string) : integer;
    (* Look up word in hash table.  If found, increment tally; If not
        found, enter with a tally of 1.  In either case, return tally. *)
...
function misspelled (word : string) : Boolean;
    (* Check to see if word is mis-spelled and return appropriate
        indication.  If yes, increment global count of mis-spellings. *)
...
while not eof (doc_file) do begin
    w := get_word (doc_file);
    if (tally (w) = 10) and misspelled (w) then
        writeln (w)
end;
writeln (total_misspellings);
```

- some languages provide both types of operators
- Ada
  - regular operators: `and` and `or`
  - short-circuit operators: `and then` and `or else`
- C
  - short-circuit operators: `&&` and `||`
  - regular (non-short-circuit) operators: `&` and `|`

5.2 Sequencing

- statements executed in the top-to-bottom order

5.2.1 Structured programming

- means no `goto`s
- the revolution of the 1970s

- cases when `goto` was considered useful

▶ mid-loop exit and continue
- `goto` to a label right after the end of the loop
- replaced by statements like `exit`, `break`, `continue`

▶ early return from subroutine
- `goto` to a label right before the end
- replaced by `return`

▶ errors and other exceptions
- nonlocal `goto`s – difficult to implement and understand
- replaced by exception handling mechanisms

5.3 Selection

5.3.1 Short-circuited conditions

- efficient code (*jump code*) for short-circuit evaluation

example - consider the source code

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

- without short-circuiting

```
     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   – label not used
     goto L3
L2:  else_clause
L3:
```

- with short-circuiting

```
     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:
```

5.3.2 Case/switch statements

- alternative syntax for nested `if...then...else`
- syntactic elegance
- efficient target code

```
i := ...  (* potentially complicated expression *)
IF i=1 THEN clause_A
ELSIF i IN 2,7 THEN clause_B
ELSIF i IN 3..5 THEN clause_C
ELSIF (i=10) THEN clause_D
ELSE clause_E
END
```

```
CASE ...  (* potentially complicated expression *)
  1:     clause_A
| 2,7:   clause_B
| 3..5:  clause_C
| 10:    clause_D
  ELSE   clause_E
END
```

- code for nested `if`s

```
     r1 := ...   – calculate tested expression
     if r1<>1 goto L1
     clause_A
     goto L6
L1:  if r1=2 goto L2
     if r1<>7 goto L3
L2:  clause_B
     goto L6
L3:  if r1<3 goto L4
     if r1>5 goto L4
     clause_C
     goto L6
L4:  if r1<>10 goto L5
     clause_D
     goto L6
L5:  clause_E
L6:
```

- code for `case` – see next page
  - the code for `T` is a table of address – *jump table*

```
       goto L6  – jump to code
                  to compute address
L1:    clause_A
       goto L7
L2:    clause_B
       goto L7
L3:    clause_C
       goto L7
L4:    clause_D
       goto L7
L5:    clause_E
       goto L7
T:     &L1  – tested expression = 1
       &L2
       &L3
       &L3
       &L3
       &L5
       &L2
       &L5
       &L5
       &L4  – tested expression = 10
```

```
L6:  r1 := ...    – calculate tested expression
     if r1<1 goto L5
     if r1>10 goto L5  – L5 is the "else" arm
     r1 -:= 1
     r2 := T[r1]
     goto *r2
L7:
```

- implementing case statements

▶ sequential testing
  - for small number of choices
  - runs in time $\mathcal{O}(n)$, n is the number of labels

▶ jump table
  - for dense set of labels, no large range
  - runs in time $\mathcal{O}(1)$

▶ hash table
  - for large non-dense range of label values, no large range
  - runs in time $\mathcal{O}(1)$

▶ binary search
  - accommodate ranges easily
  - runs in time $\mathcal{O}(\log n)$

5.4 Iteration

- iteration and recursion allow the computer to perform the same set of operations repeatedly
- without iteration and recursion, the running time of a program is a linear function of the size of the program; very weak computational power

- problems with loops
  - modifying the index in a `for`
  - modifying the bounds
  - jumps in and out of the loop; value of the index

5.5 Recursion

- recursive subroutine – a subroutine that is calling itself directly on indirectly

- linear recursive – if the activation of the function can directly initiate at most one new activation of the function.

- tail recursive – if it either returns a value without a recursive activation or it simply returns the result of a recursive activation

- tail recursive $\Rightarrow$ linear recursive $\Rightarrow$ recursive

example 1

$$\mathrm{fib}(n) = \ \textbf{if}\ (n = 0\ \textbf{or}\ n = 1)\ \ \begin{array}{l}\textbf{then}\ n \\ \textbf{else}\ \ \mathrm{fib}(n-1) + \mathrm{fib}(n-2)\end{array}$$

example 2

$$\mathrm{fact}(n) = \ \textbf{if}\ (n = 0)\ \ \begin{array}{l}\textbf{then}\ 1 \\ \textbf{else}\ n * f(n-1)\end{array}$$

example 3

$$g(n) = \ \textbf{if}\ n = 0\ \ \begin{array}{l}\textbf{then}\ 0 \\ \textbf{else}\ \ \textbf{if}\ n\%2 = 1\ \ \begin{array}{l}\textbf{then}\ g(n-1) + 2 \\ \textbf{else}\ g(n-2) * 2 + 1\end{array}\end{array}$$

example 4

$$h(n, a) = \ \textbf{if}\ n = 0\ \ \begin{array}{l}\textbf{then}\ a \\ \textbf{else}\ h(n-1, n * a)\end{array}$$

5.5.1 Iteration and recursion

- iteration and recursion provide equally powerful means of computation
- in general, iteration is more efficient; a good compiler might make the difference very small

example - the difference can be huge – Fibonacci numbers

- iterative version
    - takes linear time

```
int fib_it (int n) {
  int f1 = 1; int f2 = 1;
  int i;
  for (i = 2; i <= n; i++) {
    int temp = f1 + f2;
    f1 = f2; f2 = temp;
  }
  return f2;
}
```

- “bad” recursive version
    - takes exponential time

```
int fib_rec(int n) {
  if (n == 0) return 1;
  if (n == 1) return 1;
  return fib_rec(n-1) + fib_rec(n-2);
}
```

5.5.2 Tail recursion elimination

- iterative procedures are better than recursive ones
- elimination – save time and space

example

- given a tail-recursive procedure `P(x, y)`
- replace any tail-recursive call `P(a, b)` by

```
x = a; y = b;
goto the first executable statement in P
```

- general procedure:
- the tail-recursive

```
P(x1, x2,..., xn)
{...............
   return P(a1, a2,..., an)
}
```

- is replaced by:

```
P(x1, x2,..., xn)
{................          // declarations
L: .....               // first executable
                              statement in P
   ..................
   x1 = a1; x2 = a2; ...  xn = an;
   goto L;
}
```

example 1

```
int f(int n, int m)
{
  if (n == 0) return m;
  else return f(n - 1, n * m);
}
```

can be changed to the non-recursive:

```
int f(int n, int m)
{
L: if (n == 0) return m;
   else {
     n = n - 1; m = (n + 1) * m;
     goto L;
   }
}
```

or (without `goto`)

```
int f(int n, int m)
{
  while (n > 0) {
    n = n - 1; m = (n + 1) * m;
  }
  return m;
}
```

example 2 – `search` looks for `T` in the sorted array `X[N]`

```
int search(int lo, int hi)
{
  int k;
  if (lo > hi ) return 0;
  k = (lo + hi) / 2;
  if( T == X[k]) return 1;
  else if (T < X[k]) return search(lo, k - 1);
       else if (T > X[k]) return search(k + 1, hi);
}
```

the non-recursive variant is:

```
int search(int lo, int hi)
{
   int k;
L: if (lo > hi ) return 0;
   k = (lo + hi) / 2;
   if( T == X[k]) return 1;
   else if (T < X[k]) hi = k - 1;
        else if (T > X[k]) lo = k + 1;
   goto L;
}
```

or without goto ....

6 Types

- type systems
- type checking
  - type equivalence
  - type conversion and casts
  - type compatibility and coercion
  - type inference
- records and variants
- arrays
  - shape and allocation
  - memory layout
- pointers
  - dangling references
  - garbage collection

- types have two purposes

▶ provide context for operations (so that the programmer does not have to specify it)

  - `a + b` uses integer addition for `a, b` integers
  - `a + b` uses floating-point addition for `a, b` reals
  - `new p` allocates memory from heap of the right size
  - `new my_type()` allocates memory, calls a constructor

▶ limit the set of operations that may be performed in a semantically valid program

  - prevent adding a character and a record
  - prevent taking the arctangent of a set
  - passing a file to a subroutine that expects an integer

6.1 Type systems

- a **type system** consists of

▶ a **mechanism** for defining types and associating them with constructs

▶ a set of **rules** for *type equivalence*, *type compatibility*, and *type inference*

- type equivalence – determines when the types of two values are the same

- type compatibility – determines when a value of a given type can be used in a given context

- type inference – the type of an expression based on the types of its parts

- type checking – the process of ensuring that a program obeys the language's type compatibility rules

- type clash – violation of the type compatibility rules

- strongly typed language – prohibits the application of an operation to any object that is not intended to support it

- statically typed language – type checking is performed at compile time

  - Ada, Pascal, C

- dynamically typed languages – type checking is performed at run time

  - usually dynamically scoped languages – Lisp, Scheme

6.1.1 Definition of types

- three ways to think about types

▶ denotational
  - a type is a set of values
  - a value has a given type if it belongs to the set
  - an object has a type if its value is guaranteed to be in the set

▶ constructive
  - either a primitive (built-in) type
  - or composite; created by applying type constructors (`record`, `array`, `set`, etc)

▶ abstractional
  - a type is an interface consisting of a set of operations

6.1.2 Classification of types

- simple
  - built-in: integers, char, Boolean, rational, real, complex
  - user-defined: enumeration, subranges
- composite
  - records
  - variant records
  - arrays
  - sets
  - pointers
  - lists
  - files

- discrete type = countable, successor, predecessor
  - integers, Boolean, characters
- discrete types are simple

6.2 Type checking

6.2.1 Type equivalence

- structural equivalence

- based on content of definitions
- two types are the same if they consist of the same components, put together the same way
- used in Algol-68, Modula-3, C, ML, early Pascal

- name equivalence

- based on lexical occurrence of type definitions
- each definition introduces a new type
- used in Java, standard Pascal, Ada

- structural equivalence

- to determine whether the types are structurally equivalent, the compiler expands the definitions until they are left with a string of type constructors, field names, and built-in types; the resulting strings have to be the same

- problem: recursive and pointer-based types – their expansion does not terminate

example

```
type A = record
  x:  pointer to B
  y:  real
type B = record
  x:  pointer to A
  y:  real
```

- the expansion is infinite

```
type A = record x:  pointer to record x:  pointer to record x:  pointer to
record ...
```

- solution: based on finite automata theory
  - build a finite automaton for each type
  - build the smallest deterministic automaton
  - check the equivalence

example

- automaton for `type A`

$$\begin{tikzcd} {} \arrow[r] & \boxed{A} \arrow[r, shift left, "\texttt{record x:pointer to}"] & \boxed{B} \arrow[l, shift left, "\texttt{record x:pointer to}"] \arrow[r, "\texttt{y:real}"] & \boxed{\boxed{C}} \end{tikzcd}$$

- automaton for `type B`

$$\begin{tikzcd} {} \arrow[r] & \boxed{B} \arrow[r, shift left, "\texttt{record x:pointer to}"] & \boxed{A} \arrow[l, shift left, "\texttt{record x:pointer to}"] \arrow[r, "\texttt{y:real}"] & \boxed{\boxed{C}} \end{tikzcd}$$

- another problem: does not distinguish between types which accidentally have the same structure but are meant to be different

example

```
type student = record
  name, address:string;
  SIN:integer;
end;
type school = record
  name, address:= string;
  enrolment:integer;
end;
x:student;
y:school;
...
x := y;
```

– this is probably an error

- name equivalence

- assumption: if the programmer takes the effort to write two type definitions then those definitions are meant to represent different types

- problem: alias types
  - `type A = B` – are `A` and `B` the same types?

example 1 – where they should

```
TYPE stack_element = INTEGER;
MODULE stack;
IMPORT stack_element; ...
```

example 2 – where they should not

```
TYPE celsius_temp = INTEGER;
TYPE fahrenheit_temp = INTEGER;
```

example 3 – Ada specifies when a type is new

```
subtype stack_element is integer;
type celsius_temp is new integer;
type fahrenheit_temp is new integer;
```

- **strict name equivalence** – aliased types are distinct
  `type A = B` – is a definition
- **loose name equivalence** – aliased types are equivalent
  `type A = B` – is a declaration; `A` shares the definition of `B`

example – various type equivalences

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

- strict name equivalence
  - `p` and `q` – the same type
  - `r` and `u` – the same type
- loose name equivalence
  - `p` and `q` – the same type
  - `r`, `s`, and `u` – the same type
- structural equivalence
  - `p`, `q`, `r`, `s`, `t`, `u` – the same type

6.2.2 Type conversion and casts

- in many contexts, values of specific types are expected

  `a := expr` – `expr` should have the same type as `a`

  `a + b` – both should be integers or reals

  `foo(arg1, arg2,...,argN)` – the arguments should match the types of the formals

- cast – explicit type conversion

(i) - structurally equivalent types but the language uses name equivalence

  - no code executed at run time

(ii) - the types have different sets of values but the intersecting values are represented in the same way (e.g., subtype)

  - if provided type has values not in expected type – code at run time is needed

(iii) - different low-level representations but there is correspondence between values (e.g., integer and float)

  - run time code needed for conversion

example 1 - some type conversions in Ada

```
n :  integer;          --assume 32 bits
r :  real;             --assume IEEE double precision
type test_score is new integer range 0..100;
t :  test_score;
type celsius_temp is new integer;
c :  celsius_temp;
...
t := test_score (n);   --run-time semantic check required
n := integer (t);      --no check req.; every test_score is int
r := real (n);         --requires run-time conversion
n := integer (r);      --requires run-time conversion and check
n := integer (c);      --no run-time code required
c := celsius_temp (n)  --no run-time code required
```

example 2 - some type conversions in C

```
int n;
float r;
r = (float) n; /* generates code for run-time conversion */
n = (int) r; /* run-time conversion, no overflow check */
r = *((float *) &n); /* nonconverting type cast*/
```

example 3 - type conversions in C++

`static_cast` – type conversion
`reinterpret_cast` – nonconverting type cast
`dynamic_cast` – for assignments which cannot be guaranteed statically but can be checked at run time

6.2.3 Type compatibility and coercion

- most languages do not require type equivalence but type *compatibility*
  `a := expr` – `expr` should have a type compatible with the one of `a`
  `a + b` – both should have types compatible with integer or both with real
  `foo(arg1, arg2,...,argN)` – the arguments should be compatible with the types of the formals

- **coercion** – automatic, implicit type conversion
  - performed by the language implementation whenever the language allows a value of one type to be used in a context that expects another
  - weakening of type security – types are allowed to be mixed without explicit approval from programmer

example - some type coercions in C

```
short int s;
unsigned long int l;
char c; /* signed or unsigned -- implementation-dependent*/
float f; /* usually IEEE single-precision*/
double d; /* usually IEEE double-precision */
...
s = l /* l's low-order bits interpreted as a signed number*/
l = s; /* s is sign-extended to the longer length, then
     its bits are interpreted as an unsigned number*/
s = c; /* c is either sign- or zero-extended to s's length;*/
     the result is then interpreted as a signed number*/
f = l; /* l converted to floating-point; precision is lost*/
d = f; /* f converted to double; no precision lost*/
f = d; /* d converted to float; precision may be lost*/
```

6.2.4 Type inference

– defines the type of an expression based on the types of its parts

example $X\_type \odot X\_type$ has type X_type

- subranges

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

- what is the type of `a + b`?
- solution: the subrange's base type: `integer`

- what if `a + b` is assigned to `c` which has also a subrange type?
- run-time checks might be required
- the anonymous type `10..40` is computed
- if `10..40` is not contained in `c`'s subrange, then run-time check
- if the intersection is empty, then error message at compile time

- an unsolvable case

```
a :  integer range 0..20;
b :  integer range 10..20;
function foo (i :  integer) return integer is ...
...
a := b - foo(10);
```

- no way to predict the subrange for `b - foo(10)`
- may depend on values read at run time
- even when it doesn't, still impossible – computability theory
- run-time check required
- computing possible subranges for expressions – just a heuristic

- composite types

- in Ada

`"abc" & "defg"` has type `array (1..7) of character`

- in Pascal

```
var A : set of 1..10;
    B : set of 10..20;
    C : set of 1..15;
    i :  1..30;
...
C := A + B * [1..5, i];
```

- the type of `A + B * [1..5, i]` is `set of integer`
- because it is assigned to `C`, a dynamic semantic check might be required
- min-max heuristics

6.3 Records and variants

- records

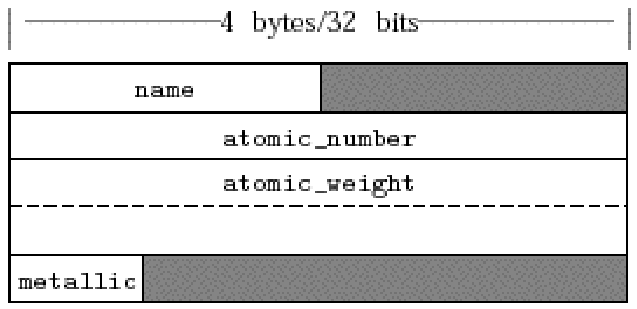
- in Pascal

```
type two_chars = packed array [1..2] of char;
type element = record
     name : two_cars;
     atomic_number : integer;
     atomic_weight : real;
     metallic : Boolean
end;
```

- in C

```
struct element {
     char name[2];
     int atomic_number;
     double atomic_weight;
     char metallic;
}
```

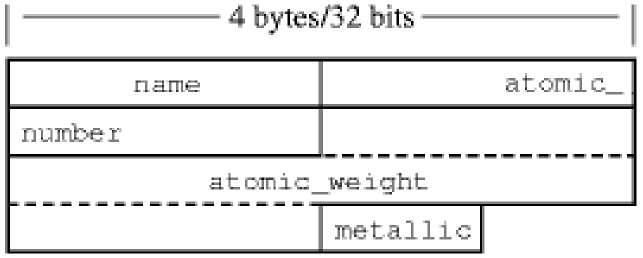
- memory layout



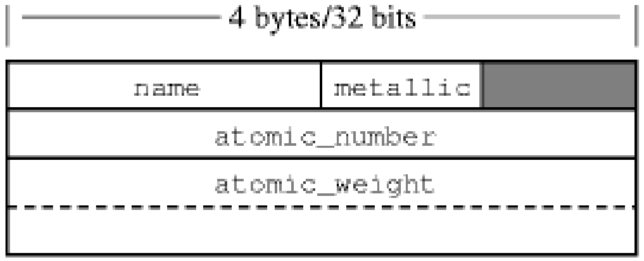
- *packed* records in Pascal – heavy access cost

```
type element = packed record
    name :  two_cars;      atomic_number :  integer;
    atomic_weight :  real;
    metallic :  Boolean
end;
```

- memory layout



- *rearranged* fields – implementation issue (no syntax)
- C and C++ guarantee no rearrangement

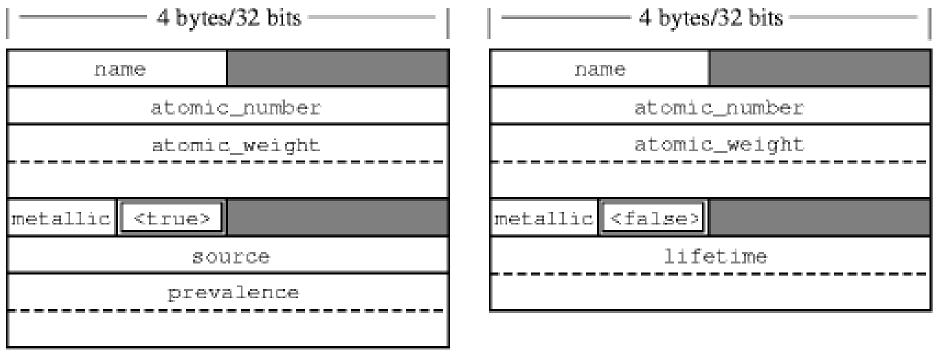


- variant records

- two or more alternative fields; only one of which is valid at any given time

example - in Pascal

```
type long_string = packed array [1..200] of char;
type string_ptr = l̂ong_string;
type element = record
     name :  two_cars;
     atomic_number :  integer;
     atomic_weight :  real;
     metallic :  Boolean
     case naturally_occurring :  Boolean of
          true :  (
               source :  string_ptr;
               prevalence :  real;
          );
          false :  (
               lifetime :  real;
          )
end;
```



- **safety** problems with variants

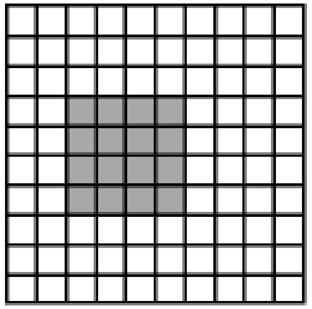
```
struct{
  int utype;
  union{
    int i;
    double d;
  } u;
} symb;
...
symb.u.i = 4;
cout << symb.u.i << endl;                                         // prints 4
cout << symb.u.d << endl;        // prints 1.97626e-323 - machine dependent
```

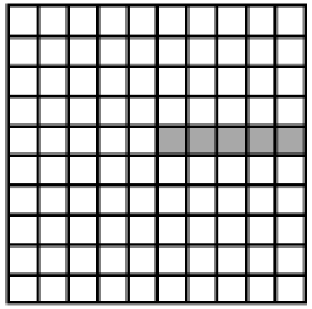
- the programmer must keep track of the current field
- some languages maintain a hidden variable for this – Algol 68, Ada
- some others forbid variants – Java

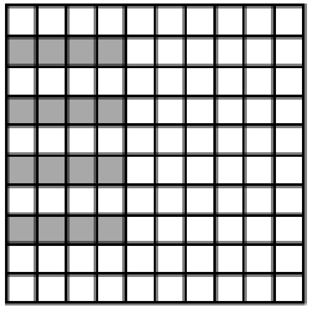
6.4 Arrays

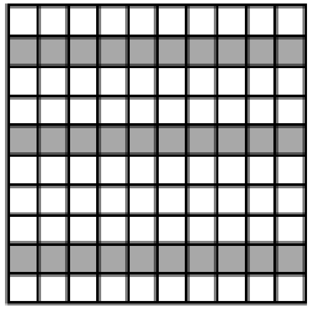
- the most common and important composite data type
- array = mapping from an index type to a component type
  - the index type can be integer only
  - usually the index type is discrete but can be non-discrete (Awk, Perl)

- **slices** – Fortran 90


matrix(3:6, 4:7)


matrix(6:, 5)


matrix(:4, 2:8:2)


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

6.4.1 Shape and allocation

- **shape** = the number of dimensions and bounds

▶ global lifetime, static shape – static global memory

▶ local lifetime, static shape – subroutine's stack frame

▶ local lifetime, elaboration time shape

  - space on stack – in the *variable-size* part
    (*fixed-size* part – all offsets are known at compile time)
  - a pointer to it is in the fixed-size part

▶ arbitrary lifetime, elaboration time shape

```
int[] A ...   A = new int[size]
```

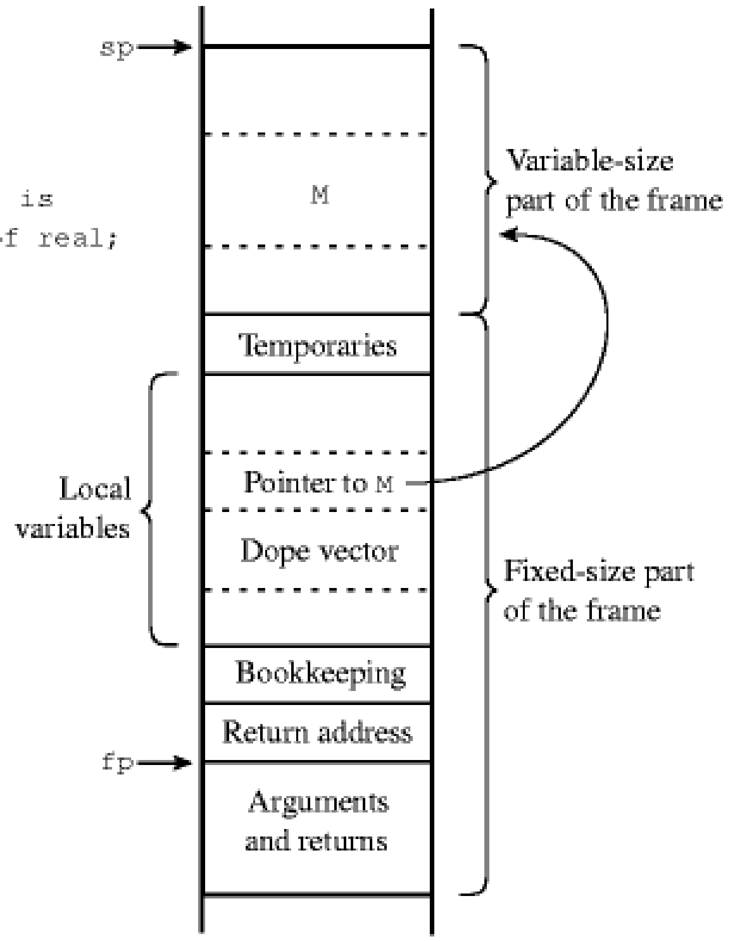
  - space on heap

▶ arbitrary lifetime, dynamic shape

  - array size can change
  - space on heap
  - increase in size – new allocation + copying

example

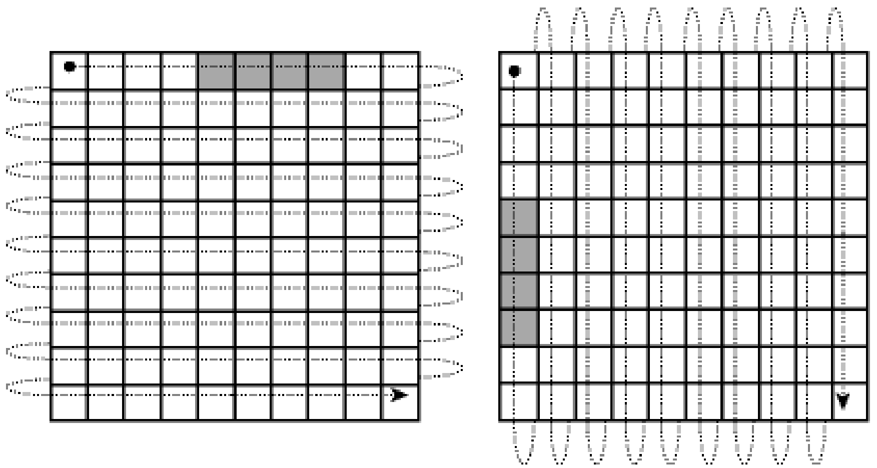
```
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
...
end foo;
```



- M - square two-dimensional array whose shape is determined by a parameter passed to foo at run time
- the space for M is in the variable-size part
- a pointer to M is in the fixed-size part
- *dope vector* (*run-time descriptor*) stores whatever information is known statically

6.4.2 Memory layout

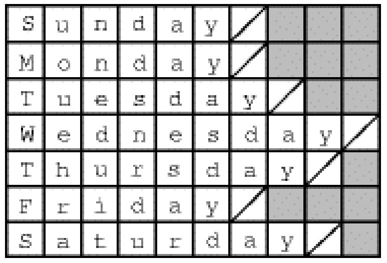
- one-dimensional arrays – contiguous in memory
- two-dimensional (or higher) – row- or column-major layout

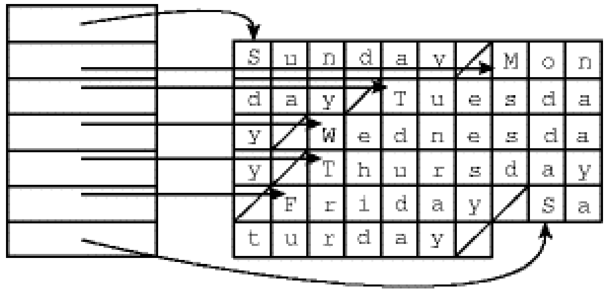


- contiguous allocation versus row pointers in C

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

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





- address calculation

- idea: compute as much as possible at compile time

example

`A:array [`L_1`..`U_1`] of array [`L_2`..`U_2`] of array [`L_3`..`U_3`] of elem_type;`

$S_3 =$ size of `elem_type`	–size of an element
$S_2 = (U_3 - L_3 + 1) \times S_3$	–size of a row
$S_1 = (U_2 - L_2 + 1) \times S_2$	–size of a plane

- the address of `A[i,j,k]` is

$$\begin{aligned} &\text{address of } \mathtt{A} + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3 \\ =\ &(i \times S_1) + (j \times S_2) + (k \times S_3) + \text{ address of } \mathtt{A} \qquad (*) \\ &-((L_1 \times S_1) + (L_2 \times S_2) + (L_3 \times S_3)) \qquad (**) \end{aligned}$$

(**) - computed at compile time

(*) - computed at run time

- if `A` is global then address of `A` is known at compile time
- if `A` is local to a subroutine then it decomposes to a static offset plus the run time frame pointer

6.5 Pointers

6.5.1 Dangling references

- **dangling reference** – a live pointer which no longer points to a live object

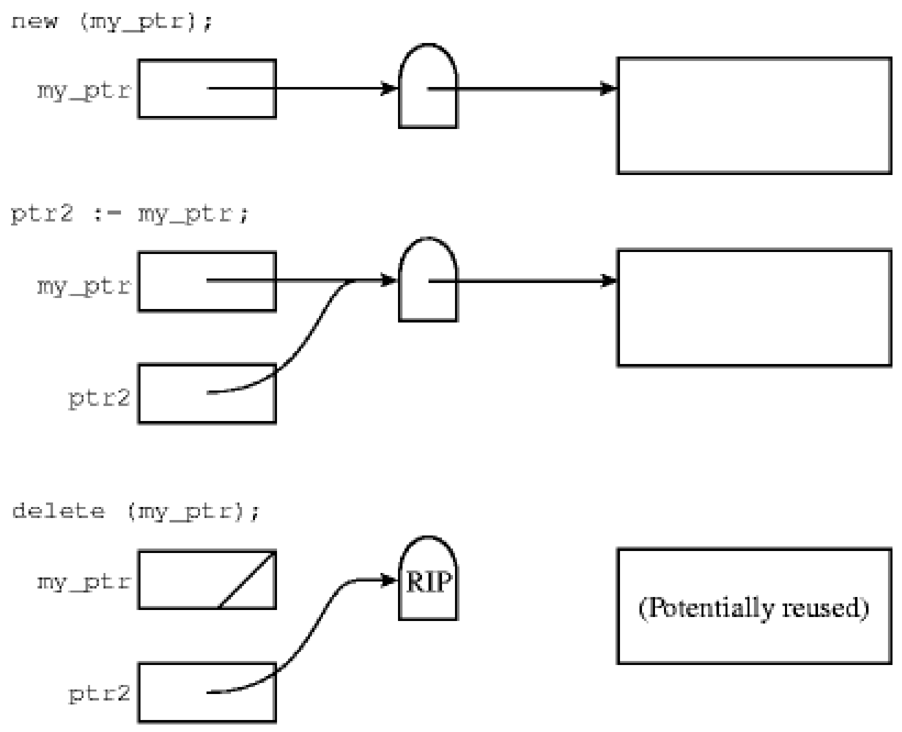
example

```
int* dangle()
{ int i = 23; return &i; }
main()
{ int* p; p = dangle(); }
```

- problem: using a dangling pointer, the reallocated space may be modified, producing unpredictable results
- solutions
  - tombstones
  - locks and keys

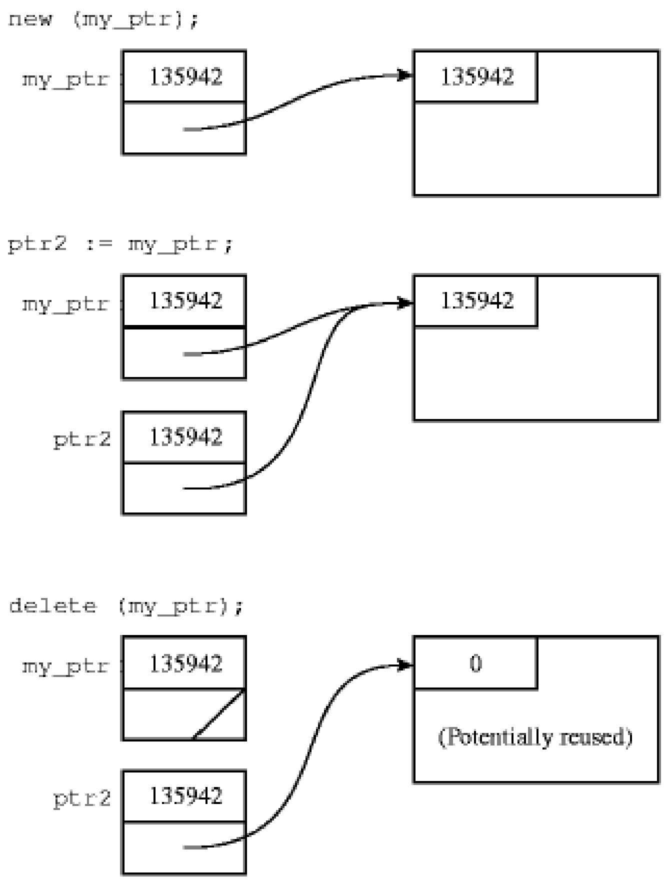
- tombstones

- a pointer contains the address of a tombstone which contains the address of an object
- when an object is reclaimed, the tombstone is modified to contain an invalid address (e.g., zero)



- locks and keys

- every object in the heap begins with a lock; every pointer consists of an address and a key; a pointer is valid only if the key matches the lock
- when an object is reclaimed, its lock is changed to an invalid value (e.g., zero)



6.5.2 Garbage collection

- memory leak – memory which the program does not need but failed to reclaim

- garbage collection – automatic storage reclamation

- eliminates any need to check for dangling references

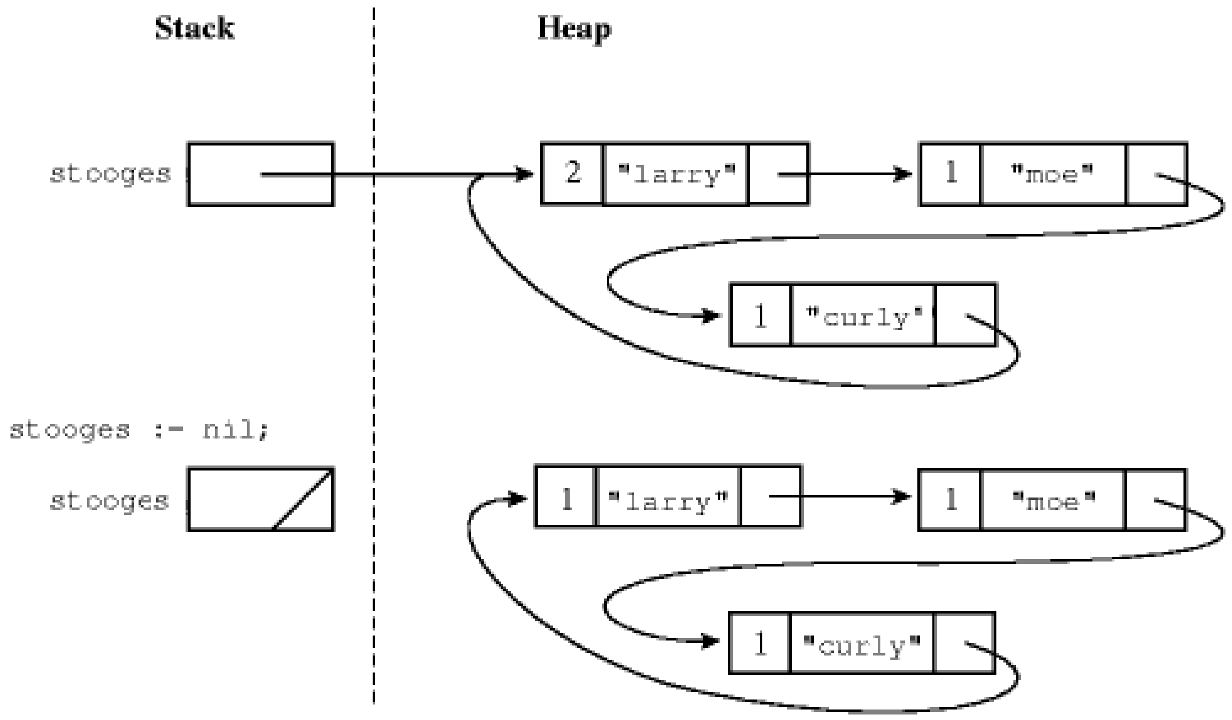
- manual storage reclamation
  - faster
  - difficult, source of memory leaks and dangling references

- methods for garbage collection
  - reference counts
  - mark and sweep

- reference counts

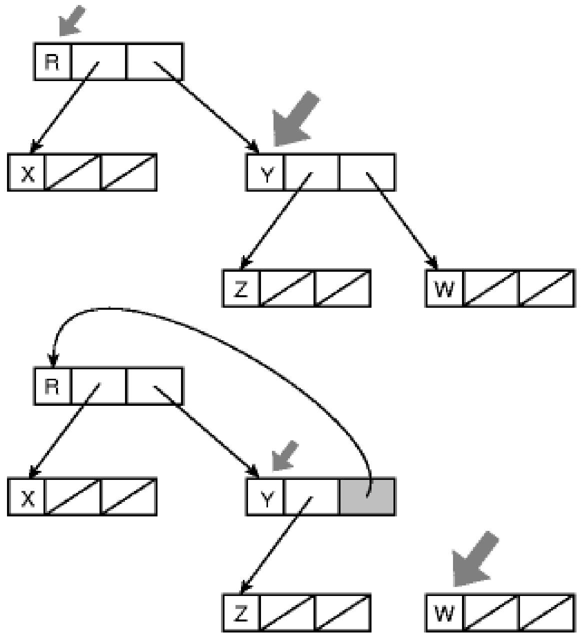
- an object is no longer useful when no pointers to it exist
- use *reference count* for each object
  - initially set to one, when object is created, (for the pointer returned by `new`)
  - when one pointer is assigned into another, increment the reference count of the object referred to by the right hand side and decrement the reference count of the object (formerly) referred to by the left-hand side
  - when returning from a subroutine, decrement the reference count of any object refereed to by a local pointer that is about to be destroyed
- when reference count is zero, the object can be reclaimed
  - recursively, when reclaiming an object, decrement reference counts for any objects refereed to by pointers within the object being reclaimed and reclaim those with zero reference counts

example - reference counts may fail to reclaim circular lists



- mark and sweep

- an object is no longer useful when it cannot be reached by following a chain of valid pointers starting from outside the heap (better definition)
- when the free space falls below some threshold, run the following steps
    - mark every block in the heap as useless
    - beginning with pointers outside the heap, explore recursively all linked data structures in the program and mark useful every reachable object
    - move all useless blocks to the free list
- *stop-and-copy*
- heap exploration via pointer reversal



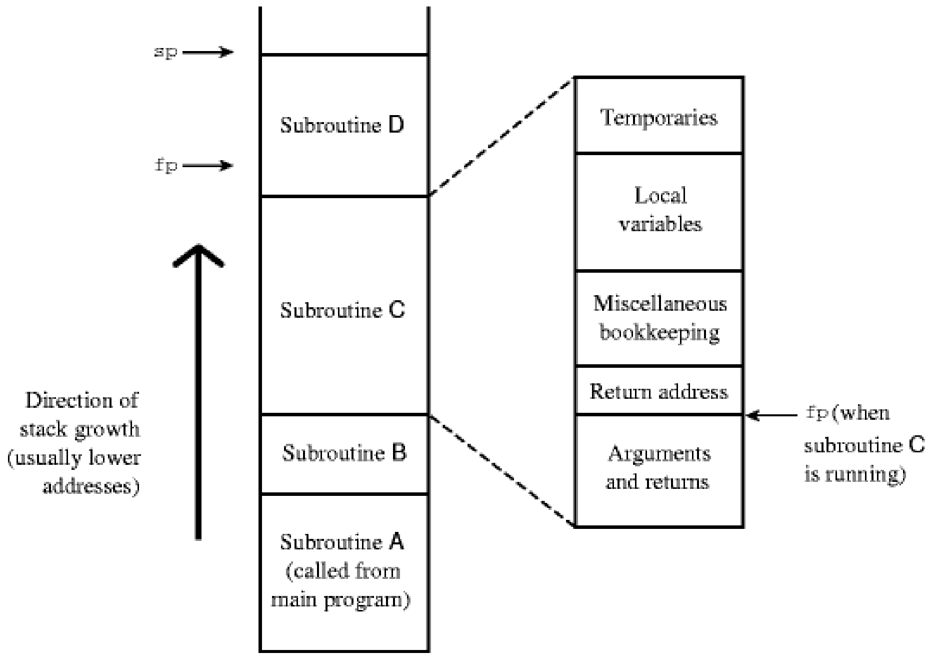
7 Subroutines

- calling sequence
  - static chains
  - displays
  - inline expansion
- parameter passing
  - call-by-value
  - call-by-reference
  - closures as parameters
  - special purpose parameters
- exception handling
  - exceptions and handlers
  - implementation

subroutines – principal mechanism for control abstraction

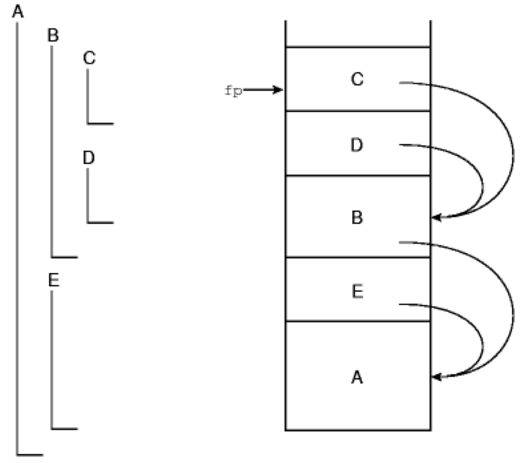
- **procedure** – subroutine which does not return a value
- **function** – subroutine which returns a value

- subroutines are usually parameterized
  - **actual parameters** – the arguments passed in a particular call
  - **formal parameters** – the names in the declaration
- **stack layout** -


sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine A (called from main program)
Direction of stack growth (usually lower addresses)
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
Arguments and returns
fp (when subroutine C is running)

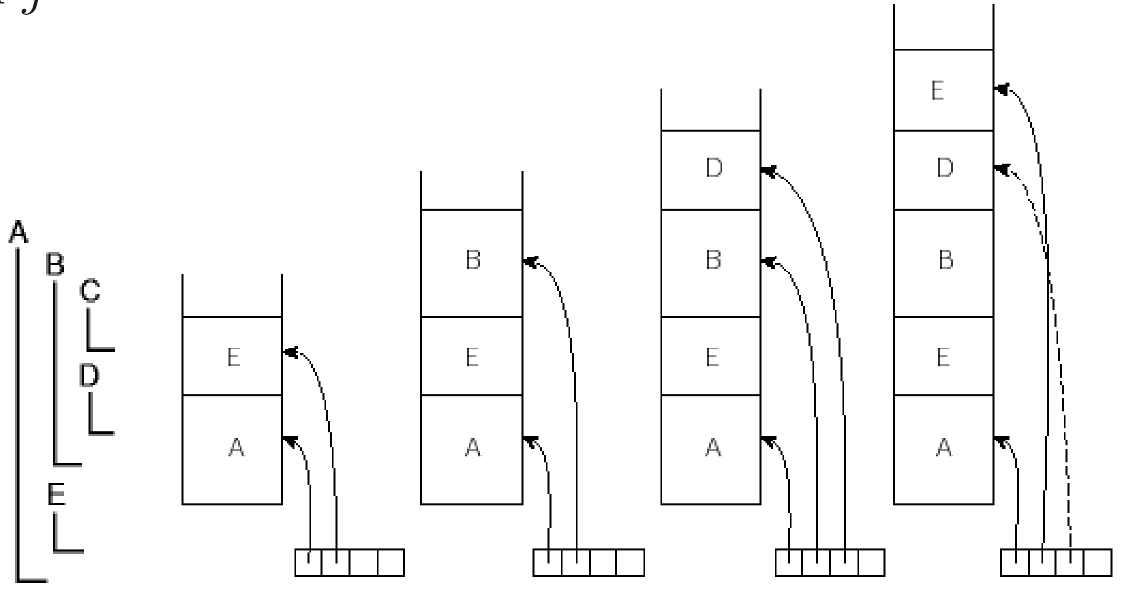
- static links

- disadvantage – access to an object in a scope k levels out requires that the static chain be dereferenced k times



- displays

- an embedding of the static chain into an array
- the jth element – a reference to the frame of the most recent active subroutine at lexical level j



- displays – no longer clear whether they are worth maintaining at all
  - static chains are seldom very long
  - variables in surrounding scopes tend not to be accessed very often
  - if they are accessed often then common subexpression optimizations can be used to ensure that a pointer to the appropriate frame remains in a register

- nested subroutines
  - some language designers argue that the development of object-oriented languages eliminated the need for nested subroutines
  - the success of C (which does not have nested subroutines) has already shown that nested subroutines are not crucial

7.1 Calling sequences

- maintenance of the subroutine call stack
  - *calling sequence* – code executed by the caller before and after the call
  - *prologue* – code executed at the beginning by the callee
  - *epilogue* – code executed at the end by the callee
- tasks on the way into a subroutine
  - pass parameters
  - save return address
  - change program counter
  - change stack pointer to allocate space
  - save registers and frame pointer
  - change frame pointer
- tasks on the way out of a subroutine
  - passing return values
  - restore stack frame to deallocate space
  - restore program counter

7.1.1 Static chains and displays

- maintaining the static chain

- the caller computes the callee's static link and pass it to the callee
(i) callee nested directly inside caller – the caller passes its own frame pointer
(ii) callee is $k \geq 0$ scopes outward – the caller dereferences its own static link k times and passes the result

- maintaining displays

- assume the callee is at lexical nesting level j
- the callee saves the current value of the jth display element into the stack and then replaces it with a copy of its own frame pointer
(i) callee nested directly inside caller – the caller and the callee share all display elements up to current
(ii) callee is $k \geq 0$ scopes outward – the caller and the callee share all display elements up to $j - 1$; the j's is different

- leaf subroutines – display can be left intact

7.1.2 In-line expansion

- alternative to stack-based calling
- subroutine is expanded in-line at the point of call
- avoids some overheads (space allocation, call and return, maintaining static chain or display, saving/restoring registers)
- semantically neutral

example in C++

```
inline int max (int a, int b) {return a > b ?  a : b}
```

- **normal order evaluation** – pass unevaluated arguments
  - **macros**

example in C

```
#define MAX(a, b) ((a) > (b) ?  (a) : (b))
```

- problem – can change semantics

example `MAX(x++, y++)` – increments twice the larger

- **applicative order evaluation** – evaluate before passing

7.2 Parameter passing

7.2.1 Call-by-value

- the values of the arguments are assigned into formals
- after that, the arguments and the formals are independent
- C (always), Pascal (default)

examples

```
x = 3;
void sqr (int x)
{ return (x * x); }
...  sqr(x)      // returns 9

void badswap (int x, int y)
{ int z;
   z = x; x = y; y = z;}
...  badswap(a, b);
// values of a and b - unchanged
```

```
procedure noget(c:  char);
  begin
    while eoln do readln;
    read(c)
  end;
...  noget(ch);  (* no effect on ch *)
```

7.2.2 Call-by-reference

- the address of the arguments are passed; each formal parameter is an alias of the corresponding argument
- changes made to one are visible to the other
- C++ (&), Pascal (`var`)

examples

```
void C_swap (int *px, int *py)                     /* C uses pointers */
{ int z; z = *px; *px = *py; *py = z; }
...  C_swap(&a, &b);                      /* the values are exchanged */

procedure Pascal_swap(var x:  integer; var y:  integer); (* Pascal uses *)
var z:  integer;                                              (* "var" *)
begin z := x; x := y; y := z; end;
...  Pascal_swap(a, b);                      // the values are exchanged

void Cpp_swap(int& x, int& y)                // C++ has call-by-reference
{ int z; z = x; x = y; y = z; }
...  Cpp_swap(a, b);                         // the values are exchanged
```

- call-by-value - requires copying the arguments
  - might be slow for large arguments
- call-by-reference - only an address is passed
  - extra level of indirection – bad if used a lot
  - good when the arguments need to be modified
- pass-by-reference for large arguments which are not modified
  - Modula (`READONLY`), C, C++ (`const`)

example `void append_to_log (const huge_record *r) { ... }`

```
append_to_log(&my_record);
```

example – reference use as function returns

– overloaded `<<` and `>>` return a reference to the first argument

```
cout << a << b << c;
         //short for ((cout.operator<<(a)).operator<<(b)).operator<<(c);
```

- without references – if `<<`, `>>` return pointers:

```
((cout.operator<<(a))->operator<<(b))->operator<<(c);
*(*(cout.operator<<(a)).operator<<(b)).operator<<(c);
*(*(cout << a) << b) << c;
```

7.2.3 Closures as parameters

example - a closure may be passed as a parameter when the parameter is a subroutine

```
procedure apply_to_A (function f(n : integer) : integer;
hspace*1pt                    var A : array[low..high : integer] of integer);
var i : integer;
begin
  for i := low to high do A[i] := f(A[i]);
end;
```

7.2.4 Special purpose parameters

▶ **conformant (open) arrays** – arrays whose shape is finalized at run time

example - in C

```
int sum(int a[], int n) {
  int i, s = 0;
  for (i = 0; i < n; ++i)
    s += a[i];
  return s; }
```

or

```
int sum(int a[][5]) { ...  }
```

▶ **default parameters** – parameters that need not be provided by the caller

example - in C++

```
int sqr_or_power (int n, int k = 2) {   // k=2 is the default if (k == 2)
  return (n * n)
else
  return(sqr_or_power(n, k - 1) * n); }
...  sqr_or_power(i + 5)  // computes a square
...  sqr_or_power(i + 5, 3)  // computes a cube
```

▸ **named parameters** – non-positional parameters

example - in Ada

```
procedure put (item : in integer;
         width : in field := default_width;
         base : in number_base := default_base);
...  put (item => 37, base => 8)
...  put (base => 8, item => 37)     - arbitrary order
...  put (37, base => 8)     - omitting any intermediate default parameters
```

example – in Ada

```
format_page (columns => 2,
       window_height => 400, window_width => 200, header_font => Helvetica,
       body_font => Times, title_font => Times_Bold, header_point_size => 10,
       body_point_size => 11, title_point_size => 13, justification => true,
       hyphenation => false, page_num => 3, paragraph_indent => 18,
       background_color => white);
```

▸ **variable number of arguments**

example - in C++ `int printf(char *format, ...)`

7.3 Exception handling

7.3.1 Exceptions and handlers

- exception – an unexpected or unusual condition that arises during execution
- most are various run-time errors
- examples - arithmetics overflow, division by zero, end-of-file on input, subscript and subrange errors, null pointer dereference, return from a subroutine that has not yet designated a return value, etc.
    - automatically detected – by the implementation
    - raised explicitly by the program

- handler – a place to branch to when an exception appears

example – in PL/I (pioneered exception handling)

`ON` *condition* *statement*
- the `ON` statement remembered when encountered
- executed later if *condition* (e.g., `OVERFLOW`) arises
    - if fatal, then terminate, if recoverable then may continue
    - *statement* – `GOTO` or `BEGIN...END` block
- binding of handlers to exceptions – at run time (confusing, error prone)

- lexically bound handlers

- more recent languages (Clu, Ada, Modula-3, C++, Java, ML)
- handlers are lexically bound to blocks of code
- execution of the handler replaces the yet-to-be-completed portion of the block

- general rules for handling exceptions

- *unwinding the subroutine call stack*
- if exception not handled in the current subroutine – *exceptional return*
  - the subroutine returns abruptly
  - the exception is raised at the point of call
- if not handled in the calling routine
  - propagated back up the dynamic chain
- if not handled in the program's main routine
  - a predefined outermost handler is invoked – the program terminates

- purposes for exception handlers

- possible recovery and execution continuation
- if recovery not possible, print helpful error message
- if exception cannot be handled locally, clean up resources and reraise the exception (hope to recover)

example – in C++

```
try
{
  ...  // protected block of code
  ...  // throw obj;
}
catch (end_of_file)
{
  ...  // handler for unexpected end_of_file
}
catch (io_error e)
{
  ...  // handler for any io_error other than end_of_file
}
catch (...)
{
  ...  // catch-all not named previously
}
```

example – in C++

```
vect::vect(int n) : size(n)
{ if (n < 1)
    throw(n);
  p = new int[n];
  if (p == 0)
    throw("FREE STORE EXHAUSTED");}
void g(int m)
{
  try
  { vect a(m);
    ...  }
  catch(int n)
  { cerr << "SIZE ERROR " << n << endl;
    g(10);}      // retry g with legal size
  catch(const char* error)
  { cerr << error << endl;
    abort();}
}
```

example – in C++

```
void foo()
{
  int i, j;
  ...
  throw i;
  ...  }
void call_foo()
{
  ...
  foo();
  ...  }
int main()
{
  try
  {
    call_foo(); // foo exists with i and j destroyed
  }
catch(int n) { ...  }
}
```

example – alternative approach to phrase-level recovery (p. 72)

- syntax_error – exception
- add handlers around bodies of expression, declaration, etc.

```
procedure statement
   try ... – code to parse a statement
   except when syntax_error =>
      loop
         if next_token $\in$ FIRST(statement)
            statement – try again
            return
         elsif next_token $\in$ FOLLOW(statement)
            return
         else get_next_token
```

7.3.2 Implementation

▸ **with a stack of handlers**

- control enters a protected block ⇒ handler for block pushed onto the stack
  - handlers for different exceptions – single handler with a multi-arm `if`
- exception ⇒ language run-time system pops and calls the innermost handler
- it first checks to see if it matches the exception; if not, reraise
- problem – run-time overhead – extra code for protected blocks and subroutines

▸ **with a compile-time generated table**

- each entry has two fields
  - starting address of a protected block
  - address of the corresponding handler
- the table is sorted on the first field
- when exception occurs, the language run-time system performs a binary search in the table, using the program counter as the key, to find the handler
- if rerisen, repeat
- for implicit handlers associated with propagation out of subroutines – the reraise code should use the return address of the subroutine, rather than the current program counter, as the key for table lookup

- simulation of exceptions in language that do not support them

- Pascal – `goto`s to labels outside subroutines
- Algol60 – labels can be passed as parameters
- PL/I – allows labels to be stored in variables

- the above mechanisms permit the program to escape from a deeply nested context, but in a very unstructured way
- Scheme – very general approach
- `call-with-current-continuation` or `call/cc`
    - takes a single argument f, a function
    - it calls f, passing as argument a continuation c (closure) that captures the current program counter and referencing environment
    - at any point in the future, f can call c to re-establish the saved environment

8 Functional programming

- basic ideas
  - no side effects
  - implicit storage management
  - functions are first-class values
- advantages / disadvantages
- theoretical foundations – λ-calculus
- Scheme

8.1 Computability models

- 1930's – efforts to formalize the notion of algorithm
- Turing, Church, Kleene, Post, etc.
- all approaches proved to be equally powerful
- Church's thesis – any intuitive model of computing is equally powerful

- *Turing machine*
  - an automaton with an unbounded tape
  - computes in an imperative way, changing the values in cells of its tape
  - *imperative programming*

- *Church's λ-calculus*
  - parametrized expressions
  - computes by substituting parameters into expressions
  - *functional programming*

8.2 Basic ideas

- functional programming – defines the output of a program as a mathematical function of the inputs, with no internal state and thus no side effects

- no side effects

- side effect = the modification of the value of a variable during expression evaluation (example: `x + f(x)`)
- effects in pure functional programming are achieved by applying function, recursively or through composition

- first-class function values – can be

- passed as a parameter
- returned from a subroutine
- assigned in a variable
- (more strictly) computed at run time

- high-order functions

- take functions as arguments or return functions as results

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
(define cadr (compose car cdr))
(cdr '(a b c d))          ;(b c d)
(car (cdr '(a b c d)))          ;b
(cadr '(a b c d))          ;b
```

- extensive polymorphism

- Lisp and its dialects are **dynamically typed**

example

```
(define list_count
  (lambda (x)
    (if (null? x) 0 (+ 1 (list_count (cdr x))))))
(list_count '(12 a + ? >)) => 5
```

- lists

- natural recursive definition
- list = first element + tail

- recursion

- the only means of doing something repeatedly in the absence of side effects

- garbage collection

- storage automatically reclaimed

- homogeneity of programs and data

- a program is itself a list

8.3 Functional vs imperative programming

- advantages:

- no side effects ⇒ predictable behaviour

  - *equational reasoning* – two expressions equivalent at any point in time are always equivalent

- simpler syntax (simpler programs)
- simpler semantics
- higher level of programming

- disadvantages:

- time consuming – every result is a new object instead of a modification of an existing one

  - *trivial update problem* – small modification of large data structure

8.4 Theoretical foundations – λ-calculus

- Church (1941) – to study computations with functions

λ-expressions or **terms** – only three kinds

- **variables:**
  - denoted by $x, y, z, u, v, \ldots$
- **applications (function applications):**
  example $(M\ N)$ – application of M to N
- **abstractions (function creation):**
  example $(\lambda x.M)$ - function with parameter x and body M

examples

$(\lambda x.x * x)$ - a function that maps x to $x * x$
$(\lambda x.x * x)4$ - the same function applied to 4

- a grammar for terms: $M \ ::= \ x \mid (M_1\ M_2) \mid (\lambda x.M)$

8.4.1 Syntactic conventions

1. application has higher precedence than abstraction:

   example $\lambda x.xy$ means $\lambda x.(xy)$

2. application is left-associative

   example xyz means $(xy)z$

3. 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 $\lambda xyz.x\ z(y\ z) = (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$

8.4.2 Lambda calculus with constants

- functional programming language = lambda calculus with appropriate constants
- c below corresponds to built-in constants and operators

$$M \ ::= \ c \mid x \mid (M_1\ M_2) \mid (\lambda x.M)$$

example

$(\lambda x.x * x)\ 4$ gives the value 16

```
(define square
  (lambda (x)
    (* x x)))
...
(square 4) => 16
```

examples

$\lambda x.x$ – identity
$\lambda x.(x+1)$
$\lambda x.(x+y^2)$
$\lambda x.\lambda y.(x+y)$
$(\lambda x.(x+1))4$ – gives 5

- the general form of a function: $\lambda x.M$
- x is the formal parameter and M is the body

8.4.3 Free and bound variables

- $\lambda x.M$ - is a *biding* of the variable 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

$$\textbf{\textit{free}}(x) = \{x\}$$
$$\textbf{\textit{free}}(MN) = \textbf{\textit{free}}(M) \cup \textbf{\textit{free}}(N)$$
$$\textbf{\textit{free}}(\lambda x.M) = \textbf{\textit{free}}(M) - \{x\}$$

- the set of variables which are not free is ***bound***(M)

- any occurrence of a variable is free or bound; not both

example

$$\begin{tikzcd}[column sep=small]
\lambda & y. \arrow[rrrrrr, bend left] & \lambda & z. \arrow[rr, bend right] \arrow[rrrrr, bend right] & x & z & (& y & z &)
\end{tikzcd}$$

8.4.4 Substitution

- $\{N/x\}M$ – *substitution* of a term N for a variable x in M

- **substitution rules:**

  - **informally**:
    (i) if $\boldsymbol{free}(N) \cap \boldsymbol{bound}(M) = \emptyset$,
    then just replace all free occurrences of x in M;
    (ii) otherwise, rename with fresh variables until (i) applies

- **formally**:

1. - in variables: the same or different variable

   (a) $\{N/x\}x = N$

   (b) $\{N/x\}y = y, \quad y \neq x$

2. - in applications – the substitution distributes
   $\{N/x\}(P\ Q) = \{N/x\}P\ \{N/x\}Q$

3. - in abstractions – several cases

   (a) - no free x
   $\{N/x\}(\lambda x.P) = \lambda x.P$

   (b) - no interaction – y is not free in N
   $\{N/x\}(\lambda y.P) = \lambda y.\{N/x\}P \qquad y \neq x, y \notin \boldsymbol{free}(N)$

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

8.4.5 Equality of pure λ-terms

1. α-***axiom*** – renaming the formal parameters
   $\lambda x.M =_\alpha \lambda y.\{y/x\}M, \quad y \notin \textit{\textbf{free}}(M)$
2. β-***axiom*** – applying an abstraction to an argument
   (a) $(\lambda x.M)N =_\beta \{N/x\}M$
   (b) $=_\beta$ is a congruence
      i. idempotent: $M =_\beta M$
      ii. commutative: $\dfrac{M =_\beta N}{N =_\beta M}$
      iii. transitive: $\dfrac{M =_\beta N \quad N =_\beta P}{M =_\beta P}$
      iv. congruence w.r.t. application: $\dfrac{M_1 =_\beta M_2 \quad N_1 =_\beta N_2}{M_1\ N_1 =_\beta M_2\ N_2}$
      v. congruence w.r.t. abstraction: $\dfrac{M =_\beta N}{\lambda x.M =_\beta \lambda x.N}$

β-**equality** is any sequence of $=_\beta$ and $=_\alpha$

examples

$$
\begin{aligned}
&(\lambda xyz.x\ z\ (y\ z))\ (\underline{\lambda x.x})\ (\lambda x.x)\\
&\qquad =_\alpha\ (\lambda xyz.x\ z\ (y\ z))\ (\lambda u.u)\ (\underline{\lambda x.x})\\
&\qquad =_\alpha\ (\underline{\lambda x}yz.x\ z\ (y\ z))\ (\underline{\lambda u.u})\ (\lambda v.v)\\
&\qquad =_\beta\ (\lambda yz.(\underline{\lambda u.u)\ z}\ (y\ z))\ (\lambda v.v)\\
&\qquad =_\beta\ (\underline{\lambda y}z.z\ (y\ z))\ (\underline{\lambda v.v})\\
&\qquad =_\beta\ \lambda z.z\ ((\underline{\lambda v.v)\ z})\\
&\qquad =_\beta\ \lambda z.z\ z
\end{aligned}
$$

$$
\begin{aligned}
&(\underline{\lambda f}gh.f\ g\ (h\ h))\ \underline{(\lambda xy.x)}\ h\ (\lambda x.x\ x)\\
&\qquad =_\beta\ (\lambda g\underline{h}.(\lambda xy.x)\ g\ \underline{(h\ h))}\ h\ (\lambda x.x\ x)\\
&\qquad =_\alpha\ \underline{(\lambda g}k.(\lambda xy.x)\ g\ (k\ k))\ \underline{h}\ (\lambda x.x\ x)\\
&\qquad =_\beta\ \underline{(\lambda k}.(\lambda xy.x)\ h\ (k\ k))\ \underline{(\lambda x.x\ x)}\\
&\qquad =_\beta\ \underline{(\lambda x}y.x)\ \underline{h}\ ((\lambda x.x\ x)\ (\lambda x.x\ x))\\
&\qquad =_\beta\ \underline{(\lambda y}.h)\ \underline{((\lambda x.x\ x)\ (\lambda x.x\ x))}\\
&\qquad =_\beta\ h
\end{aligned}
$$

8.4.6 Computing with pure λ-terms

- idea: we want to reduce the terms into as simple a form as possible
- in $(\lambda x.M)\ N =_\beta \{N/x\}M$, the right hand side is simpler

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

- rewriting rules:

α-conversion – renaming the formal parameters

$$\lambda x.M \Rightarrow_\alpha \lambda y.\{y/x\}M, \quad y \notin \textbf{\textit{free}}(M)$$

β-reduction – applying an abstraction to an argument

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

- reduction – any sequence of $\Rightarrow_\beta$ and $\Rightarrow_\alpha$
- a term is in (β)**normal form** if it cannot be β-reduced

example $\lambda x.x\ x$ – cannot be reduced – normal form

- there may be **several ways** to reduce to a normal form:

example - any path below is such a reduction

$$\begin{tikzcd}
 & & (\underline{\lambda x}yz.x\ z\ (y\ z))\ \underline{(\lambda x.x)}\ (\lambda x.x) \arrow[d] & & \\
 & & (\underline{\lambda y}z.\underline{(\lambda x.x)\ z}\ (y\ z))\ \underline{(\lambda x.x)} \arrow[dl] \arrow[dr] & & \\
 & (\underline{\lambda y}z.z\ (y\ z))\ \underline{(\lambda x.x)} \arrow[dr] & & \lambda z.\underline{(\lambda x.x)\ z}\ (\underline{(\lambda x.x)\ z}) \arrow[dl] \arrow[dr] & \\
 & & \lambda z.z\ (\underline{(\lambda x.x)\ z}) \arrow[dr] & & \lambda z.\underline{(\lambda x.x)\ z}\ z \arrow[dl] \\
 & & & \lambda z.z\ z &
\end{tikzcd}$$

- **nonterminating** reductions – which never reach a normal form

example

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

Church-Rosser Theorem: For all pure λ-terms M, P, Q, if $M \Rightarrow_\beta^* P$ and $M \Rightarrow_\beta^* Q$, then there exists a term R such that $P \Rightarrow_\beta^* R$ and $Q \Rightarrow_\beta^* R$.
In particular, the normal form, when exists, is unique.

$$\begin{tikzcd}
 & M \arrow[dl, "\Rightarrow_\beta^*"'] \arrow[dr, "\Rightarrow_\beta^*"] & \\
P \arrow[dr, "\Rightarrow_\beta^*"'] & & Q \arrow[dl, "\Rightarrow_\beta^*"] \\
 & R &
\end{tikzcd}$$

8.4.7 Reduction strategies

- call-by-value reduction (applicative order)

- parameters are evaluated first, then passed
- choose the leftmost-innermost $(\lambda x.M)\ N$
- might not reach a normal form even if there is one

<u>example</u>

$$
\begin{aligned}
(\lambda y.h)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)})& \\
&\Rightarrow_\beta\ (\lambda y.h)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)}) \\
&\Rightarrow_\beta\ (\lambda y.h)\ (\underline{(\lambda x.x\ x)\ (\lambda x.x\ x)}) \\
&\Rightarrow_\beta\ \ldots
\end{aligned}
$$

- call-by-name reduction (normal order)
- parameters are passed unevaluated
- choose the leftmost-outermost $(\lambda x.M)\ N$ (leftmost λ)

- **Theorem:** Normal order reduction reaches a normal form if there is one.

example

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

- 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

8.4.8 λ-calculus can model everything

- boolean values

- true (T) is $\lambda x.\lambda y.x$
  - interpretation: of a pair of values, choose the first
- false (F) is $\lambda x.\lambda y.y$
  - interpretation: of a pair of values, choose the second

- properties

$$((\mathsf{T}\ P)\ Q) \Rightarrow_\beta (((\lambda x.\lambda y.x)\ P)\ Q) \Rightarrow_\beta ((\lambda y.P)\ Q) \Rightarrow_\beta P$$
$$((\mathsf{F}\ P)\ Q) \Rightarrow_\beta (((\lambda x.\lambda y.y)\ P)\ Q) \Rightarrow_\beta ((\lambda y.y)\ Q) \Rightarrow_\beta Q$$

- boolean functions

- not $= \lambda x.((x\ \mathsf{F})\ \mathsf{T})$
- and $= \lambda x.\lambda y.((x\ y)\ \mathsf{F})$
- or $= \lambda x.\lambda y.((x\ \mathsf{T})\ y)$

- the interpretation is consistent with predicate logic
- for instance:

$$\text{not } \mathsf{T} \Rightarrow_\beta (\lambda x.((x\ \mathsf{F})\ \mathsf{T}))\ \mathsf{T} \Rightarrow_\beta ((\mathsf{T}\ \mathsf{F})\ \mathsf{T}) \Rightarrow_\beta \mathsf{F}$$
$$\text{not } \mathsf{F} \Rightarrow_\beta (\lambda x.((x\ \mathsf{F})\ \mathsf{T}))\ \mathsf{F} \Rightarrow_\beta ((\mathsf{F}\ \mathsf{F})\ \mathsf{T}) \Rightarrow_\beta \mathsf{T}$$

- integers

$$
\begin{aligned}
\mathbf{0} &= \lambda f.\lambda c.c \\
\mathbf{1} &= \lambda f.\lambda c.(f\ c) \\
\mathbf{2} &= \lambda f.\lambda c.(f\ (f\ c)) \\
\mathbf{3} &= \lambda f.\lambda c.(f\ (f\ (f\ c)))
\end{aligned}
$$

$\ldots\ldots\ldots$

$$\mathbf{N} = \lambda f.\lambda c.\underbrace{(f\ (f \ldots (f}_{N}\ c)) \ldots)$$

- c is the zero element and f is the successor function

- we have

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

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

- integer operations

- addition: $+ = \lambda M.\lambda N.\lambda a.\lambda b.((M\ a)((N\ a)\ b))$

$[M+N] = \lambda a.\lambda b.((M\ a)\ ((N\ a)\ b))$

$$\Rightarrow_\beta \lambda a.\lambda b.\underbrace{(a\ (a \ldots (a}_{M+N}\ b)) \ldots)$$

- multiplication: $\times = \lambda M.\lambda N.\lambda a.(M\ (N\ a))$

$[M \times N] = \lambda a.(M\ (N\ a)) \Rightarrow_\beta \lambda a.\lambda b.\underbrace{(a\ (a \ldots (a}_{M\times N}\ b)) \ldots)$

- exponentiation: $\wedge = \lambda M.\lambda N.(N\ M)$

$[M^N] = (N\ M) \Rightarrow_\beta \underbrace{(a\ (a \ldots (a}_{M^N}\ b)) \ldots)$

- in this way, we can develop all of the computable mathematical functions

- control flow

- if $= \lambda c.\lambda t.\lambda e.c\ t\ e$

- $c =$ condition; $t =$ then, $e =$ else

if T $3\ 4$
$= (\lambda c.\lambda t.\lambda e.c\ t\ e)(\lambda x.\lambda y.x)\ 3\ 4$
$\Rightarrow_\beta^* (\lambda x.\lambda y.x)\ 3\ 4$
$\Rightarrow_\beta^* 3$

if F $3\ 4$
$= (\lambda c.\lambda t.\lambda e.c\ t\ e)(\lambda x.\lambda y.x)\ 3\ 4$
$\Rightarrow_\beta^* (\lambda x.\lambda y.y)\ 3\ 4$
$\Rightarrow_\beta^* 4$

- recursion

$\mathsf{gcd} = \lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a(\mathsf{if}\ (\mathsf{greater}\ a\ b)\ (\mathsf{gcd}\ (\mathsf{minus}\ a\ b)\ b)\ (\mathsf{gcd}\ (\mathsf{minus}\ b\ a)\ a)))$

- not definition because **gcd** is in both sides
- rewrite using $\beta - abstraction$

$\mathsf{gcd} = (\lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a(\mathsf{if}\ (\mathsf{greater}\ a\ b)\ (g\ (\mathsf{minus}\ a\ b)\ b)\ (g\ (\mathsf{minus}\ b\ a)\ a))))\ \mathsf{gcd}$

- we obtain the equation $\mathsf{gcd} = f\ \mathsf{gcd}$, for f:

$f = \lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a(\mathsf{if}\ (\mathsf{greater}\ a\ b)\ (g\ (\mathsf{minus}\ a\ b)\ b)\ (g\ (\mathsf{minus}\ b\ a)\ a)))$

- **gcd** is a fixed point of f
- for any f, the *fixed point combinator* below is a fixed point of f

$\mathsf{Y} = \lambda h.(\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x))$

- if the normal order evaluation of $\mathsf{Y}f$ terminates then $f(\mathsf{Y}f)$ and $\mathsf{Y}f$ will reduce to the same normal form
- we get then a good definition for **gcd**:

$$\begin{aligned}\mathsf{gcd} = {} & (\lambda h.\ (\lambda x.h\ (x\ x))\ (\lambda x.h\ (x\ x))) \\ & \quad (\lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a(\mathsf{if}\ (\mathsf{greater}\ a\ b)\ (g\ (\mathsf{minus}\ a\ b)\ b)\ (g\ (\mathsf{minus}\ b\ a)\ a))))\end{aligned}$$

example

$$
\begin{aligned}
\mathsf{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.(\mathsf{if}\ (=a\,b)\ a\,(\mathsf{if}\ (>a\,b)\ (g(-\,a\,b)\ b)\ (g(-\,b\,a)\ a))))\ (k\,k))\ 2\ 4 \\
&\to_\beta (\lambda a.\lambda b.(\mathsf{if}\ (=a\,b)\ a(\mathsf{if}\ (>a\,b)\ ((k\,k)(-\,a\,b)\ b)\ ((k\,k)(-\,b\,a)\ a))))\ 2\ 4 \\
&\to_\beta^* \mathsf{if}\ (=2\,4)\ 2\,(\mathsf{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\,(\mathsf{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)(-\,4\,2)\ 2)) \\
&\to_\beta^* (=2\,4)\,2\ (\mathsf{if}\ (>2\,4)((k\,k)\,(-\,2\,4)\,4)((k\,k)\,(-\,4\,2)\ 2)) \\
&\to_\delta F\,2\ (\mathsf{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\equiv (\lambda x.\lambda y.y)\ 2\ (\mathsf{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\to_\beta^* \mathsf{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.(\mathsf{if}\ (=a\,b)\ a\,(\mathsf{if}\ (>a\,b)\ (g(-\,a\,b)\ b)\ (g(-\,b\,a)\ a))))\ (k\,k))\ (-\,4\,2)\,2 \\
&\to_\beta (\lambda a.\lambda b.(\mathsf{if}\ (=a\,b)\ a(\mathsf{if}\ (>a\,b)\ ((k\,k)(-\,a\,b)\ b)\ ((k\,k)(-\,b\,a)\ a))))\ (-\,4\,2)\,2 \\
&\to_\beta^* \mathsf{if}\ (=(-\,4\,2)\ 2)\ (-\,4\,2)\ (\mathsf{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) \\
&\qquad (=(-\,4\,2)\ 2)\,(-\,4\,2)\ (\mathsf{if}\ (>(-\,4\,2)\ 2)\ ((k\,k)\ (-\,(-\,4\,2)\ 2)\ 2)((k\,k)\ (-\,2\,(-\,4\,2))\ (-\,4\,2))) \\
&\to_\beta^* (=(-\,4\,2)\ 2)\,(-\,4\,2)\ (\mathsf{if}\ (>(-\,4\,2)\ 2)\ ((k\,k)\ (-\,(-\,4\,2)\ 2)\ 2)((k\,k)\ (-\,2\,(-\,4\,2))\ (-\,4\,2))) \\
&\to_\delta (=2\,2)\,(-\,4\,2)\ (\mathsf{if}\ (>(-\,4\,2)\ 2)((k\,k)\,(-\,(-\,4\,2)\ 2)\ 2)((k\,k)\,(-\,2\,(-\,4\,2))\ (-\,4\,2))) \\
&\to_\delta T\ (-\,4\,2)\ (\mathsf{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)\ (\mathsf{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}
$$

- structures

- denote

select_first $= \lambda x.\lambda y.x$ (true)
select_second $= \lambda x.\lambda y.y$ (false)

- define the following list-processing functions

cons $= \lambda a.\lambda d.\lambda x.x\ a\ d$
car $= \lambda l.l$ select_first
cdr $= \lambda l.l$ select_second
null? $= \lambda l.l\ (\lambda x.\lambda y.\mathsf{F})$

- we have then

$$\begin{aligned}
\text{car (cons } A\ B) &= (\lambda l.l \text{ select\_first}) (\text{cons } A\ B)\\
&\Rightarrow_\beta (\text{cons } A\ B) \text{ select\_first}\\
&= ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B) \text{ select\_first}\\
&\Rightarrow_\beta^* (\lambda x.x\ A\ B) \text{ select\_first}\\
&\Rightarrow_\beta \text{select\_first } A\ B\\
&= (\lambda x.\lambda y.x)\ A\ B\\
&\Rightarrow_\beta^* A
\end{aligned}$$

- similarly:

cdr (cons $A\ B) \Rightarrow_\beta^* B$

(null? (cons $A\ B) \Rightarrow_\beta^* \mathsf{F}$

8.5 Scheme

- introduced in 1975; a dialect of Lisp
  - initially very small; for teaching and research
  - now is a complete general purpose programming language
  - still derived from a small set of key concepts
- lexically scoped
- functions are first-class values
- implicit storage management

8.5.1 Interacting with a Scheme interpreter

```
"hello" ⇒ "hello"
42 ⇒ 42
22/7 ⇒ 22/7
3.141592653 ⇒ 3.141592653
+ ⇒ #<primitive:+>
(+ 5 3) ⇒ 8
'(a b c d) ⇒ (a b c d)
'(2 3) ⇒ (2 3)
(2 3) ⇒ error; 2 is not procedure
```

- a longer program should be typed in a file and loaded with

```
(load "filename")
```

8.5.2 Syntax

- identifiers
    - cannot start with a character that may start a number: digit, `+`, `-`, or `.` (except the identifiers `+`, `-`, and `...`)
    - the case is not important: `abcde` and `AbcDE` are the same
- 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: true – `#t` and false – `#f`
    - any object different from `#f` is true
- vectors: `#(this is a vector of symbols)`
- strings: `"this is a string"`
- characters: `#\a`
- comments: `;from a semicolon to the end of the line`

8.5.3 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
(+ 1/4 0.5) ⇒ 0.75
```

- defining a function

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

- anonymous functions

```
(lambda (x) (* x x))                          ;unnamed function applied to 5
((lambda (x) (* x x)) 5) ⇒ 25
```

- named functions

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

8.5.4 Quoting

- `quote` – tells Scheme to treat a list as data

examples

```
(quote ( 1 2 3 4 5)) ⇒ (1 2 3 4 5)
(quote (+ 3 4)) ⇒ (+ 3 4)
(quote +) ⇒ +                                        ;the symbol +
+ ⇒ #<primitive:+>                                ;the procedure +
(define a 5)
(quote a) ⇒ a                                        ;the symbol a
a ⇒ 5                                             ;the variable a
```

examples – `quote` may be abbreviated as `'`

```
'(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"
```

- in general: `'anything = (quote anything)`

8.5.5 Operations on lists

- `car` – gives the first element
- `cdr` – 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` – 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
```

- `list` – constructs lists; arbitrarily many arguments; builds proper lists

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

- `null?` – tests whether a list is empty

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

8.6 Lexical scope – variable binding

- *lexical scope rule* for determining the meanings of names
- local or bound variables can be renamed without changing the meaning

- variable references

```
variable                                          ;returns the value of variable
```

```
(define a 34)
a ⇒ 34
```

- local bindings

$\boxed{\texttt{(let ((}\mathit{var}\ \ \mathit{val}\texttt{)...)}\ \ \mathit{exp}_1\ \ \mathit{exp}_2\ \ \texttt{...)}}$

- 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 expressions 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*

examples

```
(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))                     ;let f be +, x be 2, ...
  (f x y)) ⇒ 5

(let ((+ *))
  (+ 2 5)) ⇒ 10
(+ 2 5) ⇒ 7                       ;+ is unchanged outside previous let
```

examples

```
(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

(let ((double (lambda (x) (+ x x))))
  (list (double (* 3 4)) (double (/ 99 11)) (double (- 2 7))))
         ⇒ (24 18 -10)
```

examples

```
(let (($x_1$ 1))                                         ;the indices show the bindings
  (let (($x_2$ (+ $x_1$ 1)))
    (+ $x_2$ $x_2$)))
⇒ 4

(let (($x_1$ 1) ($y_1$ 10))
  (let (($x_2$ (+ $y_1$ (* $x_1$ 1))))
    (+ $x_2$ (- (let (($x_3$ (+ $x_2$ $y_1$)) ($y_2$ (* $y_1$ $y_1$)))
      (- $y_2$ $x_3$)) $y_1$))))
⇒ 80


(let ((sum (lambda (ls)
             (if (null? ls)
                0
                (+ (car ls) (sum (cdr ls)))))))
                                  ;undefined sum -- sum is not in the body of let
  (sum '(1 2 3 4 5)))
```

$\boxed{\texttt{(let* ((}\mathit{var}\ \ \mathit{val}\texttt{)...)}\quad \mathit{exp}_1\ \ \mathit{exp}_2\ \ \texttt{...)}}$

- similar to `let`
- each *val* is within the scope of the variables at the left
- the expressions *val* are evaluated from left to right

example

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

`(letrec` ((*var* *val*)...) exp_1 exp_2 ...)

- similar to `let` and `let*`
- each *val* is within the scope of all variables *var*
- no order is implied for the evaluation of the expressions *val*

example

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

example

```
(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)
```

`let` – for independent variables
`let*` – linear dependency among variables
`letrec` – circular dependency among variables and order of evaluation is not important

8.7 Function creation - lambda

$\boxed{\texttt{(lambda } \textit{formals}\ \textit{exp}_1\ \textit{exp}_2\ \ldots\texttt{)}}$ - returns a function

1. *formals* is a proper list of variables (*var*$_1$ ... *var*$_n$) – exactly n parameters must be supplied and each variable is bound to the corresponding parameter

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

2. *formals* is a single variable `x` (not in a list) – `x` is bound to a list containing all actual parameters

   example

   ```
   ((lambda x x) 1 2 3) ⇒ (1 2 3)
   ((lambda x (sum x)) 1 2 3 4) ⇒ 10          ; sum has been defined before
   ```

3. *formals* is an improper list (*var*$_1$ ... *var*$_n$. *var*) terminated with a variable – 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

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

8.8 Scheme

8.8.1 Assignments

$\boxed{\texttt{(set!}\ \ var\ \ exp\texttt{)}}$ – assigns a new value to an existing variable
- this is not a new name binding but new value binding to an existing name

examples

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

▶ the above functions could 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)
```

▶ the next cannot be done without `set!`

- the following version of `cons` counts the number of times it is called in the variable `cons-count`

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

`(set-car!` *pair obj*`)`

- changes the car of *pair* to *obj*

`(set-cdr!` *pair obj*`)`

- changes the cdr of *pair* to *obj*

examples

```
(let ((x '(a b c)))
  (set-car! x 1)
  x) => (1 b c)

(let ((x '(a b c)))
  (set-cdr! x '(1 2 3))
  x) => (a 1 2 3)

(let ((x '(a b c)))
  (set-cdr! x 1)
  x) => (a . 1)
```

8.8.2 Sequencing

$\boxed{\texttt{(begin } exp_1 \; exp_2 \; \ldots\texttt{)}}$

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

example

```
(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)
```

8.8.3 Conditionals

$\boxed{\texttt{(if } \textit{test consequent alternative}\texttt{)}}$

- returns the value of *consequent* or *alternative* depending on *test*

example

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

$\boxed{\texttt{(not } \textit{obj}\texttt{)}}$ – returns `#t` if *obj* is false and `#f` otherwise

example

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

`(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

examples

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

`(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

examples

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

$\boxed{\texttt{(cond } \mathit{clause}_1\ \mathit{clause}_2\ \ldots\texttt{)}}$

- evaluates the *test* of each *clause* until one is found true or all are evaluated

example

```
(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
```

8.8.4 Recursion, iteration, mapping

$\boxed{\texttt{(let } \mathit{name}\ \texttt{((}\mathit{var}\ \mathit{val}\texttt{)...)}\ \ \mathit{exp}_1\ \mathit{exp}_2\ \texttt{...)}}$

- this is *named* `let`

- is equivalent with

((letrec ((*name* (lambda (*var* ...) exp_1 exp_2 ...)))
  name)
val ...)

example

```
(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)
```

$\boxed{\texttt{(do ((}\textit{var val update}\texttt{)}\ldots\texttt{)}\ \texttt{(}\textit{test res}\ \ldots\texttt{)}\ \ \textit{exp}\ \ldots\texttt{)}}$

- 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

example

```
(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
```

example

```
(define fibonacci
  (lambda (n)
    (if (= n 0)
        0
        (do ((i n (- i 1))(a1 1 (+ a1 a2))(a2 0 a1))
          ((= i 1) a1)))))v (fibonacci 0) ⇒ 0
(fibonacci 1) ⇒ 1
(fibonacci 2) ⇒ 1
(fibonacci 3) ⇒ 2
(fibonacci 7) ⇒ 13
```

$\boxed{\texttt{(map } \textit{procedure list}_1 \textit{ list}_2 \ldots\texttt{)}}$

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

examples

```
(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)
```

$\boxed{\texttt{(for-each } \textit{procedure list}_1 \ \textit{list}_2 \ldots\texttt{)}}$

- is similar to `map`
- does not create and return a list
- applications are from left to right

example

```
(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
```

8.8.5 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 `()`
- the `cdr` of the last pair in a proper list is the empty list (that means, `x` is a list if there is `n` such that $\texttt{cdr}^{\texttt{n}}\texttt{(x) = ()}$)
- the `cdr` of the last pair in an improper list can be anything other than `()`

example – various pairs

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

8.8.6 Predicates

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

`(pair? obj)` - `#t` if *obj* is a pair and `#f` otherwise

example

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

`(char?` *obj*`)` - `#t` if *obj* is a character and `#f` otherwise

`(string?` *obj*`)` - `#t` if *obj* is a string and `#f` otherwise

`(number?` *obj*`)` - `#t` if *obj* is a number and `#f` otherwise

`(complex?` *obj*`)` - `#t` if *obj* is complex and `#f` otherwise

`(real?` *obj*`)` - `#t` if *obj* is a real number and `#f` otherwise

`(integer?` *obj*`)` - `#t` if *obj* is integer and `#f` otherwise

`(list?` *obj*`)` - `#t` if *obj* is a list and `#f` otherwise

`(vector?` *obj*`)` - `#t` if *obj* is a vector and `#f` otherwise

`(symbol?` *obj*`)` - `#t` if *obj* is a symbol and `#f` otherwise

`(procedure?` *obj*`)` - `#t` if *obj* is a function and `#f` otherwise

8.8.7 Input / output

`(read? obj)` - returns the next object from input

`(display? obj)` - prints *obj*

example – the following program will produce the output below,

(output produced by the `display` function is italicized; input from the operator is bold faced)

```
(display "compute the square of:")          compute the square of:  4
(square (read))                             ⇒ 16
(display "find the divisors of:")           find the divisors of:  12
(display (divisors (read)))                 (2 3 4 6) is the list of divisors
(display " is the list of divisors")
(newline)
(display "give x:")                         give x:  15
(define x (read))
(+ x x)                                     ⇒ 30
(* x x)                                     ⇒ 225
```

8.9 Deep binding

- early binding of the referencing environment
- when the subroutine is passed as parameter
- restore it when called

```
(define A
  (lambda (i P)
    (let ((B (lambda () (display i))))
      (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
```

```
(define A
  (lambda (i P)
    (let ((B (lambda () (display i))))
      (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
```

8.10 Storage allocation for lists

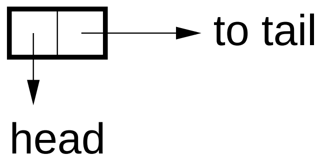
- lists are constructed with `list` and `cons`
- `list` is a shorthand version of nested `cons` functions

example `(list 'apple 'orange 'grape)`

is equivalent to `(cons 'apple (cons 'orange (cons 'grape '())))`

- memory allocation with `cons`

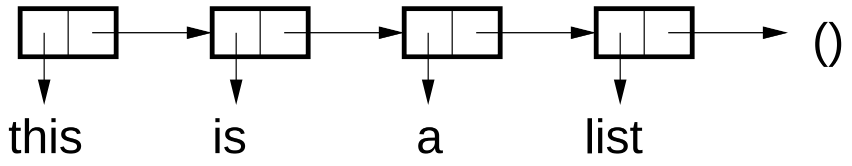
- lists are built out of cells capable of holding pointers to the head and tail (or `car` and `cdr`, resp.) of a list



- the empty list `()` is a special pointer (0 or other reserved)
- each execution of `cons` returns a pointer to a newly allocated cell

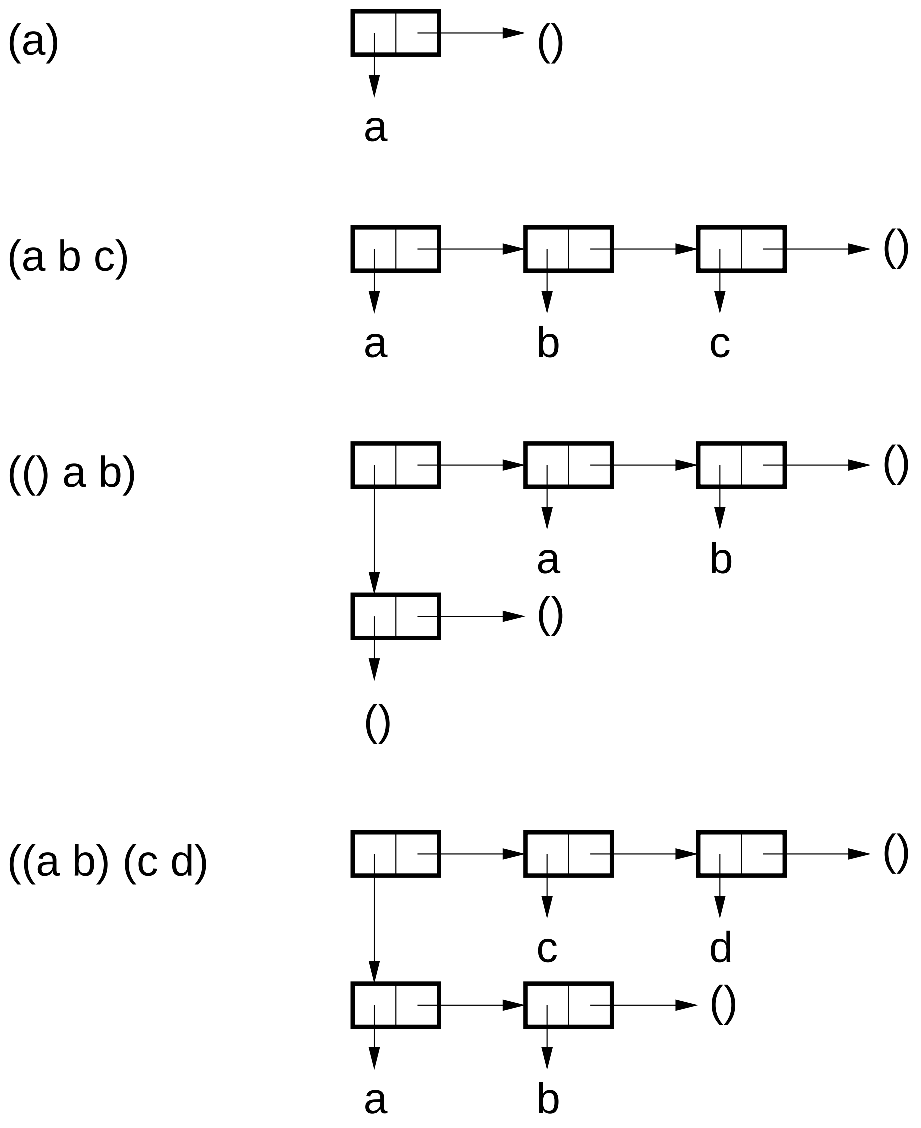
example `(cons 'this (cons 'is (cons 'a (cons 'list '()))))`

will give the list:



example

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



8.10.1 Notions of equality

`(eq?` obj_1 obj_2`)`

- returns `#t` if obj_1 and obj_2 are identical and `#f` otherwise
- implementation as fast as possible

`(eqv?` obj_1 obj_2`)`

- returns `#t` if obj_1 and obj_2 are equivalent and `#f` otherwise
- 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 and `#f` otherwise

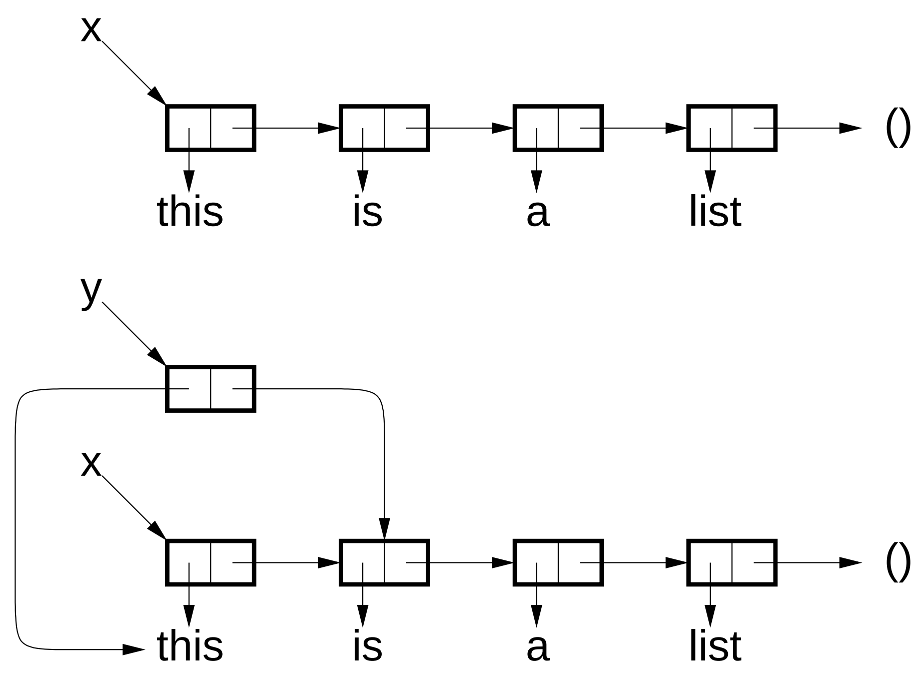
examples

```
(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?  '(a) '(a))⇒ #f
(eqv?  '(a) '(a))⇒ #f
(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
(= 3.4 (+ 3.0 .4)) ⇒ #t
```

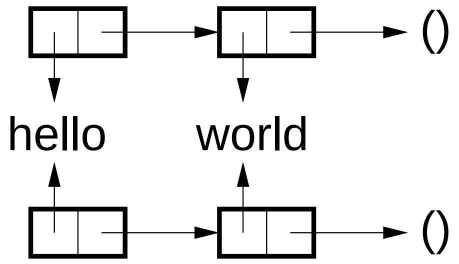
example - why the difference?

```
(define x '(this is a list))
(define y (cons (car x) (cdr x)))
(eq?  x y) ⇒ #f
(eqv?  x y) ⇒ #f
(equal?  x y) ⇒ #t
```



example - why the difference?

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



8.10.2 List searching

$\boxed{\texttt{(memq}\ \mathit{obj}\ \mathit{list}\texttt{)}}$ $\boxed{\texttt{(memv}\ \mathit{obj}\ \mathit{list}\texttt{)}}$ $\boxed{\texttt{(member}\ \mathit{obj}\ \mathit{list}\texttt{)}}$

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

$\boxed{\texttt{(assq}\ \mathit{obj}\ \mathit{alist}\texttt{)}}$ $\boxed{\texttt{(assv}\ \mathit{obj}\ \mathit{alist}\texttt{)}}$ $\boxed{\texttt{(assoc}\ \mathit{obj}\ \mathit{alist}\texttt{)}}$

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

examples

```
(assv 'b '((a . 1) (b . 2))) => (b . 2)
(assv 2/3 '((1/3 . a) (2/3 . b))) => (2/3 . b)
(assoc '(a) '(((b) . b) (a . c))) => #f
```

8.11 Evaluation order

- λ-calculus – evaluation (reduction) can be done in several ways
- **applicative order** – parameters evaluated before passed
- **normal order** – parameters passed unevaluated
- any of the two can be shorter in some cases
- **Scheme uses applicative order**

example 1

```
(define double (lambda (x) (+ x x)))
```

▶ evaluation in applicative order (as in Scheme)

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

▶ evaluation in normal order – is doing extra work: `(* 3 4)` is evaluated twice

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

example 2

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

▸ evaluation in applicative order (as in Scheme)

```
(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
```

▸ evaluation in normal order (less work)

```
(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
```

8.11.1 Strict and lazy evaluation

- strict function – if it requires all arguments to be defined; the result does not depend on the order of evaluation

- strict language – requires all functions to be strict

- expressions can be evaluated safely in applicative order

- strict – Scheme and ML
- nonstrict – Miranda and Haskell

lazy evaluation – not evaluating unneeded subexpressions

- gives the advantages of normal order evaluation
- problem: its behaviour in the presence of side effects
- evaluating sooner or later might not be the same

- in Scheme – simulated using `delay` and `force`
  - typically used in the absence of side effects

`(delay` *exp*`)` – returns a promise which can be later forced to evaluate

`(force` *promise*`)` – returns the result of forcing *promise*

example – *streams* – conceptually infinite lists

```
(define stream-car
  (lambda (s)
    (car (force s))))
(define stream-cdr
  (lambda (s)
    (cdr (force s))))
(define counters
  (let next ((n 1))
    (delay (cons n (next (+ n 1))))))
(trace stream-car)
(trace stream-cdr)
(stream-car counters) => 1
(stream-car (stream-cdr counters)) => 2
(stream-car (stream-cdr (stream-cdr counters))) => 3
```

```
(define stream-add
  (lambda (s1 s2)
    (delay (cons
            (+ (stream-car s1) (stream-car s2))
            (stream-add (stream-cdr s1) (stream-cdr s2))))))
(define even-counters
  (stream-add counters counters))
(stream-car even-counters) => 2
(stream-car (stream-cdr even-counters)) => 4
(stream-car (stream-cdr (stream-cdr even-counters))) => 6
```

8.12 Higher-order functions

- **higher-order function** – a function which takes a function as an argument

examples

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
(define cadr (compose car cdr))

(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

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

example – folding the elements of a list together, using an associative binary operator

```
(define fold (lambda (f l i)
  (if (null?  l) i                ;i is commonly the identity element for f
    (f (car l) (fold f (cdr l) i)))))
(fold + '(1 2 3) 0) ⇒ 6
```

example – one of the most common uses of higher-order functions is to build new functions from existing ones

```
(define total
  (lambda (l)
    (fold + l 0)))
(total '(1 2 3 4 5)) ⇒ 15

(define total-all
  (lambda (l)
    (map total l)))
(total-all '((1 2 3 4) (2 4 6 8) (3 6 9 12)))⇒ (10 20 30)
```

```
(define make-double
  (lambda (f)
    (lambda (x) ( f x x))))
(define twice (make-double +))
(twice 10) ⇒ 20
(define square (make-double *))
(square 10)⇒ 100
```

example – composing arbitrarily many 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
```

8.12.1 Currying

- replace a multiargument function with a function that takes a single argument and returns a function that expects the remaining arguments

example

```
(define curried-plus
  (lambda (a)
    (lambda (b) (+ a b))))
(define plus-3 (curried-plus 3))
(plus-3 4) ⇒ 7
```

- we have the possibility to pass a partially applied function to a higher-order function

```
(map plus-3 '(10 20 30)) => (13 23 33)
```

example - a function which curries a binary function

```
(define curry
  (lambda (f)
    (lambda (arg1)
      (lambda (arg2)
        (f arg1 arg2)))))
(define curried-plus
  (curry +))
(define plus-3 (curried-plus 3))
(plus-3 4) => 7
```

8.13 Examples

8.13.1 Finite automata simulation

```
(define simulate
  (lambda (dfa input)
    (cons (car dfa)                      ; start state
          (if (null? input)
              (if (infinal? dfa) '(accept) '(reject))
              (simulate (move dfa (car input)) (cdr input))))))

(define infinal?
  (lambda (dfa)
    (memq (car dfa) (caddr dfa))))

(define move
  (lambda (dfa symbol)
    (let ((curstate (car dfa)) (trans (cadr dfa)) (finals (caddr dfa)))
      (list
       (if (eq? curstate 'error)
           'error
           (let ((pair (assoc (list curstate symbol) trans)))
             (if pair (cadr pair) 'error)))
       trans
       finals))))
```

```
(simulate
 '(q0                                                ; start state
   (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)  ; transitions
    ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
   (q0))                                             ; final states
 '(0 1 1 0 1))                                      ; input string
=> (q0 q2 q3 q2 q0 q1 reject)

(simulate
     '(q0                                                ; start state
       (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)  ; transitions
        ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
       (q0))                                             ; final states
     '(0 1 0 0 1 0))                                       ; input string
=> (q0 q2 q3 q1 q3 q2 q0 accept)
```

8.13.2 Symbolic differentiation

- rules for symbolic differentiation

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

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

$$\frac{d}{dx}(y) = 0 \qquad y \text{ variable different from } x$$

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

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

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

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

```
(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)))
            )))))
```

```
(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))))
```

```
(diff 'x '(+ (* 2 x) 1))
  = (list '+ (diff 'x '(* 2 x)) (diff 'x 1))
  = (list '+ (list '+ (list '* 2 (diff 'x 'x))
                      (list '* 'x (diff 'x 2)))
              (diff 'x 1))
  = (list '+ (list '+ (list '* 2 1) (list '* 'x (diff 'x 2)))
              (diff 'x 1))
  = (list '+ (list '+ '(* 2 1) (list '* 'x (diff 'x 2)))
              (diff 'x 1))
  = (list '+ (list '+ '(* 2 1) (list '* 'x 0)) (diff 'x 1))
  = (list '+ (list '+ '(* 2 1) '(* x 0)) (diff 'x 1))
  = (list '+ '(+ (* 2 1) (* x 0)) (diff 'x 1))
  = (list '+ '(+ (* 2 1) (* x 0)) 0)
  = (+ (+ (* 2 1) (* x 0)) 0)
```

```
(require-library "trace.ss")
(trace diff)
(diff 'x '(+ (* 2 x) 1))

=>
|(diff x (+ (* 2 x) 1))
| (diff x (* 2 x))
| |(diff x x)
| |1
| |(diff x 2)
| |0
| (+ (* 2 1) (* x 0))
| (diff x 1)
| 0
|(+ (+ (* 2 1) (* x 0)) 0)
```

8.13.3 Stacks

```
(define make-stack
  (lambda ()
    (let ((ls '()))
      (lambda (msg . args)
        (cond
          ((eqv? msg 'empty?) (null? ls))
          ((eqv? msg 'push!)
           (set! ls (cons (car args) ls)))
          ((eqv? msg 'top) (car ls))
          ((eqv? msg 'pop!)
           (set! ls (cdr ls)))
          (else "oops!")))))))
```

```
(define stack1 (make-stack))
(define stack2 (make-stack))
(stack1 'empty?) => #t
(stack2 'empty?) => #t
(stack1 'push! 'a)
(stack1 'empty?) => #f
(stack2 'empty?) => #t
(stack1 'push! 'b)
(stack2 'push! 'c)
(stack1 'top) => b
(stack2 'top) => c
(stack1 'pop!)
(stack2 'empty?) => #f
(stack1 'top) => a
(stack2 'pop!)
(stack2 'empty?) => #t
(stack2 'full?) => "oops!"
```

9 Logic programming

- logic programming
  - algorithm = logic + control
  - computing with relations
- theoretical foundations – predicate calculus
- Prolog

9.1 Algorithm = Logic + Control

- **logic** - the facts and rules specifying what the algorithm does
  - *logic programs* consist of facts and rules
  - the programmer supplies the logic part

- **control** - how the algorithm can be implemented by applying the rules in a particular order
  - *computation* is deduction
  - the language supplies the control

- **rule:** P **if** Q_1 **and** $\cdots$ **and** Q_k, for some $k \geq 0$
- **meaning:**

  to deduce P,
    deduce Q_1; deduce Q_2; $\cdots$ deduce Q_k;

- problem: this strategy might get stuck in an infinite loop

Prolog - practical tool
  - it introduces few impurities to logic but it is still very close

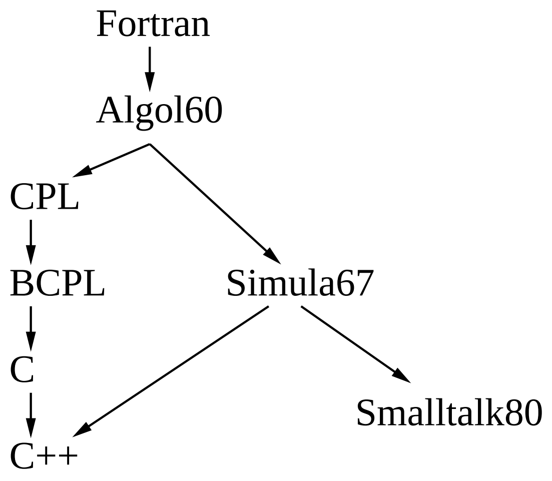
9.1.1 Computing with relations

- premise – programming with relations is more flexible than with functions because relations treat arguments and results uniformly (does not matter who is computed from whom, no sense of direction)

- relations - given n sets $S_1, S_2, \ldots, S_n$, an n-ary relation of $S_1 \times S_2 \times \cdots \times S_n$ is a subset of $S_1 \times S_2 \times \cdots \times S_n$.

example 1 – a directed graph:

- the set of edges E is a binary relation on $S \times S$, where S is the set of nodes (the 8 languages, in this case)



S = {Fortran, Algol60, CPL, BCPL, C, C++, Simula67, Smalltalk80}

E = {(Fortran, Algol60), (Algol60, CPL), (Algol60, Simula67),
(CPL, BCPL), (BCPL, C), (Simula67, Smalltalk80),
(Simula67, C++), (C, C++)}

example 2 $plus = \{(x, y, z) \mid x + y = z\}$

- is a ternary relation on $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$

- predicates

- a relation can be seen as predicate:

$$plus(x, y, z) = \begin{cases} \text{true}, & \text{if } (x, y, z) \in plus \\ \text{false}, & \text{otherwise} \end{cases}$$

example 3

Let L be a set of lists. The relation $append \subseteq L \times L \times L$ is defined by

$(x, y, z) \in append$ iff z is the catenation of x and y

- the corresponding predicate is:

$$append(x, y, z) = \begin{cases} \text{true}, & \text{if } z \text{ is } x \text{ catenated with } y \\ \text{false}, & \text{otherwise} \end{cases}$$

9.1.2 Rules, facts, queries

- **rule (clause)** – a statement of the form:

$$\boxed{P \quad \textbf{if} \;\; Q_1 \;\; \textbf{and} \;\; \cdots \;\; \textbf{and} \;\; Q_k.} \quad \text{(for some } k \geq 0\text{)}$$

- P - the *head* of the rule
- $Q_1, Q_2, \ldots, Q_k$ - the *conditions* (*body*)
- meaning – P is true if $Q_1, Q_2, \ldots, Q_k$ are true
- rules – used to specify relations

- **fact** – a rule with no conditions ($k = 0$)

$$\boxed{P.}$$

- meaning: P holds (with no condition)

example

```
append([], Y, Y).                                  % fact
append([H|X], Y, [H|Z]) :- append(X, Y, Z).          % rule

female(liz).
male(tom).
parent(tom, liz).
mother(X, Y) :- parent(X, Y), female(X).
grandparent(X, Z) :-  parent(X, Y), parent(Y, Z).
sister(X, Y) :-  parent(Z, X), parent(Z, Y), female(X),
    different(X, Y).
```

- query (goal) – a statement asking whether a relation holds between objects
- it has the form:

$\boxed{Q.}$ or $\boxed{Q_1, Q_2, \ldots, Q_m.}$ $(m \geq 1)$

- meaning – Q_1 and Q_2 and . . . and Q_m
- answer
  - **yes** if all relations hold
  - **no** if at least one fails
- $Q_1, Q_2, \ldots, Q_m$ – **subgoals**

example

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

?- append([a], [b,c], [a,b,c]).
    =>  yes
?- append([a,b], [c,d], Z).
    =>  Z = [a,b,c,d]
?- append([a,b], Y, [a,b,c,d]).
    =>  Y = [c,d]
?- append(X, [c,d], [a,b,c,d]).
    =>  X = [a,b]
?- append([a], [b], [c]).
    =>  no
?- append(X, [a], [b]).
    =>  no
```

9.2 Existential/universal quantifications

- queries are implicitly existentially quantified

example

```
?- link(algol60, X), link(X, Y).
```

is, with quantifiers in place,

$\exists X, Y,\ link(algol60, X)$ **and** $link(X, Y)$

- rules are implicitly universally quantified

example

```
path(L, M) :- link(L, X), path(X, M).
```

is, with quantifiers in place,

$\forall L, M, X,\ path(L, M)$ **if** $(link(L, X)$ **and** $path(X, M))$

or

$\forall L, M,\ path(L, M)$ **if** $(\exists X,\ \ link(L, X)$ **and** $path(X, M))$

9.3 Negation

- **negation as failure**
  - `no` means Prolog failed to deduce the query from the given facts
  - `no` - is not true logical negation

- **not** - operator

`not(P)` is treated as true if Prolog fails to deduce P

example

```
?- link(L, N), link(M, N), not(L=M).
    =>  L = c
        N = cplusplus
        M = simula67 ;
    =>  L = simula67
        N = cplusplus
        M = c ;
    =>  no
?- not(L=M), link(L, N), link(M, N).
    =>  no       % L and M are not known at the start of
                 % the query and therefore could be equal
```

9.4 Unification

- **instance** of a term T is obtained by consistently substituting subterms for one or more variables of T

example `f(a, b, a, c)` is an instance of `f(X, b, X, Y)`

- **unification**
  - two terms **unify** if they have a common instance
  - if a variable occurs in both, then the same subterm must be substituted for all occurrences of that variable in both terms

example 1 `f(X, b, X, Y)` and `f(a, Z, W, X)` unify

- the common instance is `f(a,b,a,a)`

example 2 `f(X, b, X, Z)` and `f(a, Z, W, X)` do not unify

```
?- f(X,b,X,Y) = f(a,Z,W,X).      % '=' stands for unification
    =>  X = a
        Y = a
        Z = b
        W = a
    =>  yes
?- f(X,b,X,Z) = f(a,Z,W,X).
    =>  no
?- 5 = 5.
    =>  yes
?- 5 = 3 + 2.                   % '=' unifies and not evaluates
    =>  no
```

9.5 Predicate calculus

- predicate – a function which maps constants and variables to truth values true and false

- **(first order) predicate calculus** – provides a notation and inference rules for constructing and reasoning about *propositions*
  - **operators:** and $\wedge$, or $\vee$, not $\neg$, implication $\longrightarrow$, and equivalence $\leftrightarrow$
  - **quantifiers:** existential $\exists$ and universal $\forall$

examples

$$\forall C(\mathsf{rainy}(C) \wedge \mathsf{cold}(C) \longrightarrow \mathsf{snowy}(C))$$

$$\forall A, \forall B(\exists C(\mathsf{takes}(A, C) \wedge \mathsf{takes}(B, C)) \longrightarrow \mathsf{classmates}(A, B))$$

$$\forall N((N > 2) \longrightarrow \neg(\exists A, \exists B, \exists C(A^N + B^N = C^N)))$$

- $\exists$ and $\forall$ bind variables like λ in lambda calculus

9.5.1 Clausal form

example – normal form (suitable for automatic proving)

$\forall X(\neg\mathsf{student}(X) \rightarrow (\neg\mathsf{resident}(X) \wedge \neg\exists Y(\mathsf{takes}(X,Y) \wedge \mathsf{class}(Y))))$

1. eliminate $\rightarrow$ and $\leftrightarrow$

$\forall X(\mathsf{student}(X) \vee (\neg\mathsf{resident}(X) \wedge \neg\exists Y(\mathsf{takes}(X,Y) \wedge \mathsf{class}(Y))))$

2. move $\neg$ inward – using De Morgan laws

$\forall X(\mathsf{student}(X) \vee (\neg\mathsf{resident}(X) \wedge \forall Y(\neg(\mathsf{takes}(X,Y) \wedge \mathsf{class}(Y)))))$

$\equiv \forall X(\mathsf{student}(X) \vee (\neg\mathsf{resident}(X) \wedge \forall Y(\neg\mathsf{takes}(X,Y) \vee \neg\mathsf{class}(Y))))$

3. eliminate existential quantifiers - Skolemization (not here)

4. pull universal quantifiers to the outside of the proposition (some renaming might be needed)

$\forall X\forall Y(\mathsf{student}(X) \vee (\neg\mathsf{resident}(X) \wedge (\neg\mathsf{takes}(X,Y) \vee \neg\mathsf{class}(Y))))$

- convention: rules are universally quantified:

$\mathsf{student}(X) \vee (\neg\mathsf{resident}(X) \wedge (\neg\mathsf{takes}(X,Y) \vee \neg\mathsf{class}(Y)))$

5. convert the proposition in conjunctive normal form

$(\mathsf{student}(X) \vee \neg\mathsf{resident}(X)) \wedge (\mathsf{student}(X) \vee \neg\mathsf{takes}(X,Y) \vee \neg\mathsf{class}(Y))$

- we can rewrite
$(\mathsf{student}(X) \vee \neg\mathsf{resident}(X)) \wedge(\mathsf{student}(X) \vee \neg\mathsf{takes}(X,Y) \vee \neg\mathsf{class}(Y))$
as
$(\mathsf{resident}(X) \rightarrow \mathsf{student}(X)) \wedge(\mathsf{takes}(X,Y) \wedge \mathsf{class}(Y)) \rightarrow \mathsf{student}(X))$

- this translates directly to Prolog:

```
student(X) :- resident(X).
student(X) :- takes(X, Y), class(Y).
```

9.5.2 Horn clauses

- in general, the clauses (rules) in Prolog have the form

$$\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_k \vee P \equiv Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k \rightarrow P$$

- which is, in Prolog:

```
P :- Q1, Q2,...,Qk.
```

- for $k = 0$ we have a fact

```
P.
```

9.5.3 Skolemization

- so far we did not worry about existential quantifiers
- what if we have:

$$\exists X(\mathsf{takes}(X, 342) \wedge \mathsf{year}(X, 4))$$

- to get rid of the $\exists$, we introduce a constant a (as a notation for the one which is assumed to exists by $\exists$)

$$\mathsf{takes}(\mathsf{a}, 342) \wedge \mathsf{year}(\mathsf{a}, 4)$$

- what if we do this inside the scope of a universal quantifier $\forall$:

$$\forall X(\neg\mathsf{resident}(X) \vee \exists Y(\mathsf{address}(X, Y)))$$

- we get rid again of $\exists$ by choosing an address which *depends* on X, say $\mathsf{ad}(X)$:

$$\forall X(\neg\mathsf{resident}(X) \vee \mathsf{address}(X, \mathsf{ad}(X)))$$

- in Prolog:

```
takes(a, 342).
year(a, 4).
address(X, ad(X)) :- resident(X).

class_with_4th(C) :- takes(X, C), year(X, 4).
has_address(X) :- address(X, Y).
resident(b).

?- class_with_4th(X).
    =>  X = 342
    =>  yes
?- has_address(X).
    =>  X = b
    =>  yes
?- takes(X, 342).
    =>  X = a       % we cannot identify a 4th-year student
    =>  yes         % in 342 by name
?- address(b, X).
    =>  X = ad(b)   % we cannot find out the address of b
    =>  yes
```

9.6 Automated proving

- rules – both sides of `:-`

`P :- Q1, Q2,...,Qk.` means $\mathtt{P} \longleftarrow \mathtt{Q1} \wedge \mathtt{Q2} \wedge \ldots \wedge \mathtt{Qk}$

- facts – entered to the left-hand side of the (implicit) `:-`

`P.` means $\mathtt{P} \longleftarrow \mathsf{true}$

- queries – entered to the right-hand side of the (implicit) `:-`

`?- Q1, Q2,...,Qk.` means $\mathsf{false} \longleftarrow \mathtt{Q1} \wedge \mathtt{Q2} \wedge \ldots \wedge \mathtt{Qk}$

- this suggest a mechanism for automated proving: given a collection of axioms (facts and rules), we add the *negation* of the theorem (query) we want to prove and attempt (using *resolution*) to obtain a contradiction

example 1

```
student(john).
?- student(john).
    =>  yes
```

- applying resolution to

$$\mathsf{student(john)} \leftarrow \mathsf{true} \quad \text{and} \quad \mathsf{false} \leftarrow \mathsf{student(john)}$$

will give a contradiction:

$$\mathsf{false} \leftarrow \mathsf{true}$$

example 2

```
student(X) :- resident(X).
student(X) :- takes(X, Y), class(Y).
resident(a).
takes(b, 342).
class(342).
?- student(a).
    =>  yes
```

- we apply resolution

$(\neg\mathsf{resident}(X) \vee \mathsf{student}(X)) \wedge (\neg\mathsf{takes}(X,Y) \vee \neg\mathsf{class}(Y) \vee \mathsf{student}(X)) \wedge \mathsf{resident}(a) \wedge \mathsf{takes}(b,342) \wedge \mathsf{class}(342) \wedge \neg\mathsf{student}(a)$

$(\neg\mathsf{resident}(a)) \wedge (\neg\mathsf{takes}(a,Y) \vee \neg\mathsf{class}(Y)) \wedge \mathsf{resident}(a) \wedge \mathsf{takes}(b,342) \wedge \mathsf{class}(342)$

$(\square) \wedge (\neg\mathsf{takes}(a,Y) \vee \neg\mathsf{class}(Y) \vee \mathsf{student}(a)) \wedge \mathsf{takes}(b,342) \wedge \mathsf{class}(342)$

- this formula has the empty clause $\square$ which is not satisfiable, that is, we have a contradiction showing that $\mathsf{student}(a)$ is provable

```
?- student(b).
    =>  yes
```

- similar

```
?- student(c).
    =>  no
```

$(\neg\mathsf{resident}(X) \vee \mathsf{student}(X)) \wedge (\neg\mathsf{takes}(X,Y) \vee \neg\mathsf{class}(Y) \vee \mathsf{student}(X)) \wedge \mathsf{resident}(a) \wedge \mathsf{takes}(b,342) \wedge \mathsf{class}(342) \wedge \neg\mathsf{student}(c)$

$\neg\mathsf{resident}(c) \wedge (\neg\mathsf{takes}(c,342) \vee \neg\mathsf{class}(342) \vee \mathsf{student}(c)) \wedge \mathsf{resident}(a) \wedge \mathsf{takes}(b,342) \wedge \mathsf{class}(342)$

$\neg\mathsf{resident}(c) \wedge (\neg\mathsf{takes}(c,342) \vee \mathsf{student}(c)) \wedge \mathsf{resident}(a) \wedge \mathsf{takes}(b,342)$

$\emptyset$

- the empty formula $\emptyset$ is satisfiable, therefore we cannot deduce a contradiction, and so $\mathsf{student}(c)$ cannot be proved

9.6.1 Limitations

- Horn clauses can capture most but not all of first-order predicate calculus
- the problem: a Horn clause has only one non-negated term

(i) - if we have more than one, for example:

$$\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_k \vee P_1 \vee P_2$$

then we have a disjunction in the left-hand side of $\leftarrow$ (`:-`)

$$P_1 \vee P_2 \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

or `P1 or P2 :- Q1, Q2,...,Qk` which is forbidden

(ii) - if we have less than one, that is, none:

$$\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_k$$

then we have nothing in the left-hand side of $\leftarrow$ (`:-`)

$$\leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_k$$

or `:- Q1, Q2,...,Qk` which Prolog allows as query not rule

example 1 – two heads

- consider the statement

"every living thing is an animal or a plant"

- clausal form:

$$\mathsf{animal}(X) \vee \mathsf{plant}(X) \leftarrow \mathsf{living}(X)$$

or, equivalently

$$\mathsf{animal}(X) \vee \mathsf{plant}(X) \vee \neg\mathsf{living}(X)$$

- because of the restriction of one term in the left-hand side, the closest in Prolog is:

```
animal(X) :- living(X), not(plant(X)).
plant(X) :- living(X), not(animal(X)).
```

which is not the same, because `not` indicates Prolog's inability to prove, not falsity

example 2 – empty head

- consider Fermat's last theorem (abstracting out the math)

$$\forall N(\mathsf{big}(N) \to \neg(\exists A, \exists B, \exists C(\mathsf{works}(A, B, C))))$$

- clausal form:

$$\neg\mathsf{big}(N) \vee \neg\mathsf{works}(A, B, C)$$

- in Prolog, we can have the query which never terminates:

```
?- big(N), works(A, B, C, N).
```

9.7 Prolog

9.7.1 Terms

- used in Prolog to specify facts, rules, and queries
- *simple terms*
  - *numbers:* 0, 1972
  - *variables* starting with a capital letter or underscore: X, Source, _1, _x23
  - *atoms:* lisp, algol60
- *compound (structured) term* – an atom (called *functor*) followed by a parenthesized sequence of subterms (called *arguments*)

example `link(bcpl, c), link(algol60, cpl)`

- syntax of rules, facts, and queries:

$$\begin{array}{lcl}
\langle fact\rangle & ::= & \langle term\rangle. \\
\langle rule\rangle & ::= & \langle term\rangle \ {:}{-}\ \langle terms\rangle. \\
\langle query\rangle & ::= & \langle terms\rangle. \\
\langle term\rangle & ::= & \langle number\rangle \mid \langle atom\rangle \mid \langle variable\rangle \\
 & & \qquad \mid \langle atom\rangle(\langle terms\rangle) \\
\langle terms\rangle & ::= & \langle term\rangle \mid \langle term\rangle, \langle terms\rangle
\end{array}$$

9.7.2 Interacting with Prolog

- system prompt: `?-`

- `consult(filename).` – reads the file `filename.pl` containing facts and rules

- after a query with no variables: Prolog answers `yes` or `no`
- after a query with variables:
  - Prolog answer with a **solution** = a binding of the variables that makes the query true or `no` if there is no solution

  - after a solution, there are two choices:
    (1) type `<return>` – Prolog answers `yes` (indicating that there may be other solutions) and then gives the prompt
    (2) type `;` – Prolog answers with another solution or `no` if there are no further solutions

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).
```

```
?- link(bcpl, c), link(c, cplusplus).
    =>  yes
?- link(bcpl, c), link(algol60, c).
    =>  no
?- link(simula67, X).
    =>  X = cplusplus ;           % ; introduced by user
    =>  X = smalltalk80 ;
    =>  no
?- link(algol60, X), link(X, Y).
    =>  X = cpl
        Y = bcpl ;
    =>  X = simula67
        Y = cplusplus ;
    =>  X = simula67
        Y = smalltalk80 ;
    =>  no
?- path(Y, Y).
    =>  Y = _1 ;
    =>  no
?- path(a, a).
    =>  yes
```

```
?- path(fortran, cplusplus).
    =>  yes
?- path(X, cpl).
    =>  X = cpl ;
    =>  X = fortran ;
    =>  X = algol60 ;
    =>  no
```

9.7.3 Arithmetic

`is` – operator which evaluates an expression

example

```
?- 5 = 3 + 2.
    =>  no
?- 5 is 3 + 2.
    =>  yes
```

```
?- X is 2 + 3, X = 2 + 3.
    =>  no
?- X is 2 + 3, X = 5.
    =>  X = 5
    =>  yes
```

- all variables in the right argument of `is` must be instantiated to numbers; otherwise error

example

```
?- X is 3, Y is 4 + X.
    =>  X = 3
        Y = 7
    =>  yes
?- X is W, Y is 4 + X.
    =>  ERROR: Arguments are not sufficiently instantiated
```

- basic arithmetic operators:

`+` – addition
`-` – subtraction
`*` – multiplication
`/` – division
`**` – power
`//` – integer division
`mod` – modulo

- comparison operators:

`>` – greater
`<` – less than
`>=` – greater or equal
`=<` – less than or equal
`=:=` – equal
`=\=` – not equal

example 1

```
odd(1).
odd(X) :- Y is X - 2, odd(Y).

odd1(X) :- (X > 0), 1 is (X mod 2).
odd1(X) :- (X =< 0), -1 is (X mod 2).

?- odd(3).
    =>  yes
?- odd(4).
        [infinite computation]
?- odd(-5).
        [infinite computation]
?- odd1(4).
    =>  no
?- odd1(-5).
    =>  yes
```

example 2

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

?- gcd(12, 15, D).
    =>  D = 3
    =>  yes
```

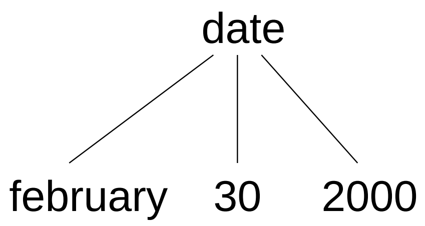
9.7.4 Structures

- objects that have several components
- the components can be in turn structured

example 1 `date(february, 30, 2000)`

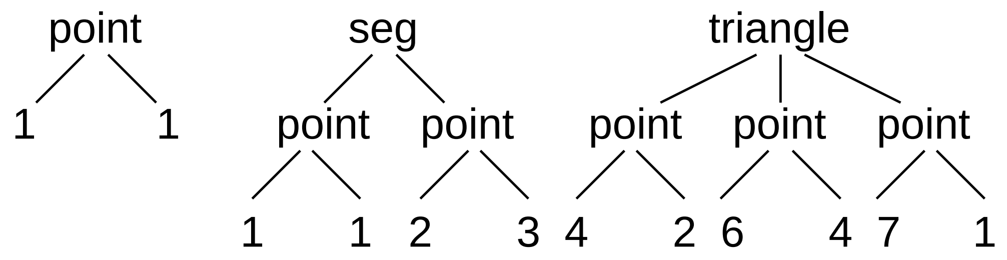
example 2 `date(February, 30, 2000)`

- the method for data structuring is simple and powerful
  - Prolog is naturally applied to problems involving symbolic manipulation
- all structured objects can be represented as trees



example 3

```
point(1, 1)
seg(point(1, 1), point(2, 3))
triangle(point(4, 2), point(6, 4), point(7, 1))
```



9.7.5 Matching

- two terms S and T **match** (**unify**) if:
  - (i) - if S and T are the same constants
  - (ii) - if S is a variable and T is anything; S is instantiated to T
  - (iii) - is S and T are structures and
    - (a) - they have the same principal functor
    - (b) - the corresponding components match
  - the instantiation is determined from the matching of the components

examples

```
?- date(M, D, 2000)
         = date(M1, 30, Y1).
    =>  M = _1
        D = 30
        M1 = _1
        Y1 = 2000
    =>  yes
```

```
?- triangle(point(1, 1), A, point(2, 3))
    = triangle(X, point(4, Y), point(2, Z)).
    =>  A = point(4, _1)
        X = point(1, 1)
        Y = _1
        Z = 3
    =>  yes
```

9.7.6 Lists

▶ enumeration – simplest way

example `[a, b, c]`

▶ as an initial sequence and a trailing list, separated by '`|`'

example

```
[a, b, c | []]
[a, b | [c]]
[a | [b, c]]
```

- when we write `[H | T]`, it means
  - `H` is the *head* of the list (the first element)
  - `T` is the *tail* of the list (the list of the remaining elements)

▶ the connection between terms and lists:

`[H|T]` and `.(H, T)` are the same

example

```
?- X = .(a, .(b, .(c, []))).
    =>  X = [a, b, c]
    =>  yes
```

(the dot functor '.' is the same as the `cons` operator)

examples

```
?- [H|T] = [a, b, c]
    =>  H = a
        T = [b, c]
    =>  yes

?- [H|T] = [[], c|[ [], b, []|a]].
    =>  H = []
        T = [c, [], b, []|a]
    =>  yes
?- [H|[X|T]] = [[], c|[ [], b, []|a]].
    =>  H = []
        X = c
        T = [[], b, []|a]
    =>  yes
?- [H1, H2|[X|T]] = [[], c|[ [], b, []|a]].
    =>  H1 = []
        H2 = c
        X = []
        T = [b, []|a]
    =>  yes
```

9.7.7 Operations on lists

- **searching** an element in a list

```
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
```

the underscore '_' is the anonymous (don't care) variable
- each occurrence of it represents a different variable

example

```
(_, _) = (a, b).
    =>  yes
```

- **appending** two lists

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

- **adding** an element in front of a list

```
add(X, L, [X|L]).
```

- **deleting** an element from any position it occurs

```
del(X, [X|Tail], Tail).
del(X, [Y|Tail], [Y|Tail1]) :- del(X, Tail, Tail1).
?- del(a, [a, b, c, a, b, a, d, e, a], X).
    =>  X = [b, c, a, b, a, d, e, a] ;
    =>  X = [a, b, c, b, a, d, e, a] ;
    =>  X = [a, b, c, a, b, d, e, a] ;
    =>  X = [a, b, c, a, b, a, d, e] ;
    =>  no
```

- **sublist** of a list

```
sublist(S, L) :- append(_, L1, L), append(S, _, L1).
```

- **subset** – list considered as sets

```
subset([], S).
subset([H|Tail], S) :- member(H, S), subset(Tail, S).
```

- **reversing** a list

```
reverse([], []).
reverse([H|Tail],R) :- reverse(Tail,R1), append(R1,[H],R).
```

- **inserting** in any place in a list

```
insert(X, L, L1) :- del(X, L1, L).

?- insert(a, [a, b, d, a, c], L).
    =>  L = [a, a, b, d, a, c] ;
    =>  L = [a, a, b, d, a, c] ;
    =>  L = [a, b, a, d, a, c] ;
    =>  L = [a, b, d, a, a, c] ;
    =>  L = [a, b, d, a, a, c] ;
    =>  L = [a, b, d, a, c, a] ;
    =>  no
```

- **permuting** a list in all possible ways

```
permute([], []).
permute([H|Tail], P) :- permute(Tail, P1), insert(H, P1, P).

?- permute([a, b, c], X).               =>  X = [a, c, b] ;
    =>  X = [a, b, c] ;                 =>  X = [c, a, b] ;
    =>  X = [b, a, c] ;                 =>  X = [c, b, a] ;
    =>  X = [b, c, a] ;                 =>  no
```

9.8 Control

- the way a response to a query is computed
- characterized by two decisions:
  - **goal order** – *choose the leftmost subgoal*
  - **rule order** – *select the first applicable rule*
- the response to a query is affected both by
  - goal order within the query
  - rule order within the database of facts and rules

- 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 the subgoal then
      select the first applicable rule not already used;
      form a new current goal
   else
      if at the root then
         fail
      else
         backtrack
      end if
   end if
end while;
succeed
```

example

```
link(fortran,  algol60).         link(c,        cplusplus).
link(algol60,  cpl).             link(algol60,  simula67).
link(cpl,     bcpl).             link(simula67, cplusplus).
link(bcpl,    c).                link(simula67, smalltalk80).

path1(X, X, 0).
path1(X, Y, L) :- link(X, Z), path1(Z, Y, L1), L is L1 + 1.
?- path1(X, cplusplus, L).
```

```
    =>  X = cplusplus
        L = 0 ;
    =>  X = fortran
        L = 5 ;
    =>  X = fortran
        L = 3 ;
    =>  X = algol60
        L = 4 ;
    =>  X = cpl
        L = 3 ;
    =>  X = bcpl
        L = 2 ;
    =>  X = c
        L = 1 ;
    =>  X = algol60
        L = 2 ;
    =>  X = simula67
        L = 1 ;
    =>  no
```

9.8.1 Unification and substitution

▶ **substitution** – a function from variables to terms

example $\sigma = \{X \rightarrow [a,b], Y \rightarrow [a,b,c]\}$

- unless stated otherwise, if a substitution maps X to T, then X does not occur in T
- $T\sigma$ – the result of applying the substitution σ to the term T

▶ **application rule** – let X be a variable and σ a substitution

$$X\sigma = \begin{cases} U & \text{if } X \rightarrow U \text{ is in } \sigma \\ X & \text{otherwise} \end{cases}$$

$(f(T_1, T_2, \ldots, T_n))\sigma = f(U_1, U_2, \ldots, U_n)$ if $T_1\sigma = U_1, \ldots, T_n\sigma = U_n$

example

$\sigma = \{X \rightarrow [a,b], Y \rightarrow [a,b,c]\}$
$Y\sigma = [a,b,c]$
$Z\sigma = Z$
$append([], Y, Y)\sigma = append([], [a,b,c], [a,b,c])$

- a term U is an **instance** of T if $U = T\sigma$ for some substitution σ
- two terms T_1 and T_2 **unify** if $T_1\sigma$ and $T_2\sigma$ are identical for some substitution σ; σ is called a **unifier** of T_1 and T_2
- σ is called **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

$L = [a, b \mid X]$
- all the following substitutions are unifiers:

$\sigma = \{L \rightarrow [a, b \mid X1], X \rightarrow X1\}$
$\sigma_1 = \{L \rightarrow [a, b, c \mid X2], X \rightarrow [c \mid X2]\}$
$\sigma_2 = \{L \rightarrow [a, b, c, d \mid X3], X \rightarrow [c, d \mid X2]\}$

$\ldots$
σ is the most general unifier

9.8.2 Control algorithm revisited

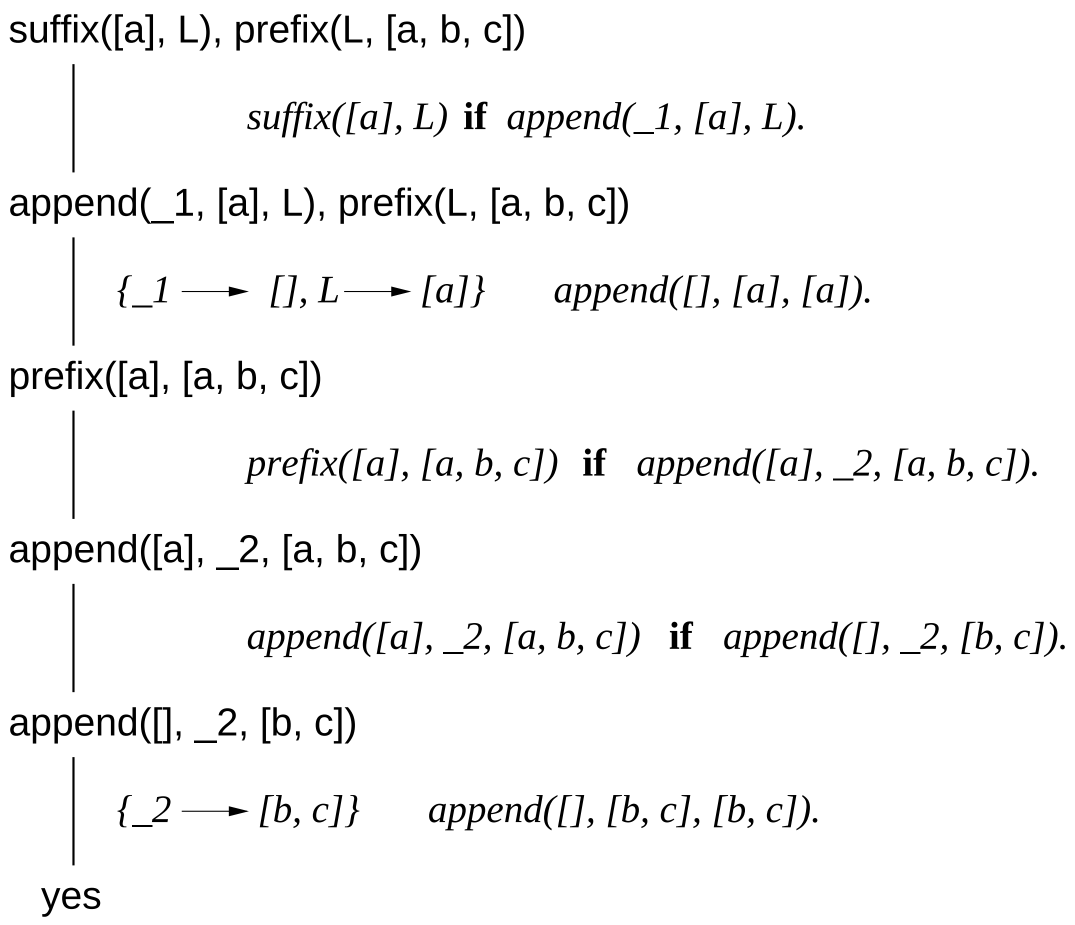
```
start with a query as the current goal;
while the current goal is nonempty do
   let the current goal be $G_1, G_2, \ldots, G_k$, where $k \geq 1$;
   choose the leftmost subgoal $G_1$;
   if a rule applies to $G_1$ then
      select the first such rule $A : -B_1, \ldots, B_j, j \geq 0$, not already used;
      let $\sigma$ 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
         fail
      else
         backtrack
      end if
   end if
end while;
succeed
```

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).
```

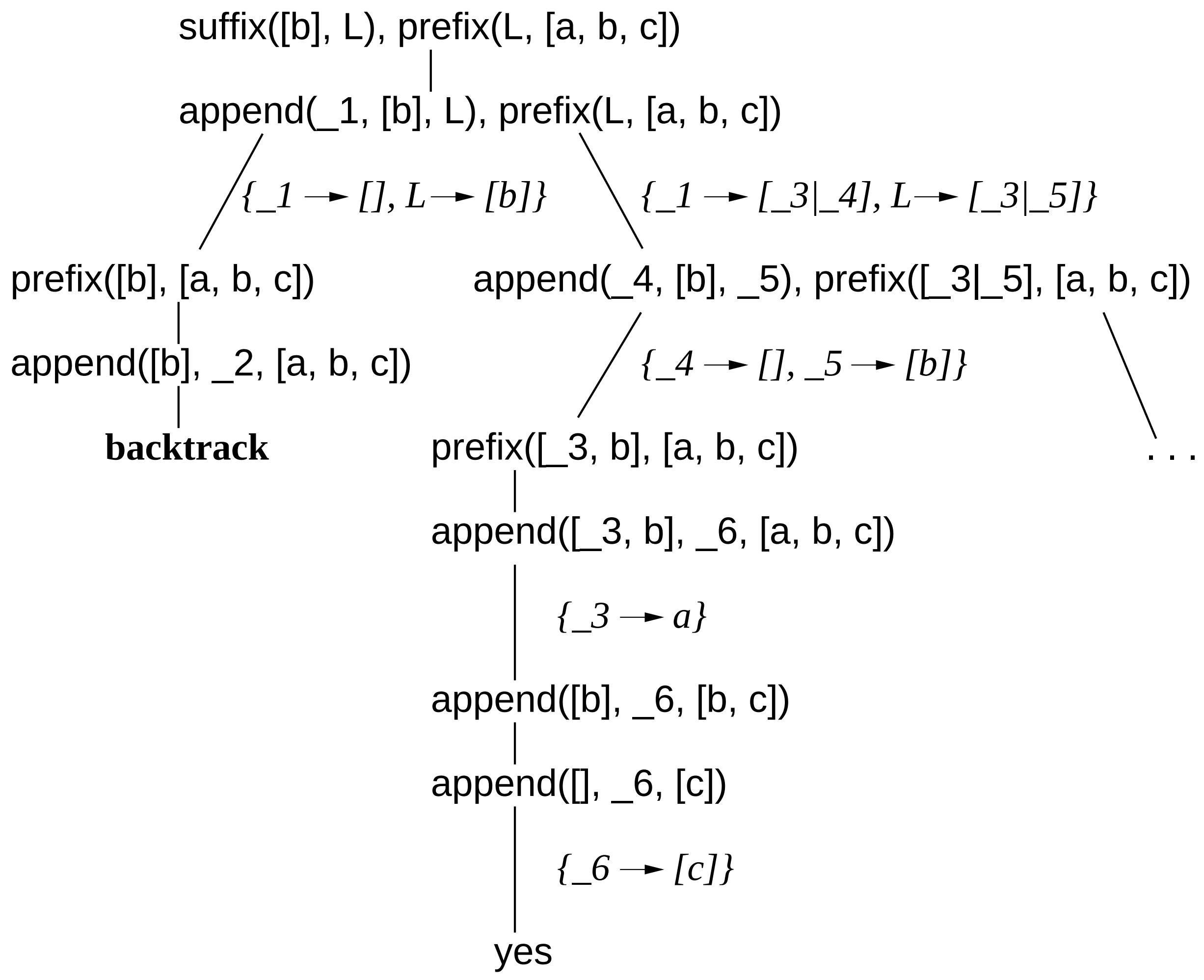
example - a computation which succeeds without backtracking

```
?- suffix([a], L), prefix(L, [a, b, c]).
    =>  L = [a]
    =>  yes
```


suffix([a], L), prefix(L, [a, b, c])
suffix([a], L) if append(_1, [a], L).
append(_1, [a], L), prefix(L, [a, b, c])
{_1 ⟶ [], L ⟶ [a]}
append([], [a], [a]).
prefix([a], [a, b, c])
prefix([a], [a, b, c]) if append([a], _2, [a, b, c]).
append([a], _2, [a, b, c])
append([a], _2, [a, b, c]) if append([], _2, [b, c]).
append([], _2, [b, c])
{_2 ⟶ [b, c]}
append([], [b, c], [b, c]).
yes

example - a computation which succeeds with backtracking

```
?- suffix([b], L), prefix(L, [a, b, c]).
    =>  L = [a, b]
    =>  yes
```



suffix([b], L), prefix(L, [a, b, c])
append(_1, [b], L), prefix(L, [a, b, c])
{_1 → [], L → [b]}
{_1 → [_3|_4], L → [_3|_5]}
prefix([b], [a, b, c])
append(_4, [b], _5), prefix([_3|_5], [a, b, c])
append([b], _2, [a, b, c])
{_4 → [], _5 → [b]}
backtrack
prefix([_3, b], [a, b, c])
. . .
append([_3, b], _6, [a, b, c])
{_3 → a}
append([b], _6, [b, c])
append([], _6, [c])
{_6 → [c]}
yes

9.8.3 Goal order

– affects the search for solutions

example

```
?- suffix([a], L), prefix(L, [a, b, c]).
    =>  L = [a] ;                   % asking for more solutions
        [infinite computation]
```

- the leftmost subgoal `suffix([a], L)` has infinitely many solutions, only the first of which satisfies the second subgoal `prefix(L, [a, b, c])`

```
?- suffix([a],L).
    =>  L = [a] ;
    =>  L = [_1, a] ;
    =>  L = [_1, _2, a] ;
    =>  L = [_1, _2, _3, a] ;
    =>  ...
```

- if we change the order of the two subgoals, we obtain a finite Prolog search tree

example (continued)

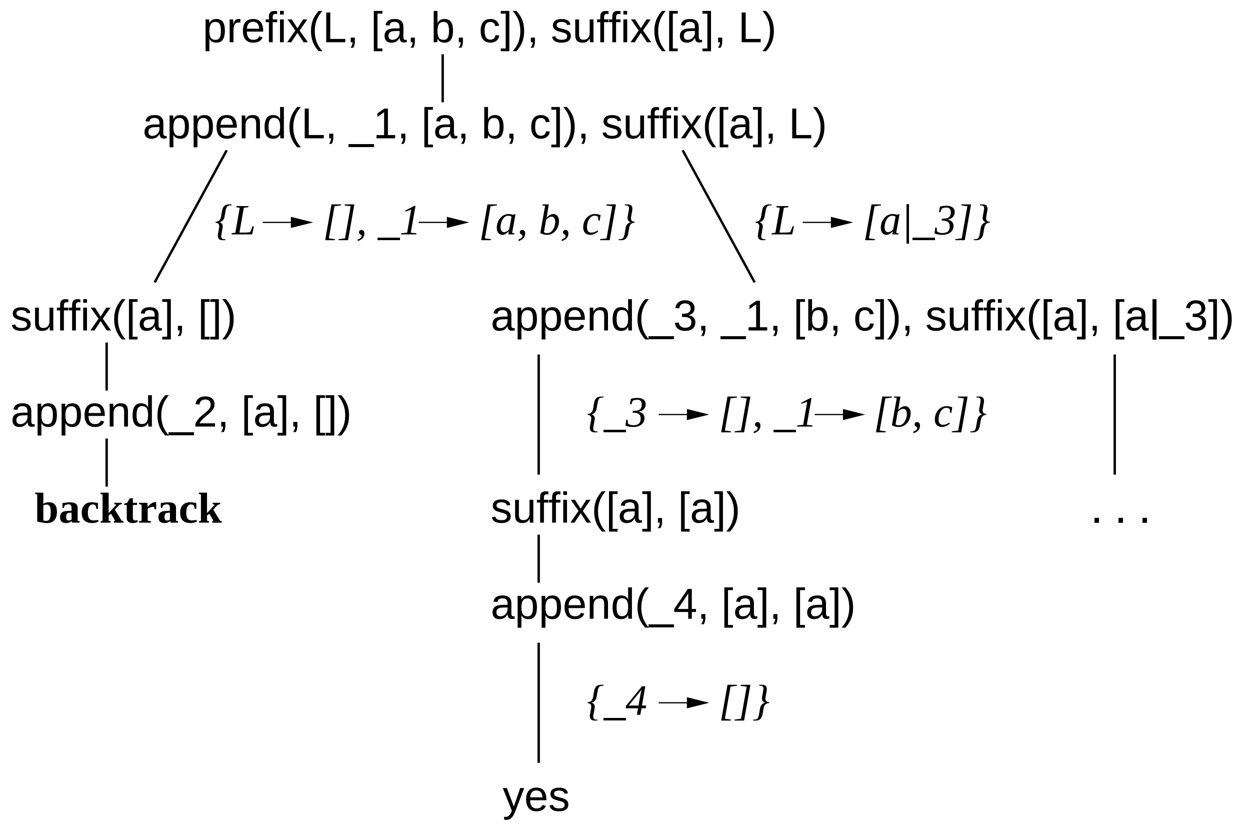
```
?- prefix(L, [a, b, c]), suffix([a], L).
    =>  L = [a] ;
    =>  no
```

- this is because the subgoal `prefix(L, [a, b, c])` has only finite many solutions

```
?- prefix(L, [a, b, c]).
    =>  L = [] ;
    =>  L = [a] ;
    =>  L = [a, b] ;
    =>  L = [a, b, c] ;
    =>  no
```

example – goal order changed (compare with page 354)

```
?- prefix(L, [a, b, c]), suffix([a], L).
    =>  L = [a] ;
    =>  no
```



9.8.4 Rule order

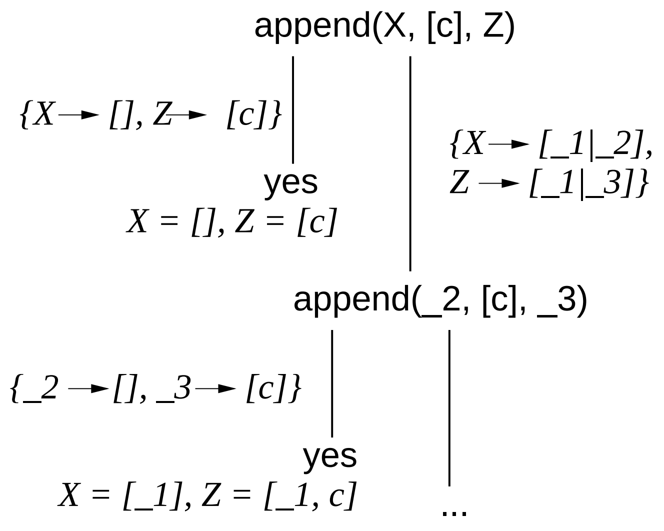
– determines the order of the children of a node in a Prolog search tree
  - changes the order in which solutions are reached

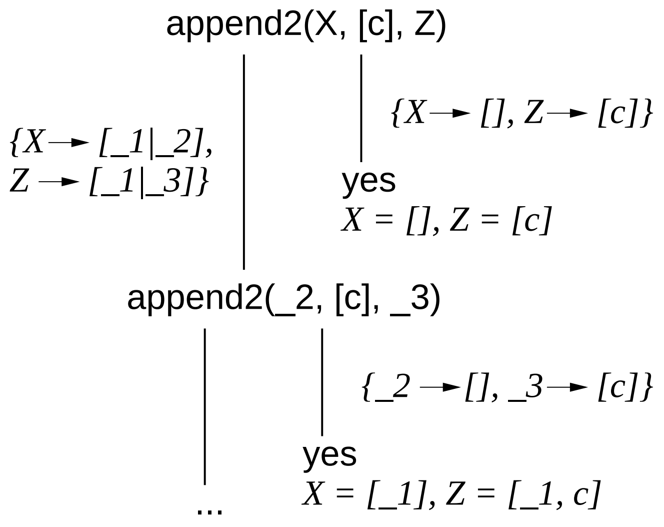
example – rule order changed

```
append([], Y, Y).
append([H|X], Y, [H|Z])
              :- append(X, Y, Z).

?- append(X, [c], Z).
    =>  X = []
        Z = [c] ;
    =>  X = [_1]
        Z = [_1, c] ;
    =>  X = [_1, _2]
        Z = [_1, _2, c] ;
    =>  X = [_1, _2, _3]
        Z = [_1, _2, _3, c] ;
```

```
append2([H|X], Y, [H|Z])
             :- append2(X, Y, Z).
append2([], Y, Y).

?- append2(X, [c], Z).
    =>  [infinite computation]





    =>  X = [_1, _2, _3, _4]
        Z = [_1, _2, _3, _4, c] ;
    =>  ...
```

example – rule order changed (continued) – the Prolog search trees





9.8.5 Occurs-check problem

- *occurs check* – whether a variable X appears in a term T before X and T are unified
- for efficiency, Prolog neglects to do these occurs checks
- in case X does occur in T, the unification leads to a nonterminating computation

example

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

?- append([], E, [a, b|E]).
    =>  E = [a, b, a, b, a, b, a, b, a|...] ;
    =>  no
```

- `append([], E, [a, b|E])` unifies with `append([], Y, Y)`
- `Y` must unify with both `E` and `[a, b|E]`
- we get then

```
            E = [a,b|E] = [a,b,a,b|E] = [a,b,a,b,a,b|E] = ...
```

9.9 Programming techniques

9.9.1 Guess and verify

- *guess-and-verify query* has the form:

$$\text{Is there } S \text{ such that } guess(S) \textbf{ and } verify(S)?$$

- solutions are generated for $guess(S)$ until one satisfying $verify(S)$ is found

- *guess-and-verify rule* has the form:

$$conclusion(\ldots) \quad \textbf{if} \quad guess(\ldots, S, \ldots) \quad \textbf{and} \quad verify(\ldots, S, \ldots)$$

example

```
member(M, [M|_]).
member(M, [_|Tail])
            :- member(M, Tail).


?- member(a, X), X = [1, 2 ,3].
        [infinite computation]
```

```
overlap(X, Y)
         :- member(M, X), member(M, Y).

?- overlap([a, b, c, d], [1, 2, c, d]).
    =>  yes
?- X = [1, 2 ,3], member(a, X).
    =>  no
```

9.9.2 Variables as placeholders

- variables
  - used in terms to simulate modifiable data structures
  - placeholders for subterms to be filled later
- **open list** – a list ending in a variable (not open = *closed*)

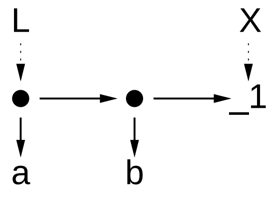
example `[a, b | X]`

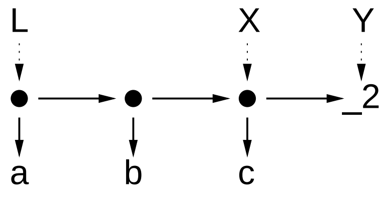
  - if `X` is unified with `[]`, the list represents `[a, b]`
  - if `X` is unified with `[c]`, the list represents `[a, b, c]`
- an open list can be modified by unifying its end marker

example

```
?- L = [a, b|X].
    =>  L = [a, b|_1]
        X = _1
    =>  Yes
```

```
?- L = [a, b|X], X = [c, Y].
    =>  L = [a, b, c, _2]
        X = [c, _2]
        Y = _2
    =>  yes
```





example – manipulating a queue

- queue – `q(L, E)`
  - `L` open list
  - `E` end marker representing some suffix of L
  - the contents of the queue `q(L, E)` are the elements of `L` that are not in `E`

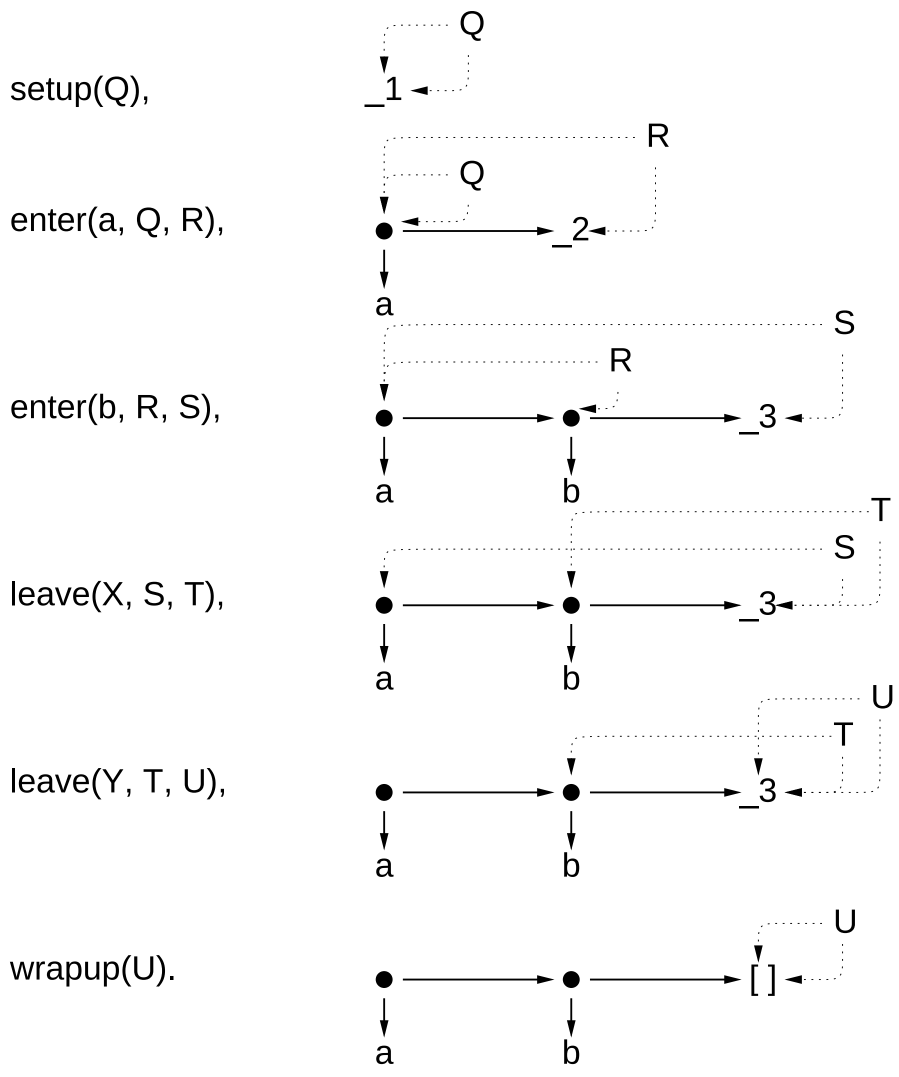
```
setup(q(X,X)).
enter(A, q(X, Y), q(X, Z)) :- Y = [A|Z].
leave(A, q(X, Z), q(Y, Z)) :- X = [A|Y].
wrapup(q([], [])).
```

```
?- setup(Q), enter(a, Q, R), enter(b, R, S), leave(X, S, T),
leave(Y, T, U), wrapup(U).
    =>  Q = q([a, b], [a, b])
        R = q([a, b], [b])
        S = q([a, b], [])
        X = a
        T = q([b], [])
        Y = b
        U = q([], [])
    =>  yes
```

- we take the subgoals one by one (see figure on page 366)

```
?- setup(Q).
    =>  Q = q(_1, _1)
    =>  yes
?- setup(Q), enter(a, Q, R).
    =>  Q = q([a|_2], [a|_2])
        R = q([a|_2], _2)
    =>  yes
?- setup(Q), enter(a, Q, R), enter(b, R, S).
    =>  Q = q([a, b|_3], [a, b|_3])
        R = q([a, b|_3], [b|_3])
        S = q([a, b|_3], _3)
    =>  yes
?- setup(Q), enter(a, Q, R), enter(b, R, S), leave(X, S, T).
    =>  Q = q([a, b|_4], [a, b|_4])
        R = q([a, b|_4], [b|_4])
        S = q([a, b|_4], _4)
        X = a
        T = q([b|_4], _4)
    =>  yes
```

```
?- setup(Q), enter(a, Q, R), enter(b, R, S), leave(X, S, T), leave(Y, T, U).
    =>  Q = q([a, b|_5], [a, b|_5])
        R = q([a, b|_5], [b|_5])
        S = q([a, b|_5], _5)
        X = a
        T = q([b|_5], _5)
        Y = b
        U = q(_5, _5)
    =>  yes
```



example (continued) – “deficit” queues – an element leaves before entering

```
?- setup(Q).
    =>  Q = q(_1, _1)
    =>  yes
?- setup(Q), leave(X, Q, R).
    =>  Q = q([_2|_3], [_2|_3])
        X = _2
        R = q(_3, [_2|_3])
    =>  yes
?- setup(Q), leave(X, Q, R), enter(a, R, S).
    =>  Q = q([a|_4], [a|_4])
        X = a
        R = q(_4, [a|_4])
        S = q(_4, _4)
    =>  yes
?- setup(Q), leave(X, Q, R), enter(a, R, S), wrapup(S).
    =>  Q = q([a], [a])
        X = a
        R = q([], [a])
        S = q([], [])
    =>  yes
```

9.9.3 Difference lists

- `dl(L, E)` – made up of two lists `L` and `E` where `E` unifies with a suffix of `L`
- the content of `dl(L, E)` – elements that in `L` but not in `E`
- can be open or closed but typically are open

<u>example</u>

```
append_dl(X,Y,Z) :- X=dl(L,M), Y=dl(M,N), Z=dl(L,N).
               % or append_dl(dl(L, M), dl(M, N), dl(L, N)).
?- append_dl(dl([a, b, c|E], E), dl([d, e|[]], []), L).
    =>  E = [d, e]
        L = dl([a, b, c, d, e], []) ;
    =>  no
?- append_dl(dl([a, b, c|E1], E1), dl([d, e|E2], E2), L).
    =>  E1 = [d, e|_1]
        E2 = _1
        L = dl([a, b, c, d, e|_1], _1) ;
    =>  no
```

```
contents(X, dl(L, E)) :- append(X, E, L).
?- contents([a, b], dl([a, b, c], [c])).
    =>  yes
?- contents([a, b], dl([a, b, c|E], [c|E])).
    =>  E = _1 ;
    =>  no
```

example (continued) – occurs-check problem again (see p. 361)

```
?- contents(X, dl([a, b|E], E)).
    =>  X = []
        E = [a, b, a, b, a, b, a, b, a|...] ;
    =>  X = [a]
        E = [b, b, b, b, b, b, b, b, b|...] ;
    =>  X = [a, b]
        E = _1 ;
    =>  X = [a, b, _2]
        E = [_2, _2, _2, _2, _2, _2, _2, _2, _2|...] ;
    =>  X = [a, b, _2, _3]
        E = [_2, _3, _2, _3, _2, _3, _2, _3, _2|...]
    .......
```

9.10 Cuts

- cut = !
- cuts can prevent backtracking, thus making a computation more efficient
- cuts can also be used to implement a form of negation which Horn clauses cannot do
- cuts are impure; they make Prolog depart further from logic (Prolog is already an aproximation, because pure logic is order independent but Prolog is not)

- preventing backtracking

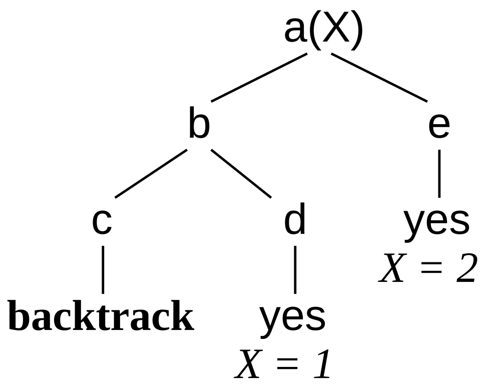
- the *general form* of a cut:

$$B :- C_1, \ldots, C_{j-1}, !, C_{j+1}, \ldots, C_k.$$

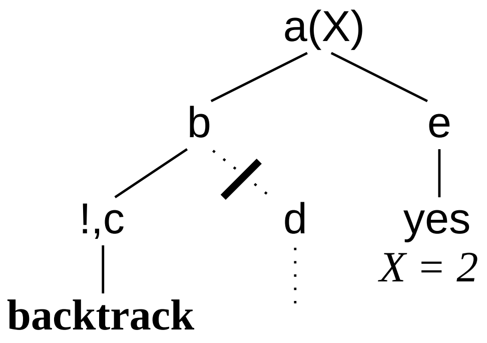
- *meaning*: the cut tells control to backtrack past $C_{j-1}, \ldots, C_1, B$ without considering any remaining rules for them

example 1 – a cut as the first condition, that is: $B :- !, C$.

```
a(1) :- b.
a(2) :- e.
b :- c.
b :- d.
d.
e.
?- a(X).
    =>  X = 1 ;
    =>  X = 2 ;
    =>  no
```

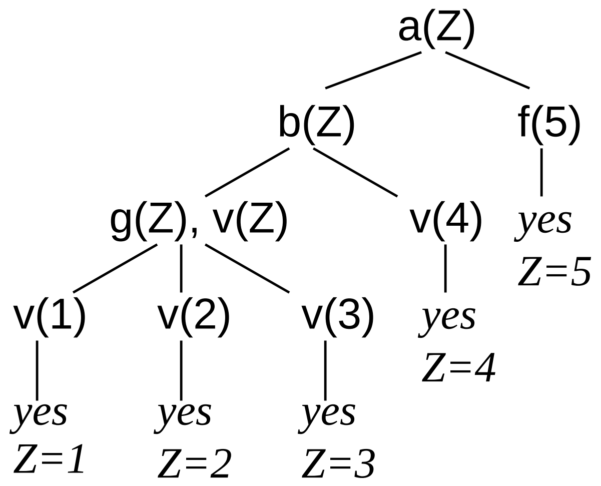


```
a(1) :- b.
a(2) :- e.
b :- !, c.   % a cut introduced here
b :- d.
d.
e.
?- a(X).
    =>  X = 2 ;
    =>  no
```



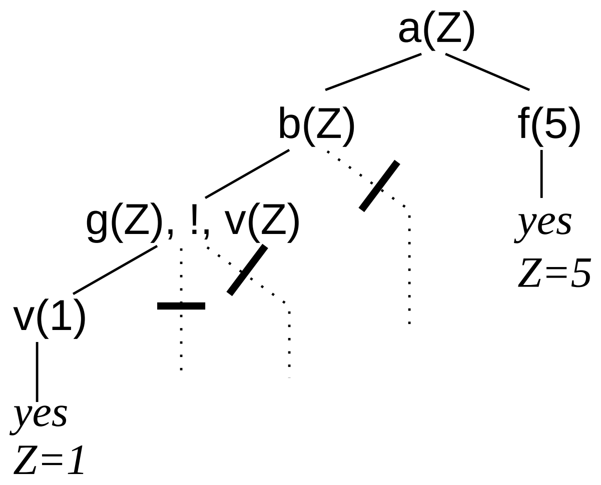
example 2 – a more general example of a cut

```
a(X) :- b(X).
a(X) :- f(X).
b(X) :- g(X), v(X).
b(X) :- X = 4, v(X).
g(1).
g(2).
g(3).
v(X).
f(5).

?- a(Z).
    =>  Z = 1 ;
    =>  Z = 2 ;
    =>  Z = 3 ;
    =>  Z = 4 ;
    =>  Z = 5 ;
    =>  no
```



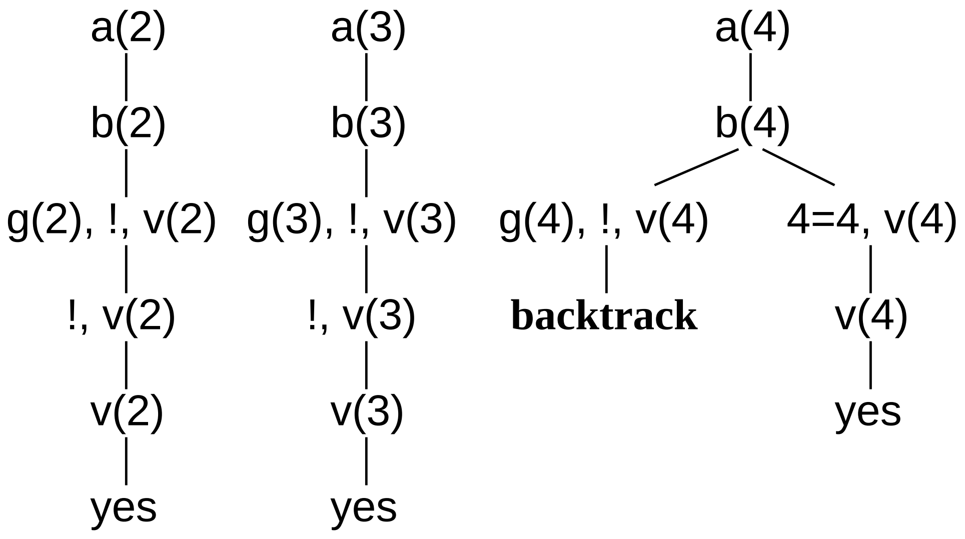
example (continued) – after the cut is introduced

```
a(X) :- b(X).
a(X) :- f(X).
b(X) :- g(X), !, v(X).          % a cut introduced here
b(X) :- X = 4, v(X).
g(1).
g(2).
g(3).
v(X).
f(5).
?- a(Z).
    =>  Z = 1 ;
    =>  Z = 5 ;
    =>  no
```



- still, all queries `a(2), a(3), a(4)` are satisfiable
- this is because their computations do not need backtracking, and thus the cut does not prevent them

```
?- a(2).
    =>  yes

?- a(3).
    =>  yes

?- a(4).
    =>  yes
```



9.10.1 Programming applications

- **green cuts** – used to prune parts of a Prolog search tree that cannot possibly reach a solution
- **red cuts** – not green
  - a green cut does not change the solutions of a program
  - a red cut does change the solutions

example 1 – green cuts

```
country(france, paris).
country(spain, madrid).
country(italy, rome).
country(canada, ottawa).
......
capital(X) :- country(_, X), !.
```

- each capital has a different name

example 2 – green cuts

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
```

- if the first clause is false, then the second is true; we may therefore write

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

example 3 – green cuts – binary search trees

```
member(K, node(K, _, _)).
member(K, node(N, S, _)) :- K < N, member(K, S).
member(K, node(N, _, T)) :- K > N, member(K, T).
```

- we can make the search more efficient by introducing a cut after `K < N`; if `K < N` but `member(K, S)` fails, then `K` is not in the tree and the cut saves the futile tests on `K > N`

```
member(K, node(K, _, _)).
member(K, node(N, S, _) :- K < N, !, member(K, S).  % cut!
member(K, node(N, _, T) :- K > N, member(K, T).
```

example 4 – red cuts

```
lookup(K, V, [(K, W)|_]) :- !, V = W.
                       % a cut here
lookup(K, V, [_|Z]) :- lookup(K, V, Z).
?- lookup(p, 72, D).  % enter information
    =>  D = [(p, 72)|_1] ;
    =>  no            % one solution only

?- lookup(p, 72, D), lookup(p, 73, D).
    =>  no  % two values for same key
                              not possible

?- lookup(l, 58, D), lookup(p, 72, D),
                           lookup(p, Y, D).
    =>  D = [(l, 58),  (p, 72)|_1]
        Y = 72 ;
    =>  no
```

```
install(K, V, [(K, W)|_]) :- V = W.
                         % no cut here
install(K, V, [_|Z]) :- install(K, V, Z).
?- install(p, 72, D).
    =>  D = [(p, 72)|_1] ;
    =>  D = [_2,  (p, 72)|_3] ;
    ...  % infinitely many solutions
?- install(p, 72, D), install(p, 73, D).
    =>  D = [(p, 72),  (p, 73)|_1] ;
    =>  D = [(p, 72), _2,  (p, 73)|_3] ;
    ...  % two values is okay
?- install(l, 58, D), install(p, 72, D),
                           install(p, Y, D).
    =>  D = [(l, 58),  (p, 72)|_1]
        Y = 72 ;
    =>  D = [(l, 58),  (p, 72),  (p, _2)|_3]
        Y = _2 ;
    ...
```

9.11 Negation as failure

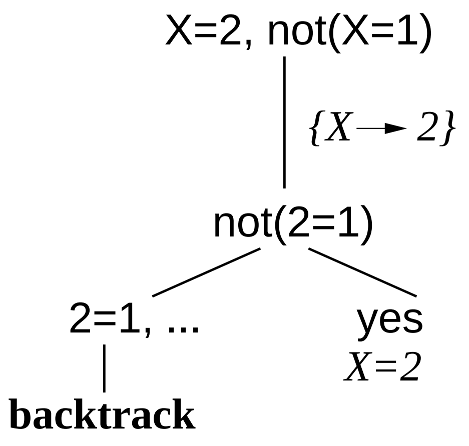
- the `not` operator is defined by

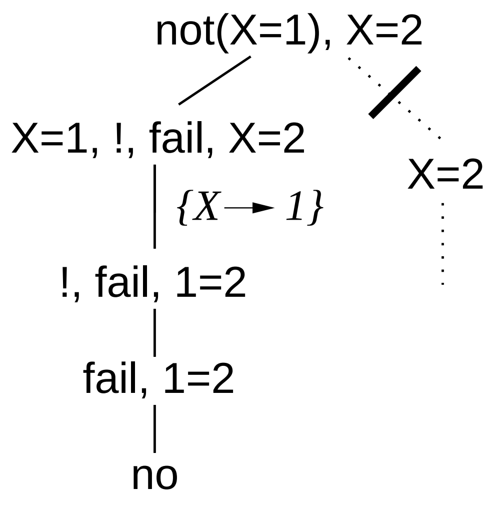
```
not(X) :- X, !, fail.
not(_).
```

- `fail` forces failure
- the first rule attempts to satisfy `X`
- if the goal `X` succeeds, then `!, fail` are reached and `fail` forces failure
- if `X` fails, then the second rule succeeds because `_` unifies with any term

example

```
?- X = 2, not(X = 1).                ?- not(X = 1), X = 2.
    =>  X = 2                              =>  no
    =>  yes
```





9.12 Example

Problem: Baker, Cooper, Fletcher, Miller, and Smith live in a five-story building. Baker doesn't live on the 5th floor and Cooper doesn't live on the first. Fletcher doesn't live on the top or the bottom floor, and he is not on a floor adjacent to Smith or Cooper. Miller lives on some floor above Cooper. Who lives on what floors?

```
floors([floor(_,5),floor(_,4),floor(_,3),floor(_,2),floor(_,1)]).

building(Floors) :- floors(Floors),
           member(floor(baker, B), Floors), B =\= 5,
           member(floor(cooper, C), Floors), C =\= 1,
             member(floor(fletcher, F), Floors), F =\= 1, F =\= 5,
             member(floor(miller, M), Floors), M > C,
             member(floor(smith, S), Floors),
             not(adjacent(S, F)),
             not(adjacent(F, C)),
             print_floors(Floors).

print_floors([A|B]) :- write(A), nl, print_floors(B).
print_floors([]).

member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y).

adjacent(X, Y) :- X =:= Y+1.
adjacent(X, Y) :- X =:= Y-1.
```

```
?- building(X).
    => floor(miller, 5)
   floor(fletcher, 4)
   floor(baker, 3)
   floor(cooper, 2)
   floor(smith, 1)

   X = [floor(miller, 5),
       floor(fletcher, 4),
       floor(baker, 3),
       floor(cooper, 2),
       floor(smith, 1)] ;
    => No
```

Contents