CS131: Programming Languages

Professor Eggert

Spring 2020

Contents

CS131: Programming Languages

Introduction

- sample problem:
  - *input*: ASCII text
  - *output*: all words (consecutive alphabetic characters) sorted by frequencies
- Knuth, a famous Turing-winning computer scientist, wanted to write a comprehensive book on programming:
  - created TeX, a language for typesetting math equations and code fragments
  - allows for creating a scolarly paper discussing a program or source file, eg. in C or Pascal
- however, the code (Pascal) does not always equal the documentation (TeX)
  - despite programmers' best interests to keep them up-to-date and consistent
- to solve this, Knuth created a *unified* file with the code and the documentation *interleaved*
  - new programming paradigm called **literate programming**
- have another program that split the unified source file into a `.tex` and `.pas` file:
  - run one way, the source file is compiled into a readable paper
  - the other way, the source file is compiled into a runnable program
- Knuth, in order to market his new paradigm, created a unified file that solves the above sample problem:
  - the associated scolarly paper was published
  - the Pascal solution uses *hashed tries*, is very fast, and features error checking through its compiler
- however, the editor of Knuth's paper brought up an *alternative* approach using **pipelines**:
  - `tr -c 'A-Za-z' '[\n]' | sort | uniq -c | sort -rn`
  - more readable, much shorter, and easier to write than the Pascal solution
- this is an issue of the **choice of notation / programming language** between the languages programmers use to implement solutions:
  - advantages and disadvantages of using full-fledged OOP languages vs. scripting solutions vs. other programming language types
- **Sapir-Whorf hypothesis** on the varieties and constraints of natural language:
  1. no limit on structural diversity

        - eg. some languages are not *recursive*, ie. they have a limit on their length
    2. the languages we use to some extent influence how we use the world
- the above ideas also apply to programming languages:
    - there is a great *diversity* in programming languages:
        - **imperative** languages like C focus on assignment and iteration
            - variables have current values, and order of execution is critical
        - **functional** languages like ML focus on recursion and single-valued variables
        - **logic programming** languages express programs in terms of rules about logical inference
        - **object-oriented** languages like Java focus on objects, ie. a bundle of data and methods
        - language types can overlap
    - programming languages will also *evolve* and change over time
- core of programming languages:
    - principles and limitations of programming **models**
    - **notations** ie. languages for these models (often textual)
    - methods to evaluate strengths and weaknesses of different notations in different contexts
        - eg. maintainability, reliability, training cost, program development
        - as well as execution overhead, licensing fees, build overhead, porting overhead
        - choice of notation can also be political, eg. a company preference for a language
- language design issues:
    - orthogonality of parts
        - eg. implementation of functions and selection of types should be independent
            - eg. C is not orthogonal in this sense since its functions cannot return an array type
    - efficiency
        - eg. CPU, RAM, energy, network access
    - simplicity, ie. ease of *implementation*
    - convenience, ie. ease of *usage*
        - eg. C provides many ways to increment a variable
    - safety
        - static, compile-time checking vs. dynamic, runtime checking
    - abstraction
        - a strength of object-oriented languages
    - exceptions
    - concurrency
    - evolution / mutability of a language

Software Construction

Traditional Approach

- conventional / traditional languages such as C/C++, Fortran follow a *staged* software construction process
- AKA the **software tools** approach:
    - each of the following stages is handled by a separate software tool
    - programs at each phase can be replaced with different ones
    - pioneered at Bell Labs as part of Unix

1. the file is translated from a series of literal character bytes into a string of **tokens** through **lexing**:
    - comments and whitespace are ignored
    - some of the tokens are associated with symbols, eg. functions such as `main` and `getchar`
    - a **lexeme** is the literal token with its associated programming language metadata
2. a **parse tree** is constructed from the string of tokens through **parsing**:
    - the root of the tree represents the entire program
    - eg. the children of a function in the tree would include its type, ID, subtrees for arguments and statements, as well as parentheses, brackets, and semicolons
    - eg. the children of a function call in the tree would include subtrees for the expression and parameters, as well as parentheses and semicolons
    - following the *fringes* or leaves of the tree recreates the string of tokens
    - different compilers lex and compile in one or two passes
3. the parse tree is *checked*
    - identifiers, types, names, etc.
    - gives a checked parse tree
4. *intermediate* code is generated from the parse tree
    - machine independent, convenient for the compiler writer
5. intermediate code is turned into assembly code
    - goes through process of optimization and machine code generation
6. assembly code (`.a`) is turned into object code (`.o`) by the assembler (`asm`)
7. object code (`.o`) is turned into an executable (`a.out`) by the linker (`ld`)
8. the executable is run by the loader

Integrated Development Environment

- **integrated development environment (IDE)** approach:
    - pioneered at Xerox PARC
    - eg. Eclipse, Emacs
    - eg. Smalltalk:
        - one large program
        - controls screen, keyboard, and mouse
        - written in Smalltalk
            - as you edit the program, modified program continues to run
            - editor also written in Smalltalk
        - implements your development environment
- in reality, there is a spectrum between these two approaches
    - eg. Emacs is written in C and compiled by gcc, and then bootstraps into an IDE

Compilers vs. Interpreters

- conventional model is that of a **compiler**:
    - translates source code to machine code
    - `i *= 2;` is translated into `shl %rax`
    - more difficult to debug after transformed into machine code
        - machine code can be executed out of order, contain condensed instructions, etc.
- another model is that of an **interpreter**:
    - leaves the program in a form *closer* to the original source
    - eg. simplified code with discarded comments, or checked parse tree, or machine-independent intermediate code
    - *pros*:
        - easier to debug since debugger can explain the the program to the developer
        - easier to write and maintain than compilers
            - can avoid machine code
        - easier to perform runtime checks such as subscript checking
    - *cons*:
        - interpreted code is slower
            - overhead of walking through a tree, and fetching and decoding instructions
- another *hybrid* approach:
    - eg. the **Java virtual machine (JVM)**:

  - `javac` translates `.java` files into `.class` bytecode
  - a Java interpreter reads in the `.class` file and can interpret it
  - however, the interpreter also contains a link to an executable that can translate bytecode to machine code in memory
  - interpreter creates a *histogram* of hot spots from profiling:
    - machine code is compiled *while* the interpreter runs, only for the most expensive hot spots in the bytecode
    - ie. use portable byte code, and then compile some it on the fly
  - this is a **just-in-time (JIT)** compiler
- popularized by Java, now used by Chromium, Firefox, etc. to run JavaScript
- **dynamic linking** is essential in this approach:
  - take a piece of a program from a file, and link it into your running program
  - allows for *self-modifying* code where programs call the dynamic linker to change the program itself
  - requires great care in embedding foreign, potentially malicious, code

Evolving Languages

- note that languages themselves will *evolve* over time:
  - C, C++, OCaml of today differs greatly from when the language was introduced
  - minor glitches will prevent compilation of 1970s C programs with modern day compilers
- evolving languages leads to issues with **compatibility**:
  - allowing for evolving languages without breaking existing software
  - eg. BASIC was developed on GE 225 computers
    - computers had very slow operations
    - original language had to grow, using extensions on the original language
  - eg. C was developed on the PDP 11 minicomputer
    - computers had much faster operations than BASIC computers
      - initially, assumptions were that accessing memory is cheap, while computation is expensive
    - nowadays, accessing RAM can be 100 times slower than adding numbers
      - operation speed assumption in older C programs no longer holds
      - old programs still run, but their performance no longer shines

- in the *future*:
  - new modern technologies, eg. GPU computing, distributed network and cloud computing
  - the languages of today might not be the best fit
  - ideally, fix these issues via language extensions
    - but in practice, this only goes so far
  - languages may not scale to larger problems:
    - machines are larger (eg. larger arrays, multiple computers)
    - more programmers (more complex programs)
    - what are some language-oriented ways of dealing with more complex programs?

Syntax and Grammar

- grammar is concerned with the lexing and parsing stages of translation:
  - the **syntax** of a programming language defines the form and structure of programs
    - ie. form independent of meaning
  - the **semantics** of a programming language dictates the behavior and meaning of programs
  - syntax without semantics: "Colorless green ideas sleep furiously."
  - semantics without syntax: "Ireland has leprechauns galore."
  - ambiguity of syntax and semantics: "Time flies."
- judging syntax:
  - inertia, ie. what people are used to
    - eg. `3 + 4 * 5` in C vs. `3 4 5 * +` in Forth vs. `(+ 3 (* 4 5))` in Lisp
  - simplicity and regularity
    - Lisp (regularly defined using parentheses) and Forth (no need for parentheses for precedence) win out
  - readability
    - form should reflect meaning
    - eg. `if (x > 0 && x < n)` vs. `if (0 < x && x < n)`
  - writability and conciseness
  - redundancy
    - avoiding silly mistakes
    - eg. in C, you explicitly match declaration to definition
  - unambiguity
- programming language **grammar** is a set of rules or definitions that describe how to build a **parse tree**:
  - the tree grows downward, where the children of each node follows the forms defined by the grammar
    - the language is the set of all possible strings formed as the *fringes* of the parse trees
    - **parsing** a language string finds its parse tree
    - an abbreviated, simplified parse tree is an **abstract syntax tree (AST)**
  - for an example grammar defining a simplified version of English:
    - `<noun-phrase> ::= <article> <noun>`
    - `<sentence> ::= <noun-phrase> <verb> <noun-phrase>`
  - for a simple language using expressions with three variables:
    - `<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | a | b | c`
    - this allows for expressions such as `a + b * c` and `((a+b) * c)`

        - a **recursive** grammar allowing for an *infinite* language
    - the language itself is a set of certain strings:
        - a sentence is a member of that set
        - a string is a finite sequence of tokens, with a corresponding parse tree

Backus-Naur Form

- **Backus-Naur form (BNF)** can be used to explicitly describe **context-free grammars (CFGs)**
- parts of grammar specified in BNF:
    - the **terminal symbols** or **tokens** are the smallest units of syntax (leaves of parse tree):
        - whitespace and comments are not tokens
        - includes identifiers, numbers, operators, **keywords** that are part of the language
            - certain keywords are **reserved words** that cannot be treated as identifiers
        - eg. strings and symbols, `if` , `≠`
    - the **non-terminal symbols** are the different kind of language constructs (interior nodes of parse tree):
        - listed in angle brackets
        - eg. sentences, statements, expressions
        - `<empty>` is a special non-terminal symbol in BNF
    - the **start symbol** is the non-terminal symbol at the root of the parse tree
    - a set of **productions** or **rules**:
        - a production consists of a left-hand side, the separator `::=` , and a right-hand side
            - left-hand side is a single non-terminal symbol
            - right-hand side is a sequence of tokens or non-terminal symbols
- other extended BNF metasymbols include `[ ]` for optional expressions, `{ }` for repeated expressions, etc.
    - uses `|` or `/` notation for multiple definitions
    - EBNF is a kind of syntactic sugar for BNF, doesn't extend the set of definable languages
- can alternatively use **syntax diagrams** (directional graphs) to express grammars

- many different variations for EBNF, have been consolidated into the standardized **ISO EBNF**:
  - `A = BC;` grammar rule
  - `"terminal symbol" 'terminal symbol'`
  - `[optional]`
  - `{repeat 1 or more}`
  - `(grouping)`
  - `(*comment*)`
  - operators:
    - `*` repetition
    - `A-B` set difference, ie. “A except B”
    - `A,B` explicit concatenation, ie. “A followed by B”
    - `A|B` disjunction, ie. “A or B”
  - eg. defining part of ISO EBNF using ISO EBNF:
    - `syntax = syntax rule, {syntax rule};`
    - `syntax rule = meta id, '=', defns list, ';' ;`
    - `defns list = defn, {'|', defns};`
    - `defn = term, {',', terms};`
    - `term = factor,  ['-', exception];`
    - `exception = factor;`
    - `factor = [integer, '*'], primary;`

- some programming languages do not purely use CFGs:
  - Fortran, predates CFGs
  - typedefs in C allow for changing token types
  - indentation rules in Python
    - whitespace becomes important, affects parsing

- writing a grammar is similar to writing a program:
  - *divide and conquer* the problem
  - eg. making a BNF grammar for Java variable declarations:
    - eg. `int a=1, b, c=1+2;`
    - `<var-dec> ::= <type-name> <declarator-list> ;`
    - `<type-name> ::= boolean | byte | short | int | long | char | float | double`
    - `<declarator-list> ::= <declarator> | <declarator> , <declarator-list>`
    - `<declarator> ::= <variable-name> | <variable-name> = <expr>`
    - ignores array declarations

- the previous examples do not consider tokens as individual characters:
  - instead, they defined the **phrase structure** by showing how to construct

    parse trees
  - they do not define the **lexical structure** by showing how to divide program text into these tokens:
    - languages can have a **fixed-format** lexical structure where columns in lines and end-of-line markers are significant to the interpretation of the language
    - or a **free-format** lexical structure where end-of-line markers are simply whitespace

- illustrates the two distinct parts of syntax definition:
  - the **scanner** or **lexer** scans the input file and converts it to a stream of tokens without whitespace and comments
    - note that the lexer usually scans *greedily*
      - eg. `a-----b` is interpreted as the syntactically incorrect `((a--)--) - b` instead of the syntactically correct `(a--) - (--b)`
  - the **parser** then reads the tokens and forms the parse tree

Alternative Grammar Notations

- there are numerous different alternate syntaxes ie. notations that have been used for expressing grammars

- eg. **regular expressions**:
  - RegEx is powerful and more compact than other syntaxes, but loses power for more complex grammars
  - eg. `ab*` expresses the grammar `<S> ::= <S> b | a`
  - eg. but `(*a)*` fails to correctly express the grammar `<S> ::= ( <S> ) | a`
  - difficult to express *nested recursion* with RegEx

- eg. the notation for grammars used in Internet protocol RFCs:
  - uses forms of RegEx when convenient, otherwise it falls back to regular grammar rules
  - specifically, for the Message-ID in emails:
    - eg. `Message-ID: <eggert."93-542-27"@cs.ucla.edu>`
  - the grammar is expressed as follows:
    - `msgid = "<" dot-atom-text "@" id-right ">"`
    - `id-right = dot-atom-text / no-fold-literal`
    - `no-fold-literal = "[" *dtext "]"`
      - `*` is an example of EBNF, indicates 0 or more occurences

      - could rewrite in pure BNF
    - `dot-atom-text = 1*atext *("." 1*atext)`
      - `1*` indicates 1 or more occurences
    - `dtext = %d33-90 / %d94-126` (printable, except for “[]\”)
      - `%d33-90` represents the characters matching the ASCII numerical codes
    - `atext = ALPHA/DIGIT/"!"/"#"...` (subset of dtext)
    - note that all elements of this grammar is left or right recursed, so it can also be expressed entirely in RegEx
  - rewriting EBNF as BNF:
    - EBNF: `no-fold-literal = "[" *dtext "[]"`
    - BNF: `no-fold-literal = "[" dtexts "]"` , `dtexts = {empty}` , `dtexts = dtexts dtexts`
      - the empty rule allows the BNF version to terminate, otherwise it would be infinite

- eg. **syntax diagrams** ie. charts as opposed to textual representations:

  - not very useful for smaller grammars
    - but used often for more complex grammars
  - eg. used with SQL extensions
  - eg. example with Scheme:
    - `<cond> ::= ( cond <condclause>+ )|(cond <condclause>* ( else <sequence> ))`
    - some repetition in the textual representation
      - instead, can draw as a directed diagram
      - avoid repetition with loops
  - diagrams are also helpful in seeing how to write a parser
    - diagrams act as a kind of **push-down automata** (ie. state machine with stack)
      - state machines on their own cannot handle recursion

Syntax and Semantics

- some types of *basic* grammar errors:

  - nonerminal used but not defined:
    - `<S> ::= <S> a | <B> c | d`
    - `<B>` is used but not defined, so that specific rule can never be applied
  - nonterminal defined but not used:
    - `<S> ::= <S> a | d` , `<B> ::= a <S>`
    - `<B>` can never be applied

  - more examples of *useless* rules:
    - `<S> ::= <S> a | b` , `<C> ::= <C> d`
    - `<C>` can still never be applied
  - grammar doesn't capture some required *constraint*:
    - eg. for a basic sentence (S) using noun phrases (NP), verb phrases (VP), etc.
      - `<S> ::= <NP> <VP> .`
      - `<NP> ::= <N> | <Adj> <NP>`
      - `<VP> ::= <V> | <VP> <Adv>`
    - eg. "blue dogs bark loudly" vs. "dog bark"
      - singular plural agreement is broken
    - have to introduce additional complexity in the grammar for singular phrases vs. plural phrases:
      - `<S> ::= <SNP> <SVP> . | <PNP> <PVP> .`
      - `<SNP> ::= <SN> | <Adj> <SNP>`
      - `<PNP> ::= <PN> | <Adj> <PNP>`
      - `<SVP> ::= <SV> | <SVP> <Adv>`
      - `<PVP> ::= <PV> | <PVP> <Adv>`
      - plural phrases can only use plural nouns with plural verbs, etc.
    - such a fix *doubles* the grammar size for each additional attribute of complexity
      - thus should use grammars *appropriately*, eg. for capturing balanced parentheses or an appropriate level of nesting
      - as opposed to using them for type-checking or name-checking
  - grammars could also act at a lower level and consider tokens as single characters:
    - *overkill* to specify such character rules as grammar
      - would have to specify whitespace and comments in the grammar
    - instead, use separate lexer/tokenizers and consider tokens at a *higher level*
      - greatly simplifies the grammar

- with **ambiguity**, different grammars may generate the *same* language ie. they create parse trees with identical fringes:
  - however, the internal *structures* of the parse trees may be very different
  - eg. `<subexp> ::= <var> - <subexp> | <var>`
    - vs. `<subexp> ::= <subexp> - <var> | <var>`
    - both grammars could create the language `a - b - c` , but represent different computations and results
      - `a - (b - c)` vs. `(a - b) - c`
  - thus, when considering semantics, the semantics represented by unique

parse trees must be *unambigious*

- consider the following grammar which has issues with precedence and associativity:
    - `<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> ) | a | b | c`
- dealing with **precedence**:
    - the grammar can generate different parse trees for `a + b * c`, including one where addition has higher precedence than multiplication, ie. `(a + b) * c`
    - the grammar must be modified to eliminate this erroneous tree:
        - `<exp> ::= <exp> + <exp> | <mulexp>`
        - `<mulexp> ::= <mulexp> * <mulexp> | ( <exp> ) | a | b | c`
        - essentially, does not allow lower-precedence operators to occur in the subtrees of higher-precedence ones, unless explicitly parenthesized
            - creates a level of precedence for multiplication
- dealing with **associativity**:
    - with subtraction instead of addition, the grammar can generate different parse trees for `a - b - c`
        - even with addition, `a + b + c` can generate *different* answers due to floating point addition rounding depending on associativity of the parse tree
    - the grammar must only generate one parse tree for each expression
    - without parenthesis, most languages are **left-associative** and choose the `(a - b) - c` tree
        - examples of a right-associative operators are the assignment operator `=` and construct operator
    - the grammar must be modified, by adding additional complexity with another nonterminal:
        - `<exp> ::= <exp> + <mulexp> | <mulexp>`
        - `<mulexp> ::= <mulexp> * <rootexp> | <rootexp>`
        - `<rootexp> ::= ( <exp> ) | a | b | c`
        - the productions are only recursive on *one* side of each operator
        - essentially, does not allow left-associative operators to appear in the parse tree as the right child of another operator at the same level of precedence, without parentheses
            - forces trees to grow down to the left
- dealing with other ambiguities, eg. the **dangling else** problem:
    - for if-statements with an optional `else`, multiple parse trees may be

    generated for the statement `if e1 then if e2 then s1 else s2`
  - could group as `if e1 then (if e1 then s1) else s2`
    - or `if e1 then (if e2 then s1 else s2)`
  - most languages attach the `else` with the nearest unmatched `if`
  - the grammar must be modified:
    - add a new non-terminal symbol for the full if-else-statement
    - substitute the new symbol within the grammar:
      - `<if-stmt> ::= if <expr> then <full-stmt> else <stmt>`
    - grammar can only match an `else` with an `if` if all of the nearer `if` parts are already matched

- dealing with more complex examples of ambiguity:

- eg. considering the following subset of grammar in the C standard:
  - `<stmt> ::= ; | break ; | continue ; | return ;`
  - `<stmt> ::= return <expr> ; | <expr> ; | { <stmts> }`
  - `<stmt> ::= while ( <expr> ) <stmt> | do <stmt> while ( <expr> ) ;`
  - `<stmt> ::= if ( <expr> ) <stmt> | if ( <expr> ) <stmt> else <stmt>`
  - a trailing `;` is excluded from some rules (eg. `while` or `if`) since it belongs to the statement within, not the overal nonterminal construct

- why are all the parentheses used (or not used)?
  - eg. use `while <expr> <stmt>` instead of `while ( <expr> ) <stmt>`
    - more generous and simpler grammar rule
      - seems cleaner and easier to understand
    - `while i < n i *= 3;` has no ambiguity
    - `while i * p == 3;` *does* introduce ambiguity
      - `while (i) (*p == 3;)` vs. `while (i*p == 3) ;`
      - empty statements in C make it easy for ambiguity to occur without the parentheses
  - eg. *conversely*, use `return ( <expr> ) ;` instead of `return <expr> ;`
    - some programmers use this specific style
    - there is no possible ambiguity even without parentheses
      - once parser reaches `;` , knows it is the end of the expression
    - parentheses are removed in order to simplify the grammar
  - eg. alternately, use `do <stmt> while <expr> ;`
    - instead of `do <stmt> while ( <expr> ) ;`
    - again, the `;` indicates the end of the expression
    - this simplification thus *does not* introduce ambiguity
    - in C, these `do-while` parentheses are there for consistency

- there is another ambiguity in the grammar:

  - previously mentioned dangling else problem
  - `<stmt> ::= if ( <expr> ) <stmt> else <stmt>` is too *generous*
    - if we want to pair with nearest unpaired else, this rule cannot be at the *top* level
  - to fix, complicate the grammar and add another new nonterminal
    - `<stmt1>` is just like `<stmt>`, except that it doesn't allow the elseless `if`
    - reorganize the grammar as follows:
      - `<stmt1> ::= ...` all previous `<stmt>` rules except the elseless `if`
      - `<stmt> ::= if ( <expr> ) <stmt1> else <stmt> | if ( <expr> ) <stmt>`
- however, by adding complexity to fix ambigious grammars, the parse tree becomes more convoluted, with extra nonterminals and rules:
  - the corresponding parse tree is named a **concrete syntax tree** vs. the original abbreviated **abstract syntax tree (AST)**
    - AST corresponds to ambiguous grammars where the *compiler* always does the "correct" parse
    - AST is simpler, and takes less memory
    - may be preferable to work with the AST
  - eg. Prolog is an example of avoiding the concrete tree due to complexities:
    - allows users to specify new operators along with their precedence and associativity
    - `op(700, xfx [=, ==, >=, ...])` defines non-associative binary operators
      - `a == b == c` is a syntax error Prolog, but not in C
    - `op(500, yfx, [+, -])` defines left-associative binary operators
    - `op(400, yfx, [*, /])` defines more left-associative binary operators with a *higher* precedence
    - `op(200, xfy, [**])` defines a right-associative binary operator
      - `a**b**c` is parsed as `a**(b**c)`
    - `op(200, fy, [+, -])` defines right-associative unary operators
      - `a**-b` is parsed as `a**(-b)` and `-b**a` is parsed as `-(b**a)`
  - thus there are no grammar rules for precedence in Prolog
    - instead, precedence is determined at runtime, depending on user defined operators

Note that ambiguity is commonly an issue with expressions, while statements can lead to different kind of issues:

```
int g(void) {
  return (a = 1) + (a = 2); // statement has undefined behavior!
  // runtime problem that has nothing to do with syntax,
  // stems from competing side effects within expressions
}
```

Semantics

- the **semantics** of a program address what a program *means*:
  - syntax is easier
    - specified through BNF, EBNF, etc.
  - *static* semantics means you can tease out meaning before program runs
    - specified through attribute grammars
  - *dynamic* semantics requires running the program to figure it out:
    - for *operational* semantics, describe how to write an imperative interpreter for the program (imperative programming)
    - for *axiomatic* semantics, describe the logic you can use to reason about the program (logic programming)
    - for *denotational* semantics, define a function from a program to its meaning (functional programming)
- **attribute grammars** are an extension of syntax grammars to capture some semantics:
  - each node in a parse tree (eg. generated via BNF) has some extra information, or **attributes**, that tell you more about the meaning of the program than a simple parse tree
  - eg. in a C program parse tree, attributes may include:
    - *type* attribute for the data type of the corresponding expression
      - can be computed from child node types, or inherited from a parent node type
    - *symtab* attribute for the data type of name-value bindings

Attribute grammar example:

```
a = b + c;

<expr> ::= <id> = <plus_expr>
<plus_expr> ::= <id1> + <id2>

% type is an *attribute* of the grammar
type(<expr>) = type(<plus_expr>)
type(<plus_expr>) =
  if type(<id1>) = int && type(<id2>) = int
    then int
  else float
```

- **operational** semantics:
  - to define a language M, write an interpreter for it in language L
  - eg. to define Python, write an interpreter for it in C
    - if you know C, and want to know what a Python program means,

        run the interpreter
        - the behavior *is* the meaning
  - eg. the first operational semantics for Lisp provided a Lisp interpreter, *written in* Lisp
    - philosophically *bad*, but still OK in some sense, eg. English dictionary written in English

Defining a subset of ML in Prolog:

```
% meaning m, expressions E, context C, value V
m(E1+E2, C, Vsum) :- % context C = [x=10, y=20, z=-19]
  m(E1, C, V1),
  m(E2, C, V2),
  Vsum is V1+V2.     % *different* use of + as operation instead of symbol

m(K, C, K) :- integer(K).

m(Var, C, Val) :- member(Var=Val, C).

% let X = E1 in E2
m(let(X,E1,E2), C, Val) :-
  m(E1, C, V1),
  m(E2, [X=V1 | C], Val). % evaluate E2 in *new* context

% fun x -> E
m(fun(X,E), C, lambda(X,E)). % function is just a construction

% E1 E2
m(call(E1,E2), C, Result) :-
  m(E1, C, lambda(Xf, Ef)), % Xf function param, Ef function body
  m(E2, C, V2),             % E2 argument expression
  m(Ef, [Xf=V2 | C], Result).

% these example semantics use dynamic scoping, for static scoping,
% something more complicated is needed
```

Identifiers

- names or identifiers are completely *arbitrary*
  - however, in software engineering, naming conventions are very important
  - in programs there are two main aspects of identifiers:
    - binding and binding times
    - scope
- there are some immediate simple issues for identifiers:
  - defining syntax for an identifier
    - trivial, usually not even part of the grammar
    - identifiers are given as tokens
  - supported characters in names
    - eg. in C/C++, names have the format `[a-zA-Z_][a-zA-Z_0-9]+`
- considerations for supported characters:
  - spaces in names
    - eg. in Fortran, spaces are supported in names
    - a bad idea, caused some rocket crashes
  - other alphabetic characters, eg. Unicode
    - eg. supported in C11
    - supports programmers in other languages
  - characters from alphabets (Unicode or not) that look very similar
    - eg. Cyrillic “O” vs. ASCII “O”, ASCII `"l"` vs. `"I"` vs. `"1"`
    - software reliability can suffer if identifiers are generalized too far
  - case sensitivity
    - eg. C is case sensitive for identifiers but not for numbers, `1e16 == 1E16`
    - eg. domain names on the Internet are not case sensitive
  - reserved words
    - a problem as languages evolve
      - difficult to add reserved words to programming languages
    - eg. `class` as an identifier in C vs. in C++
    - eg. C/C++ names beginning with `_` are all reserved

Binding and Binding Time

- a **binding** is an association between a name and a value
- a **symbol table** is a set of bindings
  - a partial function, no two names should be the same
- consider the bindings behind the statement `int a = 10` :

    - at a simple level, binds `a` to `10`
    - however, `a` has other properties, so other bindings are also occuring:
        - binds address of `a` to address of some piece of storage
        - binds type of `a` to `int`
- the statement `a = 15` *rebinds* the value of `a` but does not change its type or address:
    - occurs during execution
    - in functional languages, using this type of binding is not recommended
- illustrating the distinction between bindings:
    - `int *p = &a` vs. `int *p = &10`
        - not equivalent
    - `int b = sizeof a` vs. `int b = sizeof 10`
        - equivalent *only* because they have the same type
    - checking big-endianness vs. little-endianness
        - determined by the specific *machine* as well as the type
- **binding time** is the time during compilation and/or execution when a name becomes bound to a value:
    - *global* variables are bound when you wrote the program
        - ie. *program authorship* time
    - the representation of `int` became bound to 4-byte little endian when the ABI for x86-64 was decided
    - address of `a` became bound to some address `0x...` when the program was linked
    - but for *local* variables that will sit on the stack, their addresses become bound when the function is entered via a procedure call, during execution
- **explicit** vs. **implicit** binding:
    - explicit binding is written explicitly in the program:
        - `int a = 12` is explicit
    - implicit binding is implied by something else written:
        - in the preprocessor directives `#define FOO 27` followed by `#ifdef FOO` :
            - `FOO` is implicitly defaulted by the preprocessor to the value 0
        - OCaml uses type inference, a type of implicit binding
            - there is enough redundancy in OCaml that this isn't an issue
            - difficult to easily cause an issue with OCaml

Scoping

- **scope** allows for *information hiding* for names:
    - a scope of an identifier is the set of program locations where the identi-

  fier is visible
    - alternatively, the scope of an identifier is really the scope of the *definition* of the identifier
  - whenever a language allows different variables to have the same name, it must have some scoping rules to ensure unambiguity
    - ie. rules that bind each occurence of a name with exactly one variable, *without* ambiguity
  - identifier scoping can be achieved through different techniques, both *static* or *lexical* rules that are decided before runtime, as well as *dynamic* rules that are decided only at runtime

1. nesting with **blocks**:
   - eg. a pattern matching expression in OCaml introduces multiple new blocks
   - eg. C/C++ collects statements together using braces
   - scope of a definition is its block, minus the scopes of any redefinitions in interior blocks
     - ie. names can be redefined in an inner, nested block
     - the "minus" part of the scope definition is also known as the **scope hole**
2. **labeled namespaces** ie. a construct with definitions and a name that can be used to access the definitions from outside the construct:
   - eg. structures in OCaml, namespaces in C++, classes in general object-oriented languages
   - reduces namespace *pollution*, eg. can have multiple relevant `max` or `min` variables in unrelated classes
     - scopes should be as small and compact as possible
3. label each name as to how *public* it is going to be:
   - eg. Java keywords:
     - `public` : name is visible to another class in the same package, subclass outside the package, everywhere else
     - `protected` : name is visible to another class in the same package, subclass outside package
     - `(default)` : name is visible to another class in the same package
     - `private` : name isn't visible anywhere except inside the class
   - a bit of a *straightjacket*, cannot control visibility as much as you'd like
   - abstract names and types are those whose implementation is concealed, eg. `private` variables
4. have a *separate* section of program, giving package's API:
   - eg. interfaces in Java, signatures in OCaml
     - OCaml has structures with the actual implementation, as well as signatures that act as APIs
     - OCaml also allows for **functors** that are compile-time functions

            from structures to structures
            * allows for *editing* of source code so that it can conform to different signatures, control visibility, etc.
            * more flexible, but also more complicated

5. additionally, **primitive namespaces** that are not explicitly created using the language, but are instead part of the language definition:
    - eg. primitive types such as `int` have a certain *fragmented* scope where they are considered as types rather than variable names

Using many primitive namespaces in Java:

```java
class Reuse {
  Reuse Reuse(Reuse Reuse) {
    Reuse:
      while(true) {
        if (Reuse.Reuse(Reuse) == Reuse)
          break Reuse;
      }
      return Reuse;
  }
}
```

- all the above techniques are examples of static scoping rules, which contrasts **dynamic scoping**:
    - in dynamic scoping, each function has an *environment* of definitions
    - if a name occurs in a function that is not found in that function's environment
        * the environment of the function's *caller* is searched, etc.
    - scope of a definition is its containing function, along with any functions that are called (even indirectly) within that scope, minus the scopes of any redefinitions of the same name in those called functions
    - used in only a few language, eg. dialects of Lisp, APL
    - *pros*:
        * more power and flexibility
    - *cons*:
        * difficulty with efficient implementations
        * tendency towards large and complicated scopes

Static vs. dynamic scope in ML:

```sml
fun g x =
  let
    val inc = 1;
    fun f y = y + inc;
    fun h z =
   let
    val inc = 2;
   in
    f z
   end;
 in
  h x
 end;

g 5; (* with *static* scoping gives 6 *)
g 5; (* with *dynamic* scoping would have given 7 *)
```

- the underlying concern with scope is **separate compilation**:
  - compile different parts of the program at different times
    - eg. just one module if no other modules change
  - in C/C++, this is possible because there are two different kinds of definitions:
    1. a *full* definition
    2. a **declaration** ie. definition that gives only the name and type
       - the linker will treat a declaration as a reference to some other definition, possibly in another file

Types

- types are present in almost every programming language:
    - only languages where all operations are meaningful on all values can be considered free of types
        - eg. BCPL, an ancestor of Java
    - there are multiple definitions for types
- a **type** is a set of values or a set of objects in object-oriented languages:
    - language can specify sets of values and call them types
    - are these sets mutually exclusive, ie. does each value belong to exactly one type?
        - false for object-oriented languages through *inheritance*
        - true for other languages
- a **type** is a way to represent a set of values in a machine:
    - *primitive* types such as `int` , `char` , `float` , etc.
        - types that a program can use but not define for itself
    - *constructed* types such as `struct complex {double re, im;};`
        - types that programs can define for themselves
        - the layout of the new type is defined by the programmer
- a **type** is a set of values along with a set of operations on those values:
    - operations allow for observing properties of values and perform actions on the objects
    - an object-oriented definition
- **abstract** vs. **concrete** types:
    - abstract types hide the implementation:
        - found only via operations on the object
        - abstraction allows for *modularization*
        - eg. `float` is semi abstract in C, since where the S, E, and F fields lie is machine-dependent and not available by the C API
    - concrete or exposed types show the implementation to the programmer:
        - gives low-level access that gain *efficiency*
        - eg. a `struct` in C has a defined memory layout
- some common constructed types in programming languages:
    - an **enumeration** is an explicit listing of all a type's elements
        - AKA *scalar* types
        - defining named constants, and the new type as a set of the constants
        - usually each element represented in memory as an integer
            - but in Java, each element of an enumeration is an object of a class
        - eg. in C, `enum coin {penny, nickel, dime, quarter}`

        - eg. in ML, `type day = M | Tu | W | Th |F | Sa | Su`
    - an **n-tuple** is a Cartesian product of n sets
        - ie. tuples, triples, quadruples, etc.
        - can have unnamed tuple components:
            - eg. in ML, `type imag = float * float`
        - or named tuple components in *structure* or *record* types:
            - AKA *aggregate* types
            - eg. in C, `struct imag {double r; double i;}`
        - usually represented in memory as consecutive elements
    - **arrays** or **lists** are vectors of sets
        - in many languages, indexed by integer indices
        - in Pascal, arrays can be indexed by a variety of types, including characters and enumerations
        - many variations on arrays
            - eg. different sizes, resizing, higher dimensions
    - a **union** type can be one of many types at a time
        - eg. in C, `union elem {int i; float f;}`
        - in C, programmer is free to interpret the union as they want, type is unspecified, error-prone
            - ie. the underlying memory representation is exposed
        - in other languages, union is *discriminated* and programmer must specify the behavior corresponding to each type at runtime
            - ie. another abstracted type
    - **subtypes** are subsets of some existing set
        - subtypes are a *subset* of values, but can support a *superset* of operations due to inheritance
        - eg. in Pascal, `type digit = 0..9`
        - eg. classes in object-oriented programming
        - can optimize representation for subtype, does not necessarily require same representation as supertype
    - **function types** represent a mathematical set of mappings from domain to range of the function
        - eg. in ML, `fun (a, b) -> a > b` has type `int * int -> bool`
        - different languages support different operations on functions
            - eg. storing in variables or arrays, passing them or returning them from other functions
            - functional languages usually have the most support for function operations, ie. they have *first-class functions*
- consider the type `float` in C or C++:
    - a set of values `0.0, 1.0, ...`
    - a way to represent a set of values in a machine
    - a primitive, so C/C++ don't specify an exact representation
        - nowadays, follows the IEEE standard:

      - 1-bit sign, 8-bit exponent, 23-bit fraction
      - if $e = 0$ represents $\pm 2^{-126} \times 0.f$, otherwise $\pm 2^{e-127} \times 1.f$
      - if $e = 255$ and $f = 0$, represents $\pm\infty$
      - if $e = 255$ and $f \neq 0$, represents `NaN`
  - this is extremely complicated, with several special cases for what should be a simple type:
    - -0 and $+0$ have two different bit patterns, but *compare* equal
    - `NaN` never compare equal, but the bit pattern may be the same
- the problem with this complexity stems from:
  - **exception handling**
  - what happens when a floating point operation cannot return the correct answer:
    - overflow when exponent is too large
    - underflow when exponent is too small
    - bad arguments, eg. infinity minus infinity or zero divided by zero, no good answer
    - inexactness of float operations
  - traditionally, hardware would trap on overflow, underflow, and bad arguments, and silently round for inexact
    - not a good idea in heavy use of floating point operations
  - nowadays, most programs just use the special values
    - infinities and `NaN`
    - return exceptional value instead of throwing an exception
- using types to *address* such issues:
  - **annotation**:
    - types document the program, eg. `int i`
    - compilers can use these to generate beter code and report some errors
    - eg. Java and C require annotation definitions for most variables
  - **inference**:
    - *bottom-up* inference: `3 + 4.5 + 'a'` has type `double`
    - *top-down* inference: `[3;5;-4] = (f y z)` , right hand side has type `int list`
    - eg. ML performs extreme type inference, while Java will infer expression types from the operand types

Type Checking

- **static** vs. **dynamic** type checking:
  - static checking occurs *before* the program runs, typically at compile-time:

        - eg. C/C++, OCaml, Java
        - used for large, complex software where reliability is critical
    - dynamic checking occurs *as* the program runs, at runtime:
        - eg. Python, sh, JavaScript
        - used for smaller programs and scripting where flexibility and ease of use is more important
        - more expensive to maintain the type information of all values at runtime
        - may support optional type annotations that would help optimize the program
    - also a spectrum, some languages are mixed-use:
        - eg. Java allows for dynamic checking
        - especially important for object-oriented languages that support complex object subtypes
- some languages perform such thorough type checking that no type-incorrect application of an operation can go undetected:
    - all operations are type-checked, cannot subvert the type discipline
    - safer, but can be more cumbersome when types get in the way
    - such languages are **strongly typed**:
        - eg. OCaml, Java
        - note that some languages such as Python are strongly typed *and* dynamically checked
    - C/C++ are mostly strongly typed, with some exceptions:
        - eg. using pointers to subvert types, casting between `void *` and other pointer types
        - eg. the union type can allow for improper type applications
- **type-equivalence** decides whether two types are the same
    - in **name equivalence**, two types are the same if they have the same name:
        - eg. Pascal
        - eg. two identical structures with different names in C are not equivalent
    - in **structural equivalence**, two types are the same if their internal structure is the same:
        - ie. the same unless there is some difference in behavior
        - eg. in ML, `type pair1 = int * real` is the same as `type pair2 = int * real`
        - eg. two identical typedefs with different names in C are equivalent

Subtypes

- the semantics between **subtypes** should stem from the following relationship

between types:
  - types T = U if and only if T is a subtype of U and U is a subtype of T

Exploring subtypes in Pascal:

```
type alpha = 'a'..'z'; (* alpha is a subtype of char *)
var a: char;
var b: alpha;
a := b; (* always works *)
b := a; (* a might be out of range, compile-time or runtime error? *)
```

- in object-oriented languages, subtypes are derived naturally from classes and subclasses
  - a subclass has a *subset* of values from its parent class, but a *superset* of operations of its parent class, due to inheritance
    - allows for specification

Exploring subtypes in C:

```
// which is the subtype of the other?
char *p;
char const *q;

char buf[1000];
char *p = buf;
char const *q = buf;
*q = 'x'; // compilation error
q = p;    // OK

char const str[4] = "xyz"
char const *q = str // OK
char *p = str;      // compiler warning
*p = 'x';           // program has a SEGV

// thus char * is a subtype of char const *
// more ops on char * than char const *

// which is the subtype of the other?
char const * const *p;
char **q;

q = p; // bad
p = q; // OK in C++ but not C
// issue with rule complexity
```

Polymorphism

- **polymorphism** allows for more than one possible type in an operation
  - ie. *many forms* accepted by an operation
  - can be applied to many different programming language features
    - eg. functions, variables, functions differentiated by subclass, classes, languages
  - for dynamically checked languages, polymorphism is irrelevant in terms of type checking:
    - there are no compilation restrictions on type
    - functions can be called with any type of argument, but may cause a runtime error
    - of course, dynamically checked languages can still use other elements of polymorphism, such as overloading
  - on the other hand, for static checked languages, polymorphism allows for some of the freedoms of dynamic type checking:
    - still have compile time error checking, but with more flexibility with types
- under the surface, language may either:
  - create separate copies of the polymorphic function with specific types, and use overloading:
    - use *name-mangling* to implement multiple copies with the same names
      - eg. `cos$f` , `cos$d` for float and double implementations
      - but if parts of the program is compiled with different compilers, they have to agree about how names are mangled
  - create a single copy used by all callers:
    - not as efficient, compiler cannot optimize for a specific type
- specific types of polymorphism:
  - **ad-hoc** polymorphism:
    - each individual rule developed on its own, little overall organization
    - can lead to trouble
    - eg. overloading, parameter coercion
  - **parametrical** polymorphism:
    - a function's type can contain one or more *type variables*
    - eg. templates
- an **overloaded** name or operator has multiple definitions of different types:
  - identify a function or operation by examining the types of the operands (or its context)
    - eg. Ada has context overloading
  - eg. `+` operator in C and Java is overloaded for integers and doubles

  - language system will use the operands' types to determine which definition of the operator to apply
  - some languages allow operators to have added definitions
    - eg. in C++, almost all operators can have additional definitions, including array subscripting and function calls
  - function names can be overloaded in some languages as well
  - note that overloaded functionality can be obvious and natural, such as addition for different numbers
    - or more less obvious, such as using addition for concatenation

Overloading examples:

```
double cos(double); // cos(3.5)
float cos(float);   // cos(3.5f)
// with only one 64-bit cos, function would be slower
// and wrong for 32-bit arguments due to rounding during conversion
```

Adding definitions for existing operators in C++:

```
class complex {
  double rp, ip;
public:
  complex(double r, double i) {rp = r; ip = i;}
  friend complex operator+(complex, complex); // adding complex numbers
  friend complex operator*(complex, complex); // multiplying complex numbers
};
```

Overloading based on context:

```
float d = cos(1); // argument is int, but *context* is float
```

- **coercions** are implicit type conversions
  - eg. `double x = 2` vs. `double x = (double) 2`
  - some languages have very many and very complex coercion rules in their specification
    - others are not as generous with coercion
  - issues can come from ambiguity or coercion occuring unexpectedly
  - when a languages supports coercion in functional call parameters, this allows for **polymorphic** functions

Coercion pitfalls in C:

```
// + defined for only 9 types, with no *mixed* operations
int + int
unsigned + unsigned
double + double
```

```
...

// uses coercion for other cases
// 1. types narrower than int promote to int (no loss of information)
// 2. in type mismatch, narrower type is coerced to the wider (no loss)
//    - if the same width, unsigned wins
int + long       // int to long
int + unsigned   // int to unsigned
int = int + long // if result doesn't fit, convert back

unsigned i = -1       // coercion that *changes* numeric values
unsigned short j = -1 // coercion that *loses* information

-1 == j         // true, coerces integer to be unsigned
-1 < unsigned 0 // true
-1 < 2147483648 // false, constant is too large for int, so it is unsigned
-2147483648 < 0 // false
```

Issues from coercion with overloading:

```
int square(int x) {return x*x;}
double square(double x) {return x*x;}

square('a'); // calls int version since char is *closer* to int than double

void f(int x, char y) {}
void f(char x, int y) {}

f('a', 'b'); // compile-time error, ambiguous case
```

- with **parametric polymorphism**, parameter types use type variables:
  - eg. `fun (a, b) -> a = b` has type `'a * 'a -> bool`
  - eg. **templates** in C++:
    - templates stand for code that haven't yet been compiled
    - only become fully compilable once they are *instantiated*
  - eg. **generics** in OCaml and Java:
    - only one copy of translated code
    - all checking occurs during compilation of generic class
    - no optimization for specific types
    - all objects represented via pointers

Generic types in Java:

```
// without generics
public interface List {
  void add (Object);
  Object next();
  ...
}

List l;
l.add("abc");
if ((String) (l.next()).length() == 1)
// cast to String is necessary to compile
// but adds an uneccesary runtime check for type
  ...

// using generics
public interface List<E> {
  void add(E);
  E next();
  ...
}

List<String> c;
c.add("abc");
if ((l.next()).length() == 1)
// no need to cast
  ...
```

Templates in C++:

```
template<class X> X max(X a, X b) {return a>b ? a : b;}
void g(int a, int b, char c, char d) {
  int m1 = max(a, b);
  char m2 = max(c, d);
}
```

- with **subtype polymorphism**, parameter types can support subtypes
  - especially prevalent in object-oriented languages

Subtype polymorphism with java:

```
public class Car {
  public void brake() ...
}
```

```java
public class ManualCar extends Car {
  public void clutch() ...
}

void g(Car c) {c.brake();} // polymorphic
```

Functions

Parameters

- there are some *syntactic* issues dealing with parameter-passing:
    - correspondence, ie. how do caller's arguments match up to callee's parameters:
        - with **positional** correspondence, match arguments to parameters left to right, eg. C, Lisp, etc.
        - with **varargs** correspondence, allow for a variable number of arguments, eg. Scheme, Python
        - with **keyword** correspondence, arguments are tied to keywords, eg. Python
    - another parameter-passing feature offered by some languages are optional parameters with default values

Different correspondences in Python:

```python
def f(y):
    ...
def foo(x, *rest):
    ...
def bar(x, *y, **z):
    ...

bar(1, 2, 3, x=10, y=20) # x = 1, y = (2, 3), z = {'x': 10, 'y': 20}
```

- as well as *semantic* issues ie. different calling conventions, usually visible to programmer:

- in **call by value**, eg. C/C++, OCaml, Java, Python, etc.:

    - caller evaluates the argument, obtains a value
        - argument must be an rvalue
    - passes a copy of the value to the callee
    - callee has a copy, and can use it any way it likes, including updating the copy, without affecting the caller
    - performance *cons*:
        - value to copy may be very large eg. a large array
        - argument expression may be expensive to compute, eg. a cryptographical hash function, and callee does not need the value in this particular call

        - evaluating argument expression may result in an error

- in **call by reference**, eg. C/C++:
    - caller passes a pointer to the original value to the callee
        - note that references in C++ are compiled to pointer operations under the surface
        - essentially call by value with pointers
    - callee can inspect and modify the value (assuming a *shared* object space)
        - note that changes to the *object* are visible to the caller, while changes to the *reference* itself are purely local
    - *pros*:
        - no copying
    - *cons*:
        - trickiness due to **aliasing** ie. when there are two different names for the same variable
        - compiler forced to avoid optimizations because aliasing may occur
    - some ways to avoid aliasing, the `restrict` keyword in C means caller must *not* alias storage accessed via either poitner
        - eg. `char *strcpy(char restrict *dest, char const restrict *src)` means source and destination cannot overlap

Illustrating aliasing problems:

```
void trouble(int &x, int &y) {
  x = 5;
  y = x + 7;
  z = x; // x has to be 5 here, normally
}

// z = x = 5; // faster and equivalent, right?
// y = 12;    // nope, not if you have call by reference,
              // compiler would have to avoid many optimizations

// in caller
int n = 10;
int z;
trouble(n, n);
```

- in **call by result**, eg. Ada, Algol:
    - caller tells callee where the result should go
        - argument must be an lvalue
    - ie. the value is copied from callee to caller, when the callee returns
        - allows a function to return several values
    - eg. for `ssize_t read(int char *buf, size_t size)` in C, `buf` is really a

call-by-result array

- in **call by value-result**, eg. Ada:
  - call value, with call by result
    - pass a value into a method through a parameter, and get a different value out through the same parameter
    - ie. acts like call by value when called, and call by result when returning
  - like call by reference, except callee has a copy while it is running, so no need to worry about aliasing in callee
- in **call by unification**, eg. Prolog:
  - combination of call by value, reference, and result, with some differences
  - cannot change already bound value through unification, but unification places results in the arguments
- in **macro calls**, eg. Scheme, C/C++:
  - arguments are pieces of the program, result is another piece of the program
  - eg. `#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))` is expanded at preprocessing time in C/C++
  - *cons*:
    - has problems with **macro capture**, when macro definitions are *captured* by the callers definition
      - occurs since macros are *not* methods, and are usually expanded *textually* through textual substitution
      - body of a macro must be evaluated in the caller's context
    - arguments of a macro must be *re-evaluated* every time they are used in the macro
- in **call by name**, eg. Algol:
  - caller *does not* evaluate the argument
  - instead, it tells the callee how to evaluate it by passing a descption of the argument to the callee
    - note that the argument must return an lvalue *as well as* a rvalue depending on how the argument is to be used
  - call by name is to functions, as call by reference is to pointers
  - AKA **lazy evaluation**, putting off evaluating an expression as long as you can
    - program is simply setting up a computation to be done by creating a lot of *thunks*
  - *pros*:
    - solves the issue where callee does not need to evaluate an expensive argument if it doesn't require it
    - lazy evaluation
    - in a way, similar to macro calls, but solves macro capture issue

    - *cons*:
        - arguments must be re-evaluated every time they are used
            - useful if side effects occur, but if not, no need to re-evaluate
- in **call by need**, eg. Haskell:
    - call by name, while *caching* every thunk's result
    - *pros*:
        - even more *efficient* lazy evaluation
    - *cons*:
        - not good with side effects

Call by name examples:

```
int f (int name x) { return x++; }

int f (int (*xf) (void)) {return (*xf)() + 1; }

int zhelp (void) { return z; } // zhelp is a thunk
int z = 9;
return f(zhelp);

print_avg(int avg, int n) {
  if (n == 0)
    print("zero items");
  else
    print("%d items, avg is %d\n", n, avg);
}

print_avg(sum/n, n); // crashes in C with *eager* evaluation!
// but if call by name ie. *lazy* evaluation was used, would not crash,
// since callee would not evaluate the zero division
```

- the specific parameter-passing technique used is not always visible to the programmer:
    - eg. ML uses eager evaluation, so ML does not pass parameters by name, need, or macro
        - but since ML has no side effects, it is difficult to *distinguish* between the other possible parameter-passing styles to tell which style ML uses
    - eg. for imperative languages with side effects, it is possible to distinguish between all the different parameter-passing styles
    - eg. some languages offer multiple parameter-passing *modes*, such as Ada

Errors

- there is a *hierarchy* of types of errors in Scheme and other programming languages:
    1. *implementation* restriction:
        - program is not really at fault, it's following the Scheme spec
            - implementation is still *limited*
        - eg. memory exhaustion
    2. *unspecified* behavior:
        - Scheme may place some constraints on behavior, but it doesn't specify it exactly
            - program should work either way
        - eg. `(eq? '(x) '(x)')` gives false if there are two copies of the list, or true if compiler optimizes and keeps only one copy of the constant list
    3. an error is *signaled*, exception handling:
        - implementation is required to throw an exception or output an error message
        - eg. `(open-input-file "foo")` when `"foo"` does not exist
        - this is the approach where continuations are used
    4. *undefined* behavior:
        - a program should never do this
        - resulting behavior is undefined, implementations are not required to detect the error
            - implementation can do anything eg. dump core, etc.
        - Racket chooses to signal an error for `(car 0)` or `(car '())` , instead of crashing

Error Handling

- there are different strategies for *error handling*:

1. **static checking**: have compilers check for errors:
    - eg. dereferencing pointers in C is a runtime error, but in OCaml, statically checking with `'a option` prevents a similar runtime error
    - but some errors are harder to deal with in this way, eg. indexing an array out of bounds

Runtime errors in C:

```
char *p;
p = (NULL or somethingelse);
... *p ... // crashes, or undefined behavior
```

vs. static checking in OCaml:

```
let p = (None or ...) in
  ...
  match p with
    | None -> ...
    | Some x -> ...
```

2. **preconditions only**: do not handle errors at all, but instead *document* the preconditions for each method that should be satisfied to avoid errors:
    - caller's responsibility to make the precondition true, while callee can assume that it is true
        - leaves it up to the caller to handle errors, but also exposes implementation details
    - typically a runtime property
    - eg. `st.pop()` has a precondition that `st` is nonempty
    - can be *implemented* via runtime check, as well as via static checking

Implementing enforced preconditions:

```
char index(char *p, int i) {
  // preconditions:
  assert(p != NULL && 0 <= i && i < isizeof(p));
  return p[i];
}
```

3. **total definition**: define a *normal* behavior for *every* condition:
    - a function does *not* crash given bad inputs, but instead returns a special or error value
    - does not immediately catch errors, instead the error value propagates indefinitely until checked by the caller
    - eg. from IEEE floating point standard, 0 divided by 0 gives `NaN`
        - very popular approach in floating point applications eg. machine learning

Propagating floating point error values in Java:

```
double a = 27;
double b = 0;
double c = a / b; // c is +Inf
double d = c + 7; // d is also +Inf
```

4. **error flagging**: method flags errors in some way, requiring the caller to explicitly check for an error:
    - eg. `NULL` pointer in C denotes an error after calling `malloc`
5. **fatal errors**: any error condition detection prints an error message and terminates the program:
    - inflexible for the caller
    - prioritizing performance over safety
    - eg. `1/0` in C crashes the program
6. **exception throwing**: throw a *custom* exception:
    - like fatal errors, except control *stays* in the program
        - hybrid approach that solves some of the disadvantages of the other strategies
    - control flow is more *complicated*:
        - requires the source code to be changed
        - programmers must be aware that functions may not return and could throw an exception
        - trouble for programmers as well as compilers
    - exceptions are handled in a structured way, no need to manually use continuations

Exceptions in Java:

```
try {
  f(x ,y);
  g(w, x);
} catch (IOException e) {
  // only executed if f or g or one of their callees
  // throws an exception
  print("ouch!");
} finally {
  // always executed
  cleanup(w);
}
notcleanup(w); // might not be executed, eg. an uncaught exception is thrown
```

Exception handling as a complication:

```
// g may not return
int x = f(a, b);
delicate_operation_start(x);
int y = g(x, b);
delicate_operation_end(x);

// vs.
```

```
int x = f(a, b);
try {
  delicate_operation_start(x);
  int y = g(x, b);
} finally {
  delicate_operation_end(x);
}
```

Implementing Exceptions

- when an exception is thrown, the program must *dynamically* search through the call stack chain for a corresponding catch
  - the catcher is usually a different procedure than the thrower
    - catcher pushes a *marker* onto the stack (or marks its frame)
  - thrower than walks down stack looking for the catcher
    - this *cannot* be done statically
  - as the call stack is searched, pop the call frame, execute any remaining code, until a catcher is reached

Memory Management

- memory allocations occur for all variables and program data:
  - activation frames for automatic variables must be pushed and popped from the stack
  - unordered runtime memory allocation and deallocation must occur on the heap

Stack Management

- every function has an **activation record** or **frame** that is stored on the stack:
  - the frame records what you need to know about the particular call to the function
  - a recursive function can have multiple frames on the stack:
    - each frame *shares* the same code
    - but has *different* values for return address, arguments to function, local variables, temporaries, etc.
- originally (Fortran 1958), frames were *static*:
  - one frame per function
  - recursion was not allowed
  - but there is no stack and no problem of running out of memory
- C (1962):
  - allowed recursion
  - frame size and layout is known *statically*
    - eg. dynamically sized arrays are not allowed
- Algol (1960):
  - dynamically sized arrays are allowed
  - stack management gets harder, because the youngest frame can *grow* as the program executes
    - `%rsp - %rbp` is not a constant, so two registers are required to keep track of the frame
  - each frame header had three pointers:
    - return address
    - *caller's* frame pointer
      - creates a linked list of frames called the **dynamic chain**
    - *definer's* frame pointer
      - definer is the function that *statically* encloses this function
      - allows for nested functions, since definer is not necessarily the caller
      - creates a linked list of frames called the **static chain**

- in order to keep track of the definer, functions are represented by *two* pointers:
    - the code ie. the `ip`
    - the definer's frame ie. the `ep`
    - similar to how continuations work
    - we have *fat* 2-word function pointers instead of the thin ones in C/C++

Frame issues with nested functions:

```
int f(void) {
  int x;
  int g(void) {
    int y;
    return x + y;
    // easy to access y
    // to access x, need to know where f's frame is
    // to find f's frame, follow the definer's frame pointer
  }
  g();
  h(); // eventually calls g
}
```

- ML features all of the above, in addition to:
    - nested functions where the outer function returns, and the inner function is still valid and callable
        - currying uses this all the time
    - `let f x y = x + y` creates a nested function, so `f 3` returns the nested function that *remembers* the input 3
        - when `f` returns, its frame *should not* be reclaimed from the stack, but instead must be kept on the *heap*
            - *necessary* for such first-class functions that can be returned from other functions
        - thus function calls and returns are not as efficient, since heap management is more expensive than stack management

Heap Management

- the **heap manager** is responsible for keeping track of space on the heap:
    - two fundamental problems, keeping track of *allocated space* and keeping track of *free space*:
        - allocated space is referenced by **roots**, pointer variables on the stack

    or in static storage that point into the heap
    - other objects on the heap may be also be indirectly reachable from the root through other heap objects and pointers
  - each allocated block requires some *header* space to record size of block, previous object location, etc.
- tracking free space:
  - all heap managers need techniques to handle tracking of free space
    - whether heap manager takes the `malloc/free` or garbage collecting approach
  - maintain a **free list** so memory allocation can be done very fast, with minimal traversal of the heap
    - free list metadata is *embedded* in the heap itself, next to or within the header of blocks
    - used to quickly *coalesce* free blocks when adjacent blocks adjacent to newly-freed blocks are already free
- improving performance of free list tracking:
  - free list can become large as program's data gets larger
  - may take some time for a memory allocation request to work, to find a block that is big enough for the request
  - the *runts* or blocks at the start of the free list tend to become small, as allocations chip away at them
    - use a **roving pointer** ie. *next fit* approach on a cyclic free list, and leave the free pointer at the last successful block allocated from
- tracking roots:
  1. C/C++ force the programmer to *explicitly* take care of freeing unused heap memory after use, using `delete` and `free` :
     - after calling `free` , program promises to never use or even examine the pointer again
     - *pros*:
       - simplest approach
       - simplifies the heap manager
     - *cons*:
       - more hassle for programmers, keeping track of roots is the program's responsibility
       - forgetting to free causes a memory leak, causes performance issues
       - freeing too soon causes *dangling pointers*
       - freeing imposter pointers
  2. other languages choose to do away with explicit deallocation and use **garbage collection (GC)** instead for variables on the heap:
     - heap manager handles deallocation
     - works under the assumption that pointers cannot be forged
     - note that GC is typically done in user-mode code, without much

        help from OS for performance reasons
        - except for periodically requesting larger chunks of memory from OS
      - *pros*:
        - no freeing obligation for programmers
        - no dangling pointers
      - *cons*:
        - more overhead and complexity
        - heap manager must keep track of roots
        - heap manager must keep track of pointers from one object to another
        - unreachable storage is not *immediately* reclaimed
    - some language constructs also *implicitly* require allocation and deallocation
      - eg. file buffers, dynamically resizable arrays
- possible inclusion ie. root tracking errors from the heap:
  - in an *unused inclusion* error, a memory location in the heap is saved, but the program never actually uses the value stored there
  - in an *used inclusion* error, a memory location in the heap is saved, but the program uses the value stored there as something other than a heap address

Garbage Collectors

- there are three basic techniques for garbage collection:

1. a **mark-and-sweep** garbage collector carries out a two-phase process:
   - first, the collector traces all current roots ie. heap links and *marks* all the allocated blocks that are the targets of those links
     - needs a statically known table containing variable pointers, heap links, activation record layouts
     - table must be created by the compiler, since compiler knows object and frame layout, and location of the roots
     - could be part of the symbol tables, or separate
   - then, the collector *sweeps* over the heap, finding unmarked blocks and reclaiming them to the free list
   - *pros*:
     - allocated blocks are not moved, so inclusion errors are tolerated
   - *cons*:
     - the heap remains fragmented after garbage collection
     - not suitable for realtime applications, since an allocation can take a long time, proportional to the total number of objects in the system
     - if GC is run in a separate thread, thread now needs to collaborate with other threads, so bottlenecking locks are required

2. a **copying collector** uses only one half of its available memory at a time:
    - once the half becomes full, the collector finds all current heap links and copies all non-garbage blocks into the other half
    - *pros*:
        - compacts as it copies
        - proportional to number of bytes of *in-use* objects
        - no need to access free space and unecessarily pollute hardware cache
    - *cons*:
        - making copies can be expensive
        - have to *update* many pointers after copy (roots, as well as heap object pointers)
        - allocated blocks are moved, so sensitive to used inclusion errors
    - eg. Java uses a generation-based copying collector
        - in Java, every object has a method `o.finalize` that is called by the GC when it discovers `o` is garbage
            - gives program opportunity to release resources that are not under control of the GC
        - in mark-and-sweep, sweeper would be able to easily call `finalize` on each object
        - in a copy collector, there is no sweeper, so Java does *both* copying and sweeping
3. a **reference-counting collector** does not trace current heap links:
    - every allocated heap block includes a counter that keeps track of how many copies of its address there are
        - the reference counters are incremented and decremented throughout the program
        - once a count becomes *zero*, the block is known to be garbage and can be freed
    - *pros*:
        - garbage is quickly discovered and reclaimed immediately
    - *cons*:
        - less efficient performance due to *overhead* of maintaining counters
        - fails to collect *cycles* of garbage, since their reference counts all stay nonzero
            - programmers would need to know about cycles and modify their programs to break the cycles
        - major issues with multithreading due to global bottlenecking locks
    - eg. Python
        - performs mark-and-sweep occasionally to catch cycles

- other classes of garbage collectors:
    - **incremental collectors** recover a little garbage at a time, while the pro-

gram is running
    - avoids the issue of uneven performance due to periodic collections from mark-and-sweep and copying collectors
  - **generational collectors** divide their collector of allocated blocks into *generations* based on age
    - newer objects *tend* to point to older objects
      - older objects mutate rarely, while younger objects mutate more
      - most garbage are younger objects
    - garbage collects most often within the youngest generation ie. the *nursery*
      - GC each generation *separately*, with some special exceptions
      - speeds up GC and allocations
- is garbage collection in C/C++ possible?
  - seems impossible, because C/C++ compilers do not tell the memory manager where the roots are
    - nor do they tell the manager the object layout
  - but possible using **conservative** garbage collection:
    - using a mark-and-sweep approach
    - GC doesn't know which spots in memory or registers contain pointers, but it does know all the location where pointers *might* be
    - search in hardware registers, stack, static variables for *all* possible heap pointers:
      - if a word here *looks* like it could be a pointer into the heap, given the heap boundaries, assume it *is* a pointer to a heap object that is thus still in use
      - the words in these locations are typically small, and usually could not possibly be unintentionally valid pointer values
      - can decrease the chance of wrong assumptions by placing the heap in out-of-the-way locations, with addresses not likely to look like actual numbers
    - this is done for roots, as well as every word in every reachable object
      - solves issue of not knowing the object layout
  - no longer need to call `free` with conservative GC in C/C++
- other GC considerations:
  - GC and *multiple* threads:
    - terrible performance if allocations and deallocations grab a global lock
    - so give each thread its own nursery, so it can allocate without locking
  - *realtime* GC:
    - requires allocations to finish in a bounded amount of time
    - use incremental GC so that only a little GC is performed after allo-

cations, bounded by the time limit

- object *pooling*:
  - dedicated object pool for allocating and deallocating a type of objects that are commonly used in a program
  - but this technique is a performance hog if you are using a generation-based copying collector
    - since it may unecessarily recopy the object pool

Functional Programming

- motivation: *side effects* within expressions are bad news:
  - in C, leads to undefined behavior
  - in Java, follows certain left-to-right semantics, but the compiler omits optimizations that would have been possible in C
  - programs that care about side effects in expressions are usually buggy
    - eg. `f(x) + g(y)` vs. `g(y) + f(x)`
- Backus proposed **functional programming**, with the following motivations:
  1. **clarity**:
     - mathematical notations have been used for centuries
     - use these notations instead of inventing new ones
     - eg. `i = i+1` does not make sense *mathematically*
  2. **performance**, via parallelizability:
     - allow for clever compilers that can parallelize code (eg. across CPUs or even distributed systems)
     - "escape from the von Neumann bottleneck"
       - CPU $\Longleftrightarrow$ RAM, 1 instruction at a time
     - *avoid* thinking about programs as sequences of loads and stores of memory
- *terminology*:
  - a **function** is a mapping from a domain to a range
  - a **domain** and a **range** are a set of values
  - a **partial function** is one that doesn't map every element of the domain
    - eg. integer division is partial (`x / 0` or `INT_MIN / -1` fail)
  - having no **side effects** means that calling the same function twice on the same arguments gives the same answer
    - eg. `sin` and `cos` are **pure functions** in C, while `getc` typically has a side effect and is not pure
  - **functional forms** are functions that take functions as arguments:
    - eg. $\sum$ is a function that takes a function `f` as its argument
      - $\sum_{0 \leq i < n} f(i) = f(0) + f(1) + ... + f(n-1) = \sum(0, n, f)$
    - eg. other math notation like $\int$ or $f \circ g$
  - **referential transparency**:
    - ie. when you see a variable, you know exactly what value it refers to
    - in C, the uses of a variable might have different values because it may change
    - on the other hand, in a functional language, this can't happen
    - *pros*:
      - program is easier to understand

            - program is easier to optimize, compiler can cache values in register

- *evaluation*:
    - evaluation order is *not* controlled via sequencing
        - as opposed to `A; B; C;` in iterative programming like C
    - by giving up side effects, there is no I/O or assignments
    - instead, to control evaluation order, nested function calls are used:
        - eg. `f(g(x), h(y))` calls `g` and `h` and gets their return values before `f`
        - note that neither `g` nor `h` precedes each other, ie. partial ordered
        - thus `g` and `h` can be evaluated *in parallel* by the system
        - in C, the statement's ordering is undefined
- so how do functional languages do I/O?
    - they mainly don't, can use the read-eval-print loop is used instead
    - another example of modifying an I/O function so that it is pure:
        - `c = getc(f)` vs. `(f1, c) = getc(f)` so argument isn't modified
        - other ways to rewrite side effects
- note that side effects are inherent to dynamic linking (self-modifying state of program):
    - would like the advantage of building a program on the fly, without the dangers of side effects
- functional languages such as OCaml allow for this by *gluing* functions together, instead of modifying pre-existing code
    - using functional programming also allows for *extra* power than typical function modules
        - eg. mutual recursion

Currying

- **currying** is a concept invented by Haskell Curry
    - restricted his papers to functions with only a single argument
    - and allowed his functions to return functions, so that the effect of *many* arguments was achieved with just one
- eg. the curried function `(carctan(y)) (x)`
    - has the type `float → (float → float)`
    - `carctan` *returns* a function that gives the `arctan` fixed for a specific `y`
    - thus in ML, we can write simply `carctan y x`

Currying example:

```
let cons a b = a::b;;
let prepend_10 = cons 10;;
```

```
prepend_10 [3;0];;
```

Pattern Matching

- many functional programming languages use pattern matching
  - ML provides patterns for matching parts of lists, tuples, default matches, etc.
  - ML provides a match expression that attempts to match the first pattern
    - note that each expression must have the same type

Pattern matching in ML:

```
match EXPR with
  | P1 -> E1
  | P2 -> E2
  ...
```

Tail Recursion

- there are different types of recursive functions:
  - a **tail recursive** function is one where the returned expression is only a recursive call
    - ie. the function is *committed* to returning what the call is returning
    - typically uses an accumulator to store the value of the previous call
  - for a **non-tail recursive** function, this is not the case
  - a **backward recursion** is a recursion where the parameters values are less than in the previous step
  - a **forward recursion** is a recursion that grows bigger at each step
- the advantage to using tail recursion in functional languages is that they can perform **tail call optimization** at a low level:
  - for very long lists or recursions, **call stack** would have to record local variables and return variables
  - compiler notices that the return value is a tail call, the caller's stack frame is popped off *before* the call, since it was going to return the callee's result anyway
    - this tail call optimization prevents the stack from overflowing where it normally would
    - essentially reduces the call space performance from $O(n)$ to $O(1)$
    - important optimization to enable very deep recursion

  - many compilers for languages, even non-functional languages, perform tail call optimizations
    - eg. part of the spec for Scheme and ML, but not required by compilers for C

Different factorial implementations:

```
(* backward, non-tail recursive *)
let rec fac n =
  match n with
    | 0 -> 1
    | _ -> n * (fac n-1) (* multiplies return value *)

(* forward, tail recursive*)
let fac n =
  let rec helper acc i =
    if i >= n then
      acc
    else helper (i*acc) (i+1) (* recursion is returned directly *)
  in helper 1 1
```

Different array sum implementations:

```
(* without tail recursion *)
let rec sum l =
  match l with
    | [] -> 0
    | x::xs -> x + (sum xs)

(* with tail recursion *)
let sum l =
  let rec helper acc l =
    match l with
      | [] -> acc
      | x::xs -> helper (acc+x) xs
  in helper 0 l
```

ML

- **ML** is a popular functional language
  - has a *standard* dialect (SML) and an *object-oriented* dialect (OCaml)
  - this section uses the SML-NJ dialect
- properties of ML:
  1. **functional**, functions are *first-class objects*, ie. they can be passed into functions and treated as any other variables
     - supports higher-order functions
  2. immutable, variables (including lists) cannot be modified
  3. uses **type inference** to automatically choose types
     - allows for compile-time type checking, without having to specify the types most of the time
     - however, functions and operators can't have *overloaded* definitions
  4. features garbage collection
     - no `free` or `del`
  5. never does *implicit* casts
  6. functions never return
     - the last expression in a function is its result

Syntax

- literals:
  - `1234;` int constant
  - `123.4;` real constant
  - `~34;` int constant of -34 using negation operator `~`
  - `true;` , `false;` bool constants
  - `"fred";` string constant
  - `#"H";` char constant
- operators:
  - `12 div 5;` integer division
  - `7 mod 5;` modulo remainder
  - `~3;` negation
  - `12.0 / 5.0;` real division
  - `"tip" ^ "top";` concatenation
  - `< > ≤ ≥` ordering comparison
  - `=` equality
    - * cannot use equality operator with real numbers
  - `<>` inequality

  - `orelse andalso not` boolean operators
    - ML supports **short-circuit** evaluation
  - left-associative, typical precedence levels
- conditionals:
  - syntax: `<cond-expr> ::= if <expr> then <expr> else <expr>`
  - `(if 1 < 2 then 34 else 56) + 1;` gives int 35
- type conversion:
  - ML does not support mixed-type expressions or automatic type conversions
    - `1.0 * 2;` throws an error, multiplication is not **overloaded** for different operand types
  - `real(123);` gives 123.0 with type real
  - `floor(3.6);` gives 3 with type int
  - also `ceil round trunc` for real types
  - `ord chr str` for char and string operations
- function application:
  - can call functions *without* parentheses
  - `f(1)` , `(f)1` , `(f 1)` , `f 1` all equivalent
  - style is to use `f 1`
  - function application has the highest precedence, and is left-associative
    - eg. `f a+1` is the same as `(f a) + 1` , `f g 1` is not the same as `f(g(1))`
  - *conversly*, function types are right-associative
    - eg. `int -> float -> string` is the same as `int -> (float -> string))`
- variable definition:
  - `val x = 1 + 2 * 3;`
  - `x;` gives 7 with type int
  - can use `val` to redefine an existing variable (new value or new type)
  - note that this is *not* like an assignment statement in imperative programming:
    - a new definition does not have side effects on other parts of the program
    - parts of the program using the old definition before redefinition is still using the old definition
  - the `it` variable in the REPL always has the value of the last expression typed
- tuples:
  - `val barney = (1+2, 3.0*4.0, "brown");` gives `(3,12.0,"brown")` with type `int * real * string`
  - `val point = ("red", (100, 200));` gives `("red",(100,200))` with type `string * (int * int)`

        - `*` is a **type constructor** for tuples
        - the type `string * (int * int)` is a different type from `(string * int) * int`
    - `#2 barney;` gives 12.0 with type real (1-indexed)
    - `#1 (#2 point);` gives 100 with type int
    - note that a tuple of size one does not exist
- lists:
    - all elements are the same type
    - `[1, 2, 3];` gives `[1,2,3]` with type `int list`
        - `list` is a type constructor
    - `[true];` gives `[true]` with type `bool list`
    - `[(1,2), (1,3)]` gives `[(1,2),(1,3)]` with type `(int * int) list`
    - `[[1,2,3], [1,2]]` gives `[[1,2,3],[1,2]]` with type `int list list`
    - `nil` or `[]` is an empty list
        - has type `'a list`
        - names beginning with an apostrophe are **type variables** (unknown type)
    - `null` function checks whether a list is empty
    - `hd` function returns first element, `tl` function returns rest of list after first element
        - error on empty lists
    - `explode` function converts a string into a char list, `implode` function performs the opposite
    - `@` operator concatenates two lists of the same type
        - `[1,2] @ [3,4];` gives `[1,2,3,4]` with type `int list`
    - `::` operator pushes an element into the front of a list
        - same as `cons` or construct operator from Lisp
        - `1::[2,3];` gives `[1,2,3]` with type `int list`
        - used often for natural recursive constructions
        - right-associative, `1::2::3::[];` gives `[1,2,3]` with type `int list`
- function definitions:
    - syntax: `<fun-def> ::= fun <fun-name> <parameter> = <expression> ;`
    - `fun firstChar s = hd (explode s);` gives `firstChar` with type `fn : string -> char`
        - `->` is a type constructor for functions
        - domain and range types are automatically determined
    - `firstChar "abc"` gives `#"a"` with type char
    - for multiple parameters, use tuples:
        - `fun quot (a, b) = a div b;` gives `quot` with type `fn : int * int -> int`
        - `quot (6, 2);` gives 3 with type int
            - `val pair = (6, 2);` , `quot pair` gives the same result

- using recursion:
  - recursion is used heavily in ML
  - `fun fact n = if n = 0 then 1 else n * fact(n - 1);`
  - `fun listsum x = if null x then 0 else hd x + listsum(tl x);`
  - `fun length x = if null x then 0 else 1 + length(tl x);`
    - function `length` has type `fn : 'a list -> int`
    - indicates input is a list of elements with unknown type
    - this is a **polymorphic** function that allows parameters of different types
  - `fun badlength x = if x = [] then 0 else 1 + badlength(tl x);`
    - function `badlength` has type `fn : ''a list -> int`
    - indicates input is restricted to *equality-testable* types
    - function does not work on lists of reals, since reals cannot be tested for equality
      - due to `x = []` check
  - `fun reverse L = if null L then nil else reverse(tl L) @ [hd L]`
- types and type annotations:
  - type constructors include `list * ->`
  - `list` has the highest precedence, `->` has the lowest precedence
    - `int * int list` is the same type as `int * (int list)`
  - for the function `fun prod(a, b) = a * b;` , ML decides on the type `fn : int * int -> int`
    - ML uses the *default type* for the multiplication operator
    - to use with reals, have to include a **type annotation**
      - type annotations can be placed after any variable or expression, but best to keep it as readable as possible
    - `fun prod(a:real, b:real) : real = a * b;` has type `fn : real * real -> real`

Patterns

- ML automatically tries to match values to certain **patterns**
  - patterns also introduce new variables
  - eg. patterns appear in function parameters:
    - `fun f n = n * n;`
      - the pattern `n` matches any parameter and introduces a variable `n`
    - `fun f (a, b) = a * b;`
      - the pattern `(a, b)` matches any tuple of two items and introduces two variables `a` and `b`
- more patterns:

  - `_` in ML matches anything and does not introduce any variables:
    - `fun f _ = "yes";` has type `fn : 'a -> string`
  - can match only a single constant:
    - `fun f 0 = "yes";` has type `fn : 'int -> string'` but with a warning for *non-exhaustive* matching
      - throws an error if called on an integer value that isn't 0
  - matching a list of patterns:
    - `fun f [a, _] = a;` has type `fn : 'a list -> 'a` but with a non-exhuastive matching warning
      - only matches lists with exactly two elements
  - matching a cons of patterns:
    - `fun f (x :: xs) = x;` has type `fn : 'a list -> 'a` but with a non-exhuastive matching warning
      - matches any non-empty list and introduces `x` bound to the head element and `xs` bound to the tail
      - almost exhaustive, but fails on the empty list
- the grammar for multiple pattern function definitions:
  - `<fun-def> ::= fun <fun-bodies> ;`
  - `<fun-bodies> ::= <fun-body> | <fun-body> '|' <fun-bodies>`
  - `<fun-body> ::= <fun-name> <pattern> = <expression>`

Using multiple function patterns:

```
(* type int -> string, non-exhaustive *)
fun f 0 = "zero"
  | f 1 = "one";
```

For overlapping patterns, ML tries patterns in order:

```
(* type int -> string, exhaustive *)
fun f 0 = "zero"
  | f _ = "non-zero";
```

Equivalently, in non pattern-matching style:

```
fun f n =
  if n = 0 then "zero"
  else "non-zero";
```

Rewriting functions in this style clearly separates base case from the recursive case:

```
fun fact 0 = 123
  | fact n = n * fact(n - 1);

fun reverse nil = nil
  | reverse (first :: rest) = reverse rest @ [first];
```

```
fun sum nil = 0
  | sum (first :: rest) = first + sum rest;

fun countTrue nil = 0
  | countTrue (true :: rest) = 1 + count_true rest
  | countTrue (false :: rest) = count_true rest;

fun incrAll nil = nil
  | incrAll (first :: rest) = first + 1 :: incr_all rest;
```

Restrictions of pattern-matching style:

```
(* the same variable cannot be used more than once in a pattern
 * fun f (a, a) = ...
   * | f (a, b) = ...;
 *)

(* cannot use pattern-matching *)
fun f (a, b) =
  if (a = b) then ...
  else ...;
```

Pattern-matching in variable definitions:

```
val (a, b) = (1,2.3);
val a :: b = [1,2,3,4,5];
```

Local Variable Definitions

- the `let` expression allows for local variable definitions
  - syntax: `<let-exp> ::= let <definitions> in <expression> end`
  - definitions cannot be accessed outside the environment of the `let`
  - the value of the evaluated expression is the value of the entire `let` expression

Using `let` :

```
let val x = 1 val y = 2 in x + y end;
(* it has value 3, x and y are unbound *)
```

Alternatively:

```
let
  val x = 123
  val y = 2cm
in
  x + y
end;
```

More practical example with `let` :

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

`let` with function pattern-matching:

```
fun halve nil = (nil, nil)
  | halve [a] = ([a], nil)
  | halve (a :: b :: cs) =
      let
        val (x, y) = halve cs
      in
        (a :: x, b :: y)
      end;

fun merge (nil, ys) = ys
  | merge (xs, nil) = xs
  | merge (x :: xs, y :: ys) =
      if (x < y) then x :: merge(xs, y :: ys)
      else y :: merge(x :: xs, ys);

fun mergeSort nil = nil
  | mergeSort [e] = [e]
  | mergeSort theList =
      let
        val (x, y) = halve theList
      in
        merge(mergeSort x, mergeSort y)
      end;
```

Case Expression

- syntax for a `case` expression:
  - `<rule> ::= <pattern> => <expression>`
  - `<match> ::= <rule> | <rule> '|' <match>`
  - `<case-exp> ::= case <expression> of <match>`

Although many languages have a case construct, ML's `case` allows for powerful general pattern matching:

```
(* returns the third element if 3+ elements, second if 2, first if 1, 0 if empty *)
case x of
  _ :: _ :: c :: _ => c |
  _ :: b :: _ => b |
  a :: _ => a |
  nil => 0
```

Higher-Order Functions

- function names are variables just like any others in ML
  - the are just initially bound to a function
  - functions themselves do not *have* names
  - eg. can rebind the negation operator:
    - `val x = ~;`
    - `x 3;` gives -3 with type int
  - can *extract* the function itself from a builtin operator such as `>` using `op`
    - `quicksort([1,2,3,4,5], op >)` gives `[5,4,3,2,1]` if the quicksort function takes a list and a comparison function
- can create **anonymous** functions using the keyword `fn` followed by a match instead of `fun` :
  - `fun f x = x + 2;` has the same effect as to `val f = fn x =>  + 2;`
    - except that only the `fun` definition has a scope including the function body, so only the `fun` version can be recursive
  - `(fn x => x + 2) 1;` gives 3 with type int
- **higher-order functions (HOFs)** are functions that take another function as a parameter or returns a function
  - functions that do not involve other functions have order 1 and are not higher-order
  - HOFs provide an alternative for squeezing multiple parameters into a single tuple:

- using **currying** to write a function that takes the first parameter, and returns another function that takes the second parameter, etc., until the ultimate result is returned

Using currying:

```
fun f (a, b) = a + b;
f (2, 3);

fun g a = fn b => a + b;
g 2 3; (* same as (g 2) 3 *)
```

Calling curried functions with only some of their parameters:

```
val add2 = g 2;
val add3 = g 3;
add2 3; (* gives 5 *)
add3 3; (* gives 6 *)

(* defining quicksort as a curried function with type:
 * ('a * 'a -> bool) -> 'a list -> 'a list
 *)
quicksort (op <) [1,4,3,2,5]; (* gives [1,2,3,4,5] *)
val sortBackward = quicksort (op >);
sortBackward [1,4,3,2,5]; (* gives [5,4,3,2,1] *)
```

Extending parameters:

```
fun f (a,b,c) = a + b + c;
f (1,2,3);

fun g a = fn b => fn c => a + b + c;
g 1 2 3;

fun g a b c = a + b + c; (* equivalent abreviation *)
```

Predefined Higher Order Functions

- the `map` function:
  - has type `('a -> 'b) -> 'a list -> 'b list`
  - applies some function to every element of a list, creating a list with the same size
  - `map ~ [1,2,3,4];` gives `[-1,-2,-3,-4]`
  - `map (fn x => x+1) [1,2,3,4];` gives `[2,3,4,5]`
  - `map (fn x => x mod 2 = 0) [1,2,3,4];` gives `[false,true,false,true]`

  - `map (op +) [(1,2),(3,4),(5,6)];` gives `[3,7,11]`
- the `foldr` function:
  - has type `('a * 'b → 'b) → 'b → 'a list  → 'b → 'a list → 'b`
  - combines all the elements of a list into one value, starting from the rightmost element
  - takes a function, a starting value, and a list of elements
    - `foldr (fn (a,b) => ...) c x`
    - first call of anonymous function starts with `a` as rightmost element and `b` as `c`
    - then, `b` will hold the result accumulated so far
    - `b` , `c` , and the return value of `foldr` and the anonymous function are all the same type
    - `a` and the type of elements of `x` are the same type
    - `c` is returned when the list is empty
  - `foldr (op +) 0 [1,2,3,4];` gives 10
  - `foldr (op * ) 1 [1,2,3,4];` gives 24
    - need extra space to avoid comment delimiting
  - `foldr (op ^) "" ["abc","def","ghi"];` gives `"abcdefghi"`
  - `foldr (op ::) [5] [1,2,3,4];` gives `[1,2,3,4,5]`
  - `fun filterPositive L = foldr (fn (a,b) => if a < 0 then b else a::b) [] L;`
- the `foldl` function:
  - has same type as `foldr` , but starts from the leftmost elements
  - same result as `foldr` for associative and commutative operations
  - `foldl (op ^) "" ["abc","def","ghi"];` gives `"ghidefabc"`
  - `foldl (op -) 0 [1,2,3,4];` gives 2 as opposed to -2 called with `foldr`

Type and Data Constructors

- in C and C++, unions share the same address and only one of the types is valid at once
  - eg. `union u { long l; char *p; };`
  - *cannot* tell which of the types is valid at once, can get garbage depending on the requested type
- on the other hand, unions in ML are represented by a piece of storage with extra room to hold a type tag
  - ie. a **discriminated union**
  - can thus use matching to differentiate between these created types
- the `datatype` definition creates an enumerated type:
  - `datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;`
  - `fun isWeekDay x = not (x = Sat orselse x = Sun);`

- the name of the type is a **type constructor** and the member names are **data constructors**
    - data constructors here act as *constants* in a pattern
- the only permitted operators are comparisons for equality
- the actual ML definition for booleans is `datatype bool = true | false;`

- a parameter to a data constructor can be added with the keyword `of` :
    - `datatype exint = Value of int | PlusInf | MinusInf;`
        - each `Value` will contain an int, `Value` itself is a function that takes an int and returns exint
        - `Value 3;` gives `Value 3` with type exint
        - however, cannot treat as an int and perform operations
        - have to extract using pattern matching:
            - `val x = Value 5;` , `val (Value y) = x;` gives 5 with type int

Pattern matching with data constructors:

```
(* exhaustive matching *)
val s = case x of
  PlusInf => "infinity" |
  MinusInf => "-infinity" |
  Value y => Int.toString y;

fun square PlusInf = PlusInf
  | square MinusInf = PlusInf
  | square (Value x) = Value (x * x);
```

- a type constructor can have parameters too, allowing for *polymorphic* or *generic* type parameters:
    - `datatype 'a option = NONE | SOME of 'a;`
    - the type constructor is named `option` and takes type `'a` as a parameter
    - `SOME 4;` gives the type `int option`
    - `SOME 1.2;` gives the type `real option`
    - `SOME "pig";` gives the type `string option`

Polymorphic type parameter examples:

```
fun optdiv a b = if b = 0 then NONE else SOME (a div b);

datatype 'x bunch = One of 'x | Group of 'x list;
One 1.0; (* type real bunch *)
Group [true,false]; (* type bool bunch *)

fun size (One _) = 1
  | size (Group x) = length x;
```

```
(* here, ML resolves the returned type to int *)
fun sum (One x) = x
  | sum (Group xlist) = foldr (op +) 0 xlist;
```

Recursion with Constructors

- type constructors can also be used *recursively*
    - eg. the actual list type definition in ML is recursive
    - `datatype 'element list = nil | :: of 'element * element list;`

Defining type constructors recursively:

```
datatype intlist = INTNIL | INTCONS of int * intlist;
INTNIL; (* represents an empty list *)
INTCONS (1, INTNIL); (* represents a list of just 1 *)
INTCONS (1, INTCONS (2, INTNIL)); (* represents a list of 1 and 2 *)

fun intlistLength INTNIL = 0
  | intlistLength (INTCONS (_,tail)) =
      1 + (intlistLength tail);
```

Creating a parameterized list type:

```
datatype 'element mylist = NIL | CONS of 'element * element mylist;

fun myfoldr f c NIL = c
  | myfoldr f c (CONS(a,b)) =
      f(a, myfoldr f c b);
```

Defining polymorphic binary trees:

```
datatype 'data tree = Empty | Node of 'data tree * 'data * 'data tree;
val treeEmpty = Empty;
val tree2 = Node(Empty, 2, Empty);
val tree 123 = Node(Node(Empty,1,Empty), 2, Node(Empty,3,Empty));
```

Binary tree operations:

```
fun sumall Empty = 0
  | sumall (Node(x,y,z)) =
      sumall x + y + sumall z;

fun isintree x Empty = false
```

```
  | isintree x (Node(l,y,r)) =
      x = y
      orelse isintree x l
      orelse isintree x r;
```

OCaml

- OCaml is a newer flavor of ML
    - has some object-oriented features
    - more modernized (in some ways)
- in OCaml, statements are ended by the double semicolon `;;` rather than the single `;`
- literals:
    - `3.141;;` has type float
    - `'j';;` has type char
    - `(3, true, "hi");;` has type `int * bool * string`
        - empty tuple `()` is type `unit`
    - `[1; 2; 3];;` has type `int list`
        - empty list `[]` has variable type `'a list`
- operators:
    - negation operator is `-` instead of `~`
    - division operator for int is `/` instead of `div`
    - `-3 * (1+7) /2 mod 3`
    - `-1.0 /. 2.0 +. 1.9 *. 2` , float operations have an extra `.`
    - `a || b && c`
- variable definition:
    - uses `let` and `and` instead of `val`
    - `let name = ...`
    - `let a = 3 and b = 5 in ...`
- pattern matching:
    - uses `match` instead of `case`
    - `match opt with None -> ... | Some x -> x`
- local declarations:
    - `let x = 123 in let y = 321 in x + y` gives 444 with type `int`
- tuples:
    - cannot use `#` to index into tuple, instead use pattern matching
- lists:

  - uses `List.fold_left` and `List.fold_right` instead of `foldl` and `foldr`
- modules:
  - use `open` to open or import a module
  - `open List;;` , `length [1;2;3];;`

Functions

- functions:
  - instead of `fun f x y = ...` , `let f x y = ...`
  - can use `function` syntactical sugar for pattern matching on a single parameter
  - can omit `fun` altogether
  - for anonymous functions, `fun x -> x * 2`
  - `rec` is required for a recursive variable definition

Fibonacii in OCaml with `fun` :

```
let rec fib = fun n ->
  if n < 2 then n
  else fib (n-1) + fib (n-2)
```

Fibonacii in OCaml with `function` syntactic sugar:

```
let rec fib = function
    0 -> 0
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2)
```

Fibonacii in OCaml with fully abbreviated syntactic sugar:

```
let rec fib n =
  if n < 2 then n
  else fib (n-1) + fib (n-2)
```

- as with ML, can use match expressions with functions
  - shorthand, as well as full declaration

Matching with OCaml functions:

```
match l with (* only matches the first list in a pair of lists in a list*)
  | (p, _)::_ -> p
  | [] -> []

let car (h::_) = h (* implicit unexhaustive match expression *)
let safer_car = l ->
  match l with
    | h::_ -> h
    | [] -> 0 (* but now only works with int lists *)

let scar_def d l = (* alternative with a supplied default *)
  match l with
    | h::_ -> h
    | [] -> d
```

Types

- type declarations:
  - uses `type` instead of `datatype`
  - `type 'a option = None | Some of 'a`

Using Ocaml types and constructors:

```
type mytype =
  | Foo (* no extra payload of data, ie. constant *)
  | Bar of int
  | Baz of int * int

(* This creates three new constructors to assemble or disassemble the new type *)
Foo
Bar 12
Baz (3,5)

match myvalue with
  | Foo -> 0
  | Bar x -> x
  | Bar (y,z) -> y+z

(* custom generic, recursive type *)
type 'a sequence =
  | Empty
  | Nonempty of 'a * ('a sequence)
```

Scheme

- **Scheme** is a variant of Lisp, which is widely used in AI and other applications
  - high-level language with operations on structures data such as strings, lists, vectors
- properties of Scheme:
  1. objects are dynamically allocated and never freed:
     - objects are retained indefinitely, and are first-class data values as well
     - there is a garbage collector
     - like OCaml, Java, Prolog
     - unlike C/C++
  2. types are properties of objects, not of programs or expressions:
     - dynamic type checking, not static
     - ie. types are *latent*, not *manifest*
     - like Python, Prolog
     - unlike C/C++, Java (mostly)
  3. static scoping:
     - scope of a variable is known statically
     - to find out what an identifer refers to in **static scoping**:
       - look in this block, then in the statically containing block, that block's container, etc.
     - as opposed to **dynamic scoping**:
       - look in this function, then in the caller, caller's caller, etc.
     - eg. an expression like `let f = fun a -> a + b in (let b = 21 in f 21)` works with dynamic scoping, but not static
     - like OCaml, Java, Python, Prolog, etc.
     - unlike classic Lisp or environment variables which use dynamic scope
  4. call by value:
     - caller evaluates arguments of a function, before the function is called
     - property of isolation ie. passes copies of arguments' values to callee, and callee's changes to its copies do not affect the caller
     - like OCaml, C/C++, Java (mostly)
       - C++ also has call by reference
     - unlike Prolog which uses call by unification
  5. wide variety of builtin objects:
     - including members, strings, lists, procedures (including powerful, builtin continuations)
     - **continuations** allows for powerful arbitrary constrol structures

6. very simple syntax, with a straightforward representation of programs as objects:
    - the program syntax *itself* is the parse tree
        * the entire *program is data*, built entirely out of modifiable Scheme lists
        * allows for self-modifying programs that are useful for AI development
    - allows for simple compilers that can be very fast and reliable
    - also allows for easily defining new syntactic forms ie. **syntactic extensions**
7. tail recursion optimization is required:
    - unlike C, C++ where language spec does not require compilers to support tail recursion optimization
    - additionally, functions are first-class data objects
8. arithmetic is high level and reasonably machine-independent:
    - integers are limited in size only by available memory
    - exact rational numbers, without rounding error
        * represented by numerator, denominator pairs
    - for irrational numbers, falls back to inexact floating point numbers
    - complex numbers

- drawbacks of Scheme:
    - ugly syntax
        * LISP AKA "lots of irritating and spurious parentheses"

Syntax

- identifiers:
    - `[a-zA-Z0-9+-.?*/<=>:$%^&_~@]+`
        * many supported characters that are special characters in other languages
    - except cannot begin with `@0-9.+-`
        * except the valid identifiers `->anything + - ...`
    - case-insensitive
    - identifiers are used as:
        1. variables
        2. keywords
        3. symbols like Prolog atoms
    - eg. `(let ((x 'abc)) (cons x 3))` includes all 3 uses
- comments start with `;`
    - block comments delimited with `#| |#`
    - datum comments starts with `#;` prefix followed by a datum or printed

data value

- literals:
  - `"strings"`
  - `#\c` is the character ‘c’
  - `12 1e-9 1/2 1.3 1.2-2.7i -1.2@73` etc.
- lists:
  - lists are always built out of pairs ie. conses
    - in an *improper* list, the last element is not the empty list
    - in a *proper* list, the last element is the empty list
    - `(a . b)` is a pair and improper list
    - `(a b c)` is a proper list
  - `(l i s t s)` is syntactic sugar for `(l.(i .(s .(t .(s.)))))`
  - `'(a b .c)` evaluates to the same data as `(cons 'a (cons 'b 'c))`
  - `'()` empty list, aliased to `nil`
  - lists can be nested and contain different types of objects
    - `(4.2 "hi")` , `((1 2) (3 4))`
  - matched sets of brackets can used in place of parentheses, usually used for subexpressions for readability
- list operations in Scheme:
  - `car` extracts the first element of a list
  - `cdr` (pronounced could-er) extracts a list of the remaining elements
  - both require pairs as arguments

Manipulating lists:

```
(car '(a b c)) ⇒ a      ; head of list
(cdr '(a b c)) ⇒ (b c) ; tail of list
((car (list + - * /)) 2 3) ⇒ 5

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

(list 'a 'b 'c) ⇒ (a b c) ; always builds a proper list
```

- vectors:
  - array type, rather than a linked list type
  - `#(this is a vector)`
- booleans:
  - `#t` , `#f`
  - `#f` is the only false value in Scheme, all other values count as true
    - so the empty list, 0 are both true
    - thus boolean expressions are not restricted to boolean values

- *predicates* in Scheme answer a specific question and return a boolean:
  - eg. `<= >= < >`
  - testing equality:
    - `=` checks value equivalence for numbers, can be given more than two arguments
    - `==` checks memory equivalence
    - `eq?` tests equivalence of the same object, like pointer comparison
      - note that two pairs are not the same by `eq?` if they are created by different calls to `cons`
      - `(eq? '(1 2 3) (cons 1 '(2 3)))` gives false
      - `(eq? (cons 'a 'b) (cons 'a 'b))` gives false
      - `eqv?` is the same as `eq?` except it always gives true for the same *primitive* values
    - `equal?` will dereference numbers, strings, vectors, pairs, recursively
      - `(eq? '(1 2 3) (cons 1 '(2 3)))` gives true
  - *type predicates* test the type of an object
    - eg. `pair? number? symbol? list? null?`
- *quote* expressions using `'` :
  - `'x` evaluates to the data item `x`, without evaluating `x`
    - ie. distinguishes between lists and function evaluation
  - `'(f x)` yields `(f x)` without calling `f`
    - same as `(quote (f x))`
    - `'(+ 3 4)` gives the list `(+ 3 4)`
  - quoting an identifier treats the identifier as a symbol rather than an identifier
    - `('hello)` gives the symbol `hello`
  - note that `quote` does not evaluate its arguments, unlike normal procedures
- the backtick or *quasiquote* `` ` `` is used in conjunction with the *unquote* `,` :
  - same as `quasiquote` and `unquote`
  - the quasiquote by itself acts as a normal quote
    - but within the whole quasiquote, the unquote parts' symbols should be combined and evaluated
    - `,@` is an unquote expand to expand list into elements
  - `` `(1 2 ,(+ 1 2) ,(- 5 1)) `` gives `(1 2 3 4)`
- scheme naming conventions:
  - predicate names that return true or false end in `?`
    - eg. `eq? zero? string=?`
    - eg. type predicates like `pair?`
  - names of object procedures (that aren't lists) start with `type-`

        - eg. `string-append char-append`
    - converting functions are written as `type1→type2`
        - eg. `vector→list`
    - functions with side effects end in `!`
        - eg. `set! vector-set!`
- assignment in Scheme can be done with `set!` and other assignment procedures

Stack example:

```
(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 "invalid arg"])))))

(define mystack (make-stack))
(mystack 'empty?) ⇒ #t
(mystack 'push! 'a)
(mystack 'push! 'b)
(mystack 'pop!)
(mystack 'top) ⇒ 'a
```

Function Evaluation

- function application has the syntax `(procedure arg1 ... argn)` :
    - all Scheme structured forms are written in *prefix notation*
        - the arguments are evaluated before calling the procedure
            - unlike macros like `if` or `quote` which do not evaluate all arguments
        - no complicated rules regarding precedence or associativity in this way
        - eg. `(+ 1/2 1/2)` gives 1, `(/ 1.5 3/4)` gives 2.0, `(+ (+ 2 2) (+ 2 2))` gives 8
    - note that Scheme is free to evaluate the subexpression arguments in any order
    - note that `()` is thus an error, and `'()` should be used for empty lists
    - `eval` is a function, *not* a macro, that takes one argument and evaluates the expression

        - eg. take a program, change it, and `eval` it
- a top-level definition has the syntax `(define var definition)`
    - can be used for functions or objects

Simple Scheme functions:

```
(define square
  (lambda (n)
    (* n n)))

; equivalently, using syntactic sugar
(define (square n) (* n n))

(define reciprocal
  (lambda (n)
    (if (= n 0) "oops!" (/ 1 n))))
```

Scope

- like C, Scheme has shadowed variables:
    - but the scope of a variable defined by `let` does not extend into its initial value
    - nor into the initial values of other identifiers in the same `let`
    - this is due to the fact that `let` is defined in terms of `lambda`

Expanding `let` expressions:

```
(let ([x 3])
  (let ([x (+ x 5)]
        [y (+ x 2)]) ; x here is bound to 3
    (* x y)))
; expands to
((lambda (x)
  ((lambda (x y) (* x y))
    (+ x 5) (+ x 2)) 3))

(let* ([ x 3]
       [y (+ x 3)])
  (* x y))
; expands to
(let ([x 3])
  (let ([y (+ x 3)])
    (* x y)))
```

Additional Special Forms

- the `let` expression is used to establish the value of variables:
  - has the syntax `(let ((var expr) ...) body1 body2 ...)`
    - the variables become bound to value of the expressions
    - note that all the variables are bound at the *same* time
      - unless `let*` is used instead of `let`
      - similar to OCaml's `let in` vs. `let and`
    - return the value of the last body expression
    - syntactic sugar for a lambda with a call
  - these nest in the usual way, using static scoping
  - the `let` expression evaluates to a value, *without* creating a top level definition
    - unlike a `define` expression which creates a top level definition and does not evaluate
  - with `letrec` instead of `let` , the variables are visible within the body as well as expressions

Using `let` expressions:

```
(let ((x 2) (y 3))
  (+ x y)) ⇒ 5

(let ((a (* 4 4))) (+ a a)) ⇒ 32

(let ([f +] [x 2] [y 3]) ; using brackets instead of parens for clarity
  (f x y)) ⇒ 5

(let ([x 1])
  (let ([x (+ x 1)]) ; inner x binding shadows outer x binding
    (+ x x))) ⇒ 4

(let ([x (+ 3 5)])
  (let ([x (* 3 6)]) x) ; this shadowed, first expression is thrown away
  x) ⇒ 8 ; evaluates the value of last expression

(let ([x 'a]) (cons x x)) ; easier to understand order
; equivalent syntactically to
((lambda (x) (cons x x)) 'a) ; order is less clear
```

- the `lambda` expression is used to create a new anonymous function:
  - has the syntax `(lambda (var ...) body1 body2 ...)`
    - yields a function with formal parameters defined by the var list

    - evaluates body expressions left to right, and yields the value of last one
  - `let` is actually defined in terms of the `lambda` expression:
    - `(let ((var expr) ...) body1 body2 ...)` is equivalent to
    - `((lambda (var ...) body1 body2 ...) expr ...)`

Using `lambda` expressions:

```
((lambda (x) (+ x x)) (* 3 4)) ⇒ 24 ; 12 + 12 = 24

(let ([double (lambda (x) (+ x x))]) ; binding lambda with let expression
  (list (double (* 3 4))
        (double (/ 99 11))
        (double (- 2 7)))) ⇒ (24 18 -10)

(let ([double-any (lambda (f x) (f x x))])  ; 2 argument lambda
  (list (double-any + 13)
        (double-any cons 'a))) ⇒ (26 (a . a))

; nesting lambda and let
(let ([f (let ([x 'sam])
           (lambda (y z) (list x y z)))]) ; x is free variable here
  (let ([x 'not-sam])
    (f 'i 'am))) ⇒ (sam i am) ; uses bindings from when function was created

((lambda (x) (+ x 1)) y) ; x is a bound variable
                         ; y is a free variable (using lexical scoping)
```

- the parameter specification in a `lambda` expression has multiple forms:
  1. a proper list of n variables
     - n actual parameters must be supplied
  2. a single variable
     - any number of actual parameters is valid, all parameters become packed into a single list
     - 0 or more arguments
     - eg. `(define list (lambda x x))`
  3. an improper list of variables `(var1 ... varN . varR)`
     - at least n actual paremeters must be supplied
     - `varR` is a *rest* parameter holding the remaining parameters

Exploring `lambda` parameters:

```
(let ([f (lambda x x)])
  (f 1 2 3 4)) ⇒ (1 2 3 4)
```

```
(define list (lambda x x)) ; list function

(let ([g (lambda (x . y) (list x y))])
  (g 1 2 3 4)) ⇒ (1 (2 3 4))

(let ([h (lambda (x y . z) (list x y z))])
  (h 'a 'b 'c 'd)) ⇒ (a b (c d))
```

- in definitions where the expression is a `lambda` expression, Scheme provides the following syntactic sugar:
  - `(define var0 (lambda (var1 ... varN) ...))` can be abbreviated as
    - `(define (var0 var1 ... varN) ...)`
  - `(define var0 (lambda varR ...))` can be abbreviated as
    - `(define (var0 . varR) ...)`
  - `(define var0 (lambda (var1 ... varN . varR) ...))` can be abbreviated as
    - `(define (var0 var1 ... varN . varR) ...)`

Common syntactic sugar for lambda expressions:

```
(define cadr (lambda (x) (car (cdr x)))) ; builtin
(define (cadr x) (car (cdr x)))

(define cddr (lambda (x) (cdr (car x)))) ; builtin
(define (cddr x) (cdr (car x)))

(define list (lambda x x))
(define (list . x) x) ; parameter list in shorthand
```

Another `lambda` example:

```
(define doubler (lambda (f)
  (lambda (x) (f x x))))

(define double (doubler +))
(double 13/2) ⇒ 13

(define double-cons (doubler cons))
(double-cons 'a) ⇒ (a . a)
```

- the *named* `let` expression has the syntax `(let name ((var expr) ...) bodies)` :
  - binds `var` to `expr` within the specified bodies

  - binds `name` within the bodies to a procedure that may be called to recur
    - the arguments to the procedure become the new values for the variables
  - equivalently `((letrec ((name  (lambda (var ...) bodies))) name) expr ...)`

Conditionals

- the `if` expression has the syntax `(if test consequent alternative)` :
  - evaluates test, if true, evaluate and return consequent, otherwise evaluate and return alternative
  - note that in Scheme, only `#f` is considered false and all other objects are considered true
  - `if` is a macro that only evaluates part of its arguments, *unlike* normal procedures
    - eg. `(if #t 97 (/ 1 0))` does not give an error
    - thus `if` must be defined as a special form ie. macro instead of as a normal function
    - `or` and `and` are also macros (that do not evaluate all arguments), while `not` is a function
- `or` expression has the syntax `(or expr ...)` :
  - evaluate expressions left to right, stop and return the first expression that yields true
    - if none of them yield true, return false
    - if zero expressions, return false
  - eg. `(or #f 1 3)` gives 1
  - eg. `(or (getenv "PATH"))` gives either a path string or `#f`
- `and` expression has the syntax `(and expr ...)` :
  - evaluate expressions left to right, stop and return false when the first expression yields false
    - if none of them yield false, return the value of the last expression
    - if zero expressions, return true
  - note that unlike C, these expressions can yield values of subexpressions, instead of just true or false
  - eg. `(and #f 2 3)` gives `#f`
- `not` function negates its argument
- the `cond` expression allows for multiple test and alternative expressions
  - has the syntax `(cond (text expr) ... (else expr))`

        - `else` clause may be omitted
    - convenient way to write multiple if branches
    - because Scheme is dynamically typed, a nonexhaustive `cond` expression may return *nothing*

Using conditionals:

```
(define abs (lambda (n)
  (if (< n 0) (- 0 n) n)))

(define reciprocal (lambda (n)
  (and (not (= n 0)) (/ 1 n)))) ; #f if n is zero, 1/n otherwise

(define reciprocal (lambda (n)
  (if (and (number? n) (not (= n 0))) ; check arg is a number
    (/ 1 n)
    (assertion-violation 'reciprocal "improper argument" n))))

(define safe-cdr (lambda (x)
  (if (null? x) '() (cdr x))))

(define sign (lambda (n)
  (cond [(< n 0) -1]
        [(> n 0) +1]
        [(= n 0) 0]))) ; or [else 0] or [#t 0]
```

Syntactic Extensions

- new **syntactic extensions** ie. **macros** are designed using a `define-syntax` expression, similar to a `define` expression
    - unlike normal procedures, macros themselves do not have a value and do not always evaluate all their arguments when evaluating
    - allows for a simple core that lets programmers extend the syntax
    - eg. `(if ...)`, `(lambda ...)`, can add additional ones like `(try ...)`

Example builtin syntactic extensions:

```
(define-syntax let
  (syntax-rules () ; list of auxiliary keywords
    ; list of substitutions to apply at compile-time
    [(let ([name val] ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...) val ...)]
    [(let tag ([name val] ...) body1 body2 ...)
```

```
     ((letrec ([tag (lambda (name ...) body1 body2 ...)]) tag) val ...)]))

(define-syntax and
  (syntax-rules ()
    [(and) #t] ; identity case
    [(and e) e]
    [(and e1 e2 ...)
     (if e1 (and e2 ...) #f)])) ; recursive rule definition
```

Illustrating some problems with `or` macros:

```
(define-syntax or
  (syntax-rules ()
    [(or) #f]
    [(or e) e]
    [(or e1 e2 ...)
     (if e1 e1 (or e2 ...))])) ; issue with this or macro!

(or (getenv "PATH") "/bin")
(if (getenv "PATH") (getenv "PATH") "/bin")
; above substitution unecessarily evaluates getenv twice
; if the first expression returns true!

(define-syntax or
  (syntax-rules ()
    [(or) #f]
    [(or e) e]
    [(or e1 e2 ...)
     (let ([t e1])
       (if t t (or e2 ...)))])) ; save evaluation of e1 in t

; a more subtle issue, the problem of *capture*:

(let ([x "/bin"])
  (or (getenv "PATH") x)) ; should give the same result as original or

(let ([x "/bin"])
  (let ([x (getenv "PATH")])
    (if x x x))) ; always gives getenv result, since macro has *captured* x

; Scheme solves this by examining variable scope *before* macro expansion:
; (AKA *hygienic* macros, cleaner than C/C++ macros)
```

```
(let ([x1 "/bin"])
  (let ([x2 (getenv "PATH")])
    (if x2 x2 x1))) ; Scheme internally performs a similar substitution
```

Procedures vs. macros:

```
(define (change+/- prog)  ; a procedure, so (change+/- (+ 1 2)) gives an error
  (cons '- (cdr prog)))   ; (change+/- '(+ 1 2)) gives unevaluated (- 1 2)

(defmacro change+/ (prog) ; a macro, so (change+/- (+ 1 2)) evaluates properly
  (cons '- (cdr prog)))
```

Recursion

Recursive examples:

```
(define length (lambda (ls)
  (if (null? ls) 0 (+ (length (cdr ls)) 1))))

(define member (lambda (x ls)
  (cond [(null? ls) #f]        ; false if object not found
        [(eqv? (car ls) x) ls] ; otherwise list whose car equals x
        [else (member x (cdr ls))])))

(define remove (lambda (x ls)
  (cond [(null? ls) '()]
        [(eqv? (car ls) x) (remove x (cdr ls))]
        [else (cons (car ls) (remove x (cdr ls)))])))

(define (reverse ls ; O(n^2), non-tail recursive
  (if (null? ls)
    ls
    (append (reverse (cdr ls)) (list (car ls)))))

(define (revapp ls a)
  (if (null? ls)
    a
    (revapp (cdr ls) (cons (car ls) a))))
(define (reverse ls)
  (revapp ls '()))

; a more general map accepting > 1 arg functions and > 1 lists is builtin
```

```
(define map (lambda (p ls)
  (if (null? ls) '()
    (cons (p (car ls)) (map p (cdr ls))))))
```

Using `letrec` for local bindings:

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

; letrec for mutual recursion
(letrec ([even? (lambda (x)
            (or (= x 0) (odd? (- x 1))))]
         [odd? (lambda (x)
            (and (not (= x 0)) (even? (- x 1))))])
  (list (even? 20) (odd? 20))) ⇒ (#t #f)
```

Using named `let` as syntactic sugar for auxiliary functions:

```
; like the base let expression, named let makes the *order*
; more clear when using auxiliary helper functions

(define (reverse ls)
  (let revapp ([ls ls] [a '()]) ; named let defines an internal, recursive function
    (if (null? ls)
      a
      (revapp (cdr ls) (cons (car ls) a)))))

(define factorial (lambda (n) ; recursive
  (let fact ([i n])
    (if (= i 0) 1 (* i (fact (- i 1)))))))

(define factorial (lambda (n) ; accumulator
  (let fact ([i n] [a 1])
    (if (= i 0) a (fact (- i 1) (* a i))))))

; non tail recursive
(define fibonacci (lambda (n)
  (let fib ([i n])
    (cond [(= i 0) 0]
          [(= i 1) 1]
          [else (+ (fib (- i 1)) (fib (- i 2)))]))))

; tail recursive, maintain accumulators for current and preceding fib
```

```
(define fibonacci (lambda (n)
  (if (= n 0) 0
    (let fib ([i n] [a1 1] [a2 0])
      (if (= i 1) a1 (fib (- i 1) (+ a1 a2) a1))))))
```

Continuations

- continuations are a controversial feature of Scheme:
  - proponents say "essence of Scheme"
  - opponents say "a big disaster, too low-level"
- as a Scheme expression is evaluated, the implementation must keep track of:
  1. what to evaluate
     - ie. the instruction pointer `ip` or `%rip`
  2. what to do with the value
     - ie. the environment pointer `ep` or `%rsp`
       - which contains the return address ie. caller's instruction pointer
     - scheme allows the continuation of an expression to be captured using the procedure `call/cc`
  - also true of OCaml, C, Java
  - a **continuation** is an `(ip, ep)` *pair*
    - allows a program to *manipulate* what it'll do next, and what context it'll do it in
    - in Scheme, can ask the interpreter to create a continuation object using `call/cc`
    - eg. can use continuations to escape from deeply nested recursions in order to avoid uneccesary work, like Prolog's cut
- `call/cc` AKA `call-with-current-contination` takes a procedure `p` of one argument:
  1. it constructs a *concrete* representation `k` of the current continuation
  2. *passes* `k` to `p` , ie. `(p k)`
     - `k` acts as a function taking an expression
     - flow of control passes back to wherever `call/cc` was
     - that `call/cc` returns the value of the expression
  3. returns whatever `p` returns
  - ie. restores `(ip, ep)` from `k` , like a `goto` on steroids
    - in C/C++, a `goto` affects only the instruction pointer
- usages of continuations:
  - *escape* from a function to its caller:
    - create a continuation and give it to the main code

      - main code can use the continuation whenever it wants
    - this is the basis for exception handling
    - eg. `try { main code } except (E e) { catching code }`
  - jump *back* into a function that has already returned
    - eg. *green threads* or multithreading with just one CPU ie. one ip
      - use continuations to switch from one thread to another when one thread voluntarily gives up the CPU
    - in imperative languages, harder to manually create continuations
      - compilers will implicitly provide support for continuations for threads, etc.
    - C/C++ does provide `setjmp` library with one continuation functionality
      - `setjmp` saves important registers into a `jmp_buf`, ie. `try` operation
      - `longjmp` restores registers from a `jmp_buf` and copies `val` into return register, acting as a nonlocal goto, ie. `throw val`
      - *but* unlike Scheme continuations, can only jump to yourself or one of your calling ancestors
    - C/C++ also provide `ucontext` library with more powerful continuations that allow for switching stacks
    - Python has iterators and `yield`

Using `call/cc`:

```
(call/cc (lambda (k) (* 5 4))) ⇒ 20 ; k never invoked
(call/cc (lambda (k) (* 5 (k 4)))) ⇒ 4
(+ 2 (call/cc (lambda (k) (* 5 (k 4))))) ⇒ 6

; exception handling:
; product with a nonlocal exit from recursion:
;   given a very long list of potentially very large numbers,
;   want to exit immediately if any are ever 0
(define product (lambda (ls)
  (call/cc (lambda (break)
    (let f ([ls ls])
      (cond [(null? ls) 1]
            [(= (car ls) 0) (break 0)] ; break if element is ever 0
            [else (* (car ls) (f (cdr ls)))])))))))

; jumping back into factorial
(define retry #f)
(define factorial (lambda x)
  (if (= x 0)
    (call/cc (lambda (k) (set! retry k) 1))
    (* x (factorial (- x 1)))))
```

Green threads example using continuations:

```
(define thread-list '())

(define (new-thread thunk) ; thunk is 0-arg function to run on new thread
  (set! thread-list (append thread-list (list thunk))))

(define (start)
  (let ([next-thread (car thread-list)])
    (set! thread-list (cdr thread-list))
    (next-thread)))

(define (yield)
  (call/cc (lambda (k)
    (new-thread (lambda () (k #f))) ; make sure continuation takes 0 args
    (start))))

(new-thread (lambda () (let f () (display "h") (yield) (f))))
(new-thread (lambda () (let f () (display "i") (yield) (f))))
(new-thread (lambda () (let f () (newline) (yield) (f))))
(start)
; gives:
; hi
; hi
; hi
; ...
```

- in **continuation passing style (CPS)**, we replace implicit continuations with explicit procedures
  - more complicated expressions, but allows procedures to pass more than one result to its continuation
  - an alternative to using the `call/cc` continuation primitive

Converting to CPS:

```
(letrec ([f (lambda (x) (cons 'a x))]
         [g (lambda (x) (cons 'b (f x)))]
         [h (lambda (x) (g (cons 'c x)))])
  (cons 'd (h '()))) ⇒ (d b a c)

; in CPS
```

```
(letrec ([f (lambda (x k) (k (cons 'a x)))]
         [g (lambda (x k) (f x (lambda (v) (k (cons 'b v)))))]
         [h (lambda (x k) (g (cons 'c x) k))])
  (h '() (lambda (v) (cons ('d v))))) ⇒ (d b a c)

; using CPS to pass back multiple results
(define car&cdr (lambda (p k) (k (car p) (cdr p))))

(car&cdr '(a b c) (lambda ( x y) (list y x))) ⇒ ((b c) a)
(car&cdr '(a b c) cons) ⇒ (a b c)
(car&cdr '(a b c a d) member) ⇒ (a d)

; rewriting product to use CPS instead of call/cc
(define product (lambda (ls k)
  (let ([break k])
    (let f ([ls ls] [k k])
      (cond [(null? ls) (k 1)]
            [(= (car ls) 0) (break 0)]
            [else (f (cdr ls) (lambda (x) (k (* (car ls) x))))])))))
```

Using Functional Programming

- OCaml, as a functional language, lends itself to recursion very easily
    - `rec` keyword defines something in terms of itself

Recursive reverse list in OCaml:

```
let rec reverse = function
  | [] -> []
  | h::t -> (reverse t) @ [h]
```

- this code is inefficient, because the concatenation operator has linear time
    - function will run in $O(n^2)$
- use an **accumulator** to speed up the recursive function
- in addition, solve the reverse problem by solving a more general variant:
    - how to compute the reverse of a list concatenated with another list?
        - use an accumulator and `::` to create an $O(n)$ function

Redefining reverse list through another variant:

```
(* revapp [] [3;9;2] => [3;9;2] *)
let rec revapp l a =
  match l with
    | [] -> a
    | h::t -> revapp t (h::a)

let reverse l = revapp l []
```

Using `function` syntactic sugar and local declarations:

```
let reverse l =
  let rec apprev a = function
    | [] -> a
    | h::t -> apprev (h::a) t
  in apprev [] l
```

Minimum value in a list:

```
let rec minlist = function
  | h::t ->
      let m = minlist t
      in if h < m then h else m
  | [] -> ??? (* identity element for min? *)
```

- this function only works on integers, and there is no identity element for the minimum value

Delegate identity element and comparison functions to the user:

```
(* minlist (<) 1000000 [3;5;7;-20] => -20 *)
let rec minlist lt id = function
  | h::t
      let m = minlist lt id t
      in if lt h m then h else m
  | [] -> id
```

Appendix

- can build a pipeline operator to deal with many nested matches

Defining and using a pipeline operator:

```
let (>>=) v f = match v with
  | None -> None
  | Some x -> f x

let div_abcd a b c d =
  let div_some y x =
    if y = 0 then None
    else Some (x / y) in
  match div_some a b with
    | None -> None
    | Some ab -> match div_some ab c with
      | None -> None
      | Some abc -> match div_some abc d with
        | None -> None
        | Some abcd -> Some abcd

let div_abcd a b c d =
  let div_some_rev y x =
    if y = 0 then None
    else Some (x / y) in
  Some a
  >>= div_some_rev b
  >>= div_some_rev c
  >>= div_some_rev d
```

- a useful functional programming paradigm is **continuation-passing style (CPS)**
  - allows us to dictate what a function does with its return value, instead

of immediately returning it

Implementing the previous operation with CPS:

```
let plus_cps x y k = k (x + y)

let div_avcd_cps a b c d k =
  let div_some_rev y k2 x =
  if y = 0 then
    None
  else k2 (x / y) in
  div_some_rev b (div_some_rev c (div_some_rev d k) a)
```

Imperative Programming

Java

- **Java** is an object-oriented, imperative programming language
  - with **object-oriented** programming, programs are designed using **objects**
  - each object is a bundle of data that know how to do things to themselves
  - specifically, the data is stored in **fields** and the operations are **methods**
    - there can be multiple **instances** of a class of objects
  - with **imperative** programing, programs focus on assignment and iteration
    - variables have current states

Barebones `point` class:

```java
public class Point {
  // field definitions
  private int x, y;
  private Color myColor;

  // method definitions
  public int currentX() { return x; }
  public int currentY() { return y; }
  public void move(int newX, int newY) {
    x = newX;
    y = newY;
  }
}
```

History

- like Linux, in the Solaris OS, kernel and apps are written in C/C++
  - mostly ran on SPARC architecture
  - placed an emphasis on multiprocessor workstations and servers
- Solaris developers sought to create **Internet of Things (IOT)** applications, eg. running programs on toasters, refrigerators, etc.
  - *issues*:

        - C/C++ crashes a lot
            - unacceptable in a consumer environment
        - suboptimal multiprocessing support with C/C++
        - applicance makers didn't want to use SPARC
            - C/C++ code thus would have to be portable
            - requires compiling different versions for different architectures
        - slow software updates due to slow networking
        - C/C++ is *overly* complicated
    - *solution*:
        - simplify the language into the language C+- AKA C more or less
        - take ideas from competition, eg. Smalltalk, an object-oriented, dynamically checked language
- Smalltalk *features*:
    - interpreted runtime checking and exception handling for errors
        - in C/C++, exception handling takes a sideline for performane
    - much simpler than C++
    - operated internally via **bytecodes**, representation of source code that is smaller, closer to source, and machine independent
    - garbage collector, more reliability
- Smalltalk *issues*:
    - weird syntax
    - single threaded
    - bad marketing
- Solaris programmers glued together C+- with Smalltalk:
    - syntax of C/C++
    - better multithreading support
        - good for server side applications
    - better marketing
    - resulting language originally called Oak, became Java
        - marketed by writing a browser
- browser history aside:
    - Mosaic was the first successful browser, created for academic use
        - written in C++, which lead to frequent crashes
        - wasn't extensible except by changing source code and recompiling
    - eventually became Netscape, Firefox, Internet Explorer, etc.
- HotJava, an extensible browser was writen in order to to market Java:
    - accepted Java bytecode over the net to run in a sandbox
    - not a very successful browser, but sold Java well

Roots of Java Basics

- static type checking at compile time
    - extra reliability

- variables are always *initialized* to zero
    - extra reliability, portability, and reproducibility at the cost of performance
- a fixed set of primitive types, with very *well-defined* properties:
    - unlike C, whose documentation did *not* specify implementation
    - portability over performance
    - eg. `fmuladd`, a fused multiply add instruction
        - used for faster and more accurate dot products
        - because not every machine supported this instruction, Java required completing the multiply add in separate steps, even though this was slower and more inaccurate than the fused instruction
        - now, Java is not as ironclad in such rules and `fmuladd` is supported
- *strict* order of evaluation:
    - eg. `f(x, y) + g(h)`
    - in C, the ordering is implementation defined
        - and if there are competing side effects, the operation is actually undefined
    - in Java, always evaluated left-to-right
    - portability at the cost of compilation optimizations
- reference types:
    - every object implemented as if by a pointer to its value
    - thus all have the same pointer representation
    - allows generic types to work
- arrays are reference types:
    - unlike C:
        - subscript is always checked
        - arrays are all on the heap, instead of the stack, allowing for garbage collection
        - size is fixed when allocated
        - contents initially zero
        - typed, member types are always known
    - safety over performance
- object-oriented, but only supporting *single inheritance*
    - restrictive, but Java allows implementing multiple interfaces
        - interfaces are the duck typing of Java

Expressions and Statements

- basic primitives: `int char double boolean void null`
    - eg. `10 -1 'a' '\n' 3.4 1.3e2 1E2`
        - Java uses Unicode charset to interpret integral `char` values, not

only ASCII

  - other primitives such as `byte short long float` are specialized number sizes
  - Java specification defines exactly what the types are:
    - `int` is set of 32-bit, 2s-complement binary numbers
    - `double` is set of 64-bit, IEEE floating point numbers
    - `boolean` includes only the constants `true false`
    - `null` includes only special constant `null` that can be assigned to any variable of any reference type
      - different from C/C++ which allow for using integer 0 to act as a null pointer
- any value that is not of a primitive type is a reference to an object:
  - these **reference types** are constructed and have three kinds:
    - class names, references to objects from a class, references to objects from a subclass
    - interface names
    - array types
- unlike ML, there is no primitive type for strings:
  - instead, has a predefined `String` class
  - an object of the `String` class *contains* a string of `char` values
  - `"foobar"` is a string constant, or more accurately an instance of the `String` class
- integer arithmetic operators:
  - `+- * / %`
  - `/` performs integer division
  - `-` can be used as a unary operator for negation
- real arithmetic operators:
  - `+- * /` are overloaded for type `double`
  - `-` overloaded for negation on type `double`
- special syntax for `String` objects:
  - no need to use `new` keyword
  - `+` concatenates two `String` values
    - when either operand is a `String`
    - if one operand is not a `String` , it is coerced to a string before concatenation
      - Java knows how to coerce all primitive types to `String`
      - for reference-type values, it calls the `toString` method
    - eg. `"1"+2+3` gives `"123"` , `1+2+"3"` gives `"33"`
- comparison operators:
  - `< ≤ ≥ >` can be used on any numeric types
  - equality operators `= ≠` can be used on values of any type
    - only considered equal if they refer to the *same object*

        - eg. two strings with the same text may not be considered equal if they are different instances
- boolean operators:
    - `&& ||` , short-circuiting, only work on operands of type `boolean`
    - `!` complements
    - ternary syntax `a ? b : c`
- operators with side effects:
    - different from functional languages, which had mostly *pure* operations
    - assignment operator `a = b`
        - *rvalues* can be anything with a value
        - *lvalues* must have a *memory location*, eg. variables, fields, parameters
        - note that some lvalues can be rvalues
            - most languages, like Java, use *context* to distinguish between lvalues and rvalues
        - assignment has the side effect of changing the value stored in the lvalue
        - right associative, `a=b=c` is equivalent to `a=(b=c)`
    - note that an expresion with side effects still has a value
        - eg. `a+(b=c)`
    - compound assignment `a+=b a-=b a*=b a/=b a%=b`
    - shorthand `++a --a a++ a--`
        - `++a` and `a++` have the same side effect, but different values
- method calls:
    - **instance methods** require an object to operate on:
        - `s.length() s.equals(r) s.toUpperCase() s.charAt(3)`
        - reference of the object is passed to the method
        - if the reference is left out, the called method will operate on the same instance as the caller
    - **class methods** do not require a particular object to operate on:
        - `String.valueOf(1==2)` gives `"false"`
- object creation:
    - the `new` keyword is used to create a new object with a constructor instance method
    - eg. `new String()`
    - note that Java uses garbage collection to free up allocated memory, ie. no need to explicitly delete
- general coercion rules:
    - `null` to any reference type
    - any value to `String` for concatenation
    - `char` to `int` before any operators
    - `int` to `double` if mixed

- statements:
  - an expression followed by a semicolon is a **statement**
  - unlike C/C++, an expression in a statement must have a side effect
    - eg. `x == y;` is not a statement in Java
  - **compound statements** are any number of statements enclosed in braces
    - also serves as a block for scoping
  - **declaration statements** introduce a local variable, following block scoping
    - eg. `Point p;` or `int temp = 3;`

Class Definitions

- in Java, everything in a class is defined using an **access specifier**
  - eg. `public` and `private`
  - constructors have no return type, named after the name of the class
    - when no constructors are defined, Java creates a zero-parameter constructor automatically
- in addition, freestanding functions cannot be written, functions must always be a method of some class
- `static` methods are class methods that operate only on explicit parameters and have no fields

Example linked list of integers class:

```
public class ConsCell {
  private int head;
  private ConsCell tail;

  public ConsCell(int h, ConsCell t) { head = h; tail = t; }

  public int getHead() { return head; }
  public int getTail() { return tail; }
  public void setTail(ConsCell t) { tail = t; }

  public static int length(ConsCell a) {
    int len = 0;
    while (a) {
      len++;
      a = a.getTail();
    }
    return len;
  }
  public static ConsCell cons(int a, ConsCell b) {
    return new ConsCell(a, b);
  }
}

ConsCell a = null;
ConsCell b = ConsCell.cons(2,a);
int x = ConsCell.length(a) + ConsCell.length(b);
```

- is there a way to obtain the above functionality without class methods?
  - after all, length and cons operations should be embedded for a specific list
  - issue in the above implementation since an empty list is `null`
    - would not be able to call instance methods on a `null` object
    - create a separate class to abstract away the `null` case

Linked list of integers without class methods:

```
public class IntList {
  private ConsCell start; // first in list or null

  public IntList(ConsCell s) { start = s; }

  public int length() {
    int len = 0;
    ConsCell cell = start;
    while (cell) {
      len++;
      cell = cell.getTail();
    }
    return len;
  }
  public IntList cons(int h) {
    return new IntList(new ConsCell(h, start));
  }
}

IntList a = new IntList(null);
IntList b = a.cons(2);
int x = a.length() + b.length();
```

- references and pointers:
  - `start` is a reference to a `ConsCell` object
  - the constructor does not take an object, but actually the address of the

    object
        - and `start` stores the address of the object, not a copy of the object
        - thus Java references are implemented as pointers, but from the program's perspective, it is just a unique identifier for an object

Example C++ statements:

```
IntList* p;
p = new IntList(0);
p->length();
p = q;
```

Equivalent Java statements:

```
IntList p;
p = new InstList(null);
p.length();
p = q;
```

Compilation

- to run code in a Java program, a **main method** is needed

Example Driver class:

```
public class Driver {
  public static void main(String[] args) {
    IntList a = new IntList(null);
    int x = a.length();
    System.out.println(x);
  }
}
```

- with the three source files `ConsCell.java` , `IntList.java` , and `Driver.java` :
  - `javac Driver.java` will run the compiler on all three classes
  - creates three new files `ConsCell.class` , `IntList.class` , `Driver.class`
  - `java Driver` will run the program

Polymorphism

- polymorphism allows for relationships between classes in Java
- under the surface, Java keeps track of several properties for every class:

1. the interfaces it implements
2. the methods it is obliged to define
3. the methods that are defined for it
4. the fields that are defined for it

- implementing an interface affects (1) and (2)
- extending a class affects all properties
- note that the `abstract` keyword for methods allows for adding method obligations for a class without using an interface

Interfaces

- an **interface** in Java is a collection of method prototypes:
  - prototypes include method return type, name, and parameters, but not method bodies
  - all prototypes considered as `public` by Java
  - another class then `implements` the interface, promising to provide a public implementation of all methods in the interface
    - more than one interface can be implemented
    - an interface can extend other interfaces
  - using interfaces allows for using the name of the interface as a type in a type declaration
    - allows for polymorphic methods and functionality
    - Java can call interface methods on an object without knowing exactly what class of object it is
  - also allows for interface-implementation separation

A Drawable interface example:

```java
public interface Drawable {
  void show(int xPos, int yPos);
  void hide();
}

public class Icon implements Drawable {
  public void show(int x, int y) {...}
  public void hide() {...}
  ...
}
public class Square implements Drawable, Scalable {...}

Drawable d;
d = new Icon(...);
d.show(0, 0);
d = new Square(...);
```

```
d.show(10, 0);
```

A polymorphic function:

```
static void flashoff(Drawable d, int k) {
  int i = 0;
  while (i++ < k) {
    d.show(0,0);
    d.hide();
  }
}
```

Inheritance

- when a class `extends` a base class, all of the **base class**'s fields and methods are *inherited* by the **derived class**
  - ie. like a subtype, a derived class is a subset of the base class, but offers a superset of the base class's functionality
  - essential feature of object-oriented languages
  - like interfaces, allows for polymorphism
    - a base type reference can refer to objects of a derived class as well
  - a `protected` access specifier allows definitions to be visible within the class and any derived classes extending the class
    - as well as other classes in the same *package*
  - note that every class implicitly inherits from the base `Object` class
    - provides useful methods such as `toString`
  - inherited definitions can also be *overrided* to allow for specification
    - overriding occurs on inherited methods when the name and types are identical
  - creates a *hierarchy* of classes that can help solve problems

Inheritance example:

```
public class Graphic {
  protected int x, y;
  protected int width, height;
  public void move(int newX, int newY) { x = newX; y = newY; }
}

public class Icon extends Graphic {
  private Gif image;
  public Gif getImage() {return image; }
}
```

```java
public class Label extends Graphic {
  private String text;
  public string getText() {return text; }
}
```

- unlike C++, Java inheritance only allows for extending a *single* base class
  - prevents the complexity and ambiguity possible in *diamond inheritance*
  - some workarounds for *multiple* inheritance:
    - classes can implement as many interfaces as desired
    - **forwarding** can be used where a single class contains references to multiple other objects
      - can forward methods to specific objects, but not actually implement any new functionality

Generics

- before generics in Java, type variables would have been defined using the base class `Object`
  - has issues with compile-time type checking, as well as inconvenient type casting
    - eg. have to use wrapper classes for primitives
- **generics** are Java parametrized classes, interfaces, methods, and constructors
  - uses angle bracket syntax

Inconveniences when using `Object` as type variables:

```java
public class ObjectNode {
  private Object data;
  ...
}
public class ObjectStack {
  private ObjectNode top = null;
  ...
}

ObjectStack s1 = new ObjectStack();
s1.add(new Integer(1)); // int is not an object, must use Integer wrapper class
int i = ((Integer) s1.remove()).intValue(); // type cast and unwrap of Integer
```

Using generics instead:

```java
public class Node<T> {
  private T data;
  private Node<T> link;
  ...
}
public class Stack<T> {
  private Node<T> top = null;
  public void add(T data) {
    top = new Node<T>(data, top);
  }
  public T remove() {
    Node<T> n =top;
    top = n.getLink();
    return n.getData();
  }
}

Stack<Integer> s1 = new Stack<Integer>(); // wrapper class here
s1.add(1); // but int parameter still accepted due to *boxing coercion*
int s = s1.remove(); // *unboxing coercion*
```

- although a type may be a subtype of another, one of its generic types may not be a subtype of one of the other generic types
  - eg. `List<String>` is *not* a subtype of `List<Object>`, although `String` is a subtype of `Object`
    - since `List<Object>` supports *more* operations, ie. more kinds of objects can be passed to its `add` method
  - if it were, uncastable / improper objects could be added to the list through aliasing
  - the problem stems from the fact that `List` is a *mutable* data type
  - however this causes issues in generic methods that should otherwise work:
    - Java allows for **wildcard types** as an unknown type for these situations
      - wildcards can be *bounded* for more flexibility
      - wildcards are just shorthand for type parameters
- rules when using type parameters:
  - in the *read-only case*, can operate on all subclasses only:
    - consider the generic types `Arr<T>` and `Arr<U>`, where `T < U`
    - `Arr<? extends U> au = at` is valid
      - the resultant list `au` is a **producer**, ie. allows for pulling elements out but not adding in
      - effectively read-only
  - in the opposite *write-only case*, can operate on all superclasses only:
    - consider the generic types `Arr<T>` and `Arr<V>`, where `T < V`
    - `Arr<? super T> av = at` is valid

        - the resultant list `av` is a **consumer**, ie. allows for adding elements in but not pulling out
        - effectively write-only
    - ie. PECS mnemonic: producer extends, consumer super
    - if both a producer and consumer is required, only an exact type-match without type parameters will work

Trouble with subtypes and generics:

```
// String is a subtype of Object
List<String> ls = ...;

// reference / pointer assignment, not object copy
List<Object> lo = ls; // problematic, allows for subverting types
                      // therefore, Java doesn't allow it, compile-time error
                      // List<String> ls = lo is not allowed either


lo.add(new Thread()); // OK on its own
String s = ls.get();  // OK on its own

// altogether, the code block would have aliasing issues, and does not compile
```

Using wildcards:

```
// void printList(List<Object> lo) { // no good! will not work with List<String>
void printList(List<?> l) {
  for (Object o : l) // every type is an Object
    System.out.println(o);
}

printList(ls);
printList(lo);
```

Bounded wildcards:

```
// A is a subtype of B
// Shape < Object
// Rectangle < Shape
// Ellipse < Shape
// Shape < Shape

void displayShapes(List<?> l) {
  for (Shape s : l) // no good! assigning Object to Shape
    displayShape(s);
}
```

```
void displayShapes(List<? extends Shape> l) { // bounded wildcard, (? < Shape)
// void displayShapes(List<Shape> l) { // also OK, would fix consumer problem
  for (Shape s : l)      // now OK
    displayShape(s);     // l is producing, knows (? < Shape)
  // l.add(new Shape()); // no good! l is consuming, cannot tell if (Shape < ?)
}
```

Explicit type parameters:

```
void convert(Object[] ao, List<Object> lo) { // does not work for Strings
  for (Object o : ao)
    lo.add(o);
}

void convert(Object[] ao, List<?> lo) {
  for (Object o : ao)
    lo.add(o); // no good! type mismatch
}

<T> void convert(T[] a, List<T> l) {
  for (T o : a)
    l.add(o); // now OK
}

// still not good enough, too restrictive
String[] as = ...;
List<Object> lo = ...;
convert(as, lo); // should be OK, but it's not, type mismatch

// bounded wildcard with type parameters
<T> void convert(T[] a, List<? super T> l) { // (T < ?)
  for (T o : a)
    l.add(o); // now OK, l is consuming
}
convert(as, lo); // now OK
```

Wildcard are shorthand for type parameters:

```
// ? stands for a type variable that's used only once
// ? is analagous to _ for OCaml
// full convert function
<T,U> void convert(T[] a, List<U super T> l) { // (T < U)
  for (T o : a)
```

```
    l.add(o);
}
```

- overall, type algebra used to build practical, polymorphic Java methods is more complicated
  - how is this type algebra implemented at runtime?
- one implementation approach used with Java, using **erasure**:
  - every object is represented by a pointer to the storage implementing the object
  - at the start of the storage is a **type tag**, eg. a pointer to a runtime representation of the type
  - for generic types `List<String>` as an instantiation of `List<T>` , is there a separate runtime representation?
    * erasure says no, all the type tag says is `List<T>`
  - runtime system *trusts* the compiler to have done the proper checking
  - *pros*:
    * simpilied runtime system
    * allows Java to be implemented atop simpler JVMs that don't support generics at all
    * only one copy of a class involving generics
  - *cons*:
    * more care around runtime casting must be taken

Casting with generic types:

```
Object o = ...;
String s = (String) o; // runtime check of type tag

List<Shape> ls = ...;
List<Rectangle> lr = (List<Rectangle>) ls; // type tag only says it is List<T>
// leads to more runtime checks later when processing list elements
// but casts should usually be avoided anyway
```

Duck Typing

- Java goes through a lot of trouble for its complicated type rules
  - like OCaml, which has complicated error messages for typing errors
  - some argue this tradeoff is not worth it
    * extensive typing is more trouble than it is worth
- instead, such programmers may be a proponent of **duck typing**:
  - a style of typing where the semantics of an object is defiend by its current set of methods, not its type
  - objects don't have types, all they have are methods
    * instead of worrying about what an object *is*, worry about what it

does

  - “if it waddles like a duck, and quacks like a duck, it is a duck”
  - used in Python, Ruby, Smalltalk

Duck typing:

```
#bad duck typing
if isADuck(o):
  print("is a duck")
  o.waddle()
  o.quack()

#good duck typing
try:
  o.waddle()
  o.quack()
except:
  #deal with non-ducks
```

- but can duck typing be used with compile-time checks instead of runtime exceptions?

Interfaces

- like classes, **interfaces** specify an API, without specifying implementation
  - allows for a hierarchy where classes can implement *multiple* interfaces
    - whereas classes only inherit from a single parent
  - ie. inherit *wealth* or code from a parent, but inherit *obligation* or an API from implemented interfaces
  - Java interfaces is a form of duck typing, with static checking

Example using interfaces:

```
interface Lengthable {
  int length();
}

interface Geometry2D extends Lengthable {
  int width();
}

class Canvas implements Geometry2D {
  int length() { ... }
  int width() { ... }
}
```

- an **abstract class** is a combination of a normal class and an interface, hybrid
  - some implemented methods, some methods left as API
  - abstract classes *cannot* be instantiated
    - *forced* to subclass and define methods
  - as oppose to **concrete classes** are entirely implemented

Example using abstract classes:

```
abstract class C { // new C() is not allowed
  int length() { ... }
  abstract int width();
}

abstract class D extends C { // new D() is not allowed
  int height() { ... }
}

class E extends D { // concrete class, can be instantiated
  int width() { ... }
}

C x = new E(); // OK
```

- opposite of abstract classes are **final classes** that *prohibit* subclassing or defining submethods
  - ie. leaves in the object hierarchy
  - similarly, a **final method** cannot be overriden in a subclass
  - *advantages*:
    - speed
      - allows Java compiler to inline the bodies of final methods
    - barrier preventing subclasses from misbehaving
    - lower-level projects, native methods

Exceptions

- **exceptions** are error conditions that stop ordinary program flow
  - Java allows exceptions to be handled in different ways, so that programs can recover
- different execption types:

  - `NullPointerException` :
    - eg. `String s = null; s.length();`
  - `ArithmeticException` :
    - eg. `int a = 3 / 0;`
  - `ArrayIndexOutOfBoundsException` :
    - eg. `int [] a = new int[2]; a[2] = 1;`
  - `ClassCastException` :
    - eg. `Object x = new Integer(1); String s = (String) x;`
  - `StringIndexOutOfBoundsException` :
    - eg. `String s = "hello"; s.charAt(5);`
- exceptions in Java are objects
  - when the exception is triggered, Java creates an object of the appropriate class and *throws* the object
  - exceptions are derived from the predefined base class `Throwable`
    - the `Error` class also derives from `Throwable` , and indicates serious unrecoverable errors such as `StackOverflowError`
    - throwable objects have the `printStackTrace` and `getMessage` methods
  - most exceptions derive from `RuntimeException` class which derives from the `Exception` class
  - exceptions can be **caught**, as well as **thrown**
  - some exceptions are **checked exceptions** that cannot be ignored, and must be caught by a `catch` part eventually
    - only exceptions derived from `Error` and `RuntimeException` are unchecked
  - Java uses static checking to make sure that a caller is prepared for exceptions
    - each method *declares* the exceptions it might throw, as part of the *signature*

Catching exceptions:

```java
public class Test {
  public static void main(String[] args) {
    try {
      int i = Integer.parseInt(args[0]);
      int j = Integer.parseInt(args[1]);
      System.out.println(i/j);
    }
    catch (ArithmeticException a) {
      // catches all ArithmeticExceptions, even ones in other called methods
      // ie. a long-distance throw
      System.out.println("dividing by zero");
    }
    catch (ArrayIndexOutOfBoundsException a) {
      System.out.println("requires two params");
    }
    catch (RuntimeException a) {
      // will not reach here, tries catching in order
      System.out.println("runtime");
    }
  }
}
```

Throwing custom exceptions:

```
public class OutOfGas extends Exception {
  private int miles;
  public OutOfGas(String details, int m) {
    super(details);
    miles = m;
  }
  public int getMiles() { return miles; }
}

try {
  throw new OutOfGas("run out of gas", 10);
}
catch (OutOfGas e) {
  System.out.println(e.getMessage());
  System.out.println("Odometer: ", e.getMiles());
}
```

`throws` clause:

```
// if called, eventually some function must catch thrown error
// method is permitted to throw the checked exceptions in its throws clause
void z() throws OutOfGas {
  throw new OutOfGas("run out of gas", 10);
}
```

`finally` block:

```
file.open();
try {
  workWith(file);
}
finally {
  // finally block always executes when the try-catch is finished
  file.close();
}
```

Concurrency

- motivation involving ultra powerful computers:
  - top500 lists the 500 fastest computers known in the world
    - indexed by linpack, a demanding linear programming benchmark
  - programmers on such machines write code almost entirely in C or assembly to squeeze out every last bit of performance
  - for future supercomputers, ordinary programmers should not have to write in low level for optimal performance
    - need an alternative option that also scales well enough to perform on supercomputers with many processors and cores
- Java aimed at the early supercomputers of the 1990s:
  - approaches for lashing many processors and cores together:
    1. **single instruction multiple data (SIMD)**
       - eg. an ADD instruction causes many additions to multiple data
    2. **multiple instruction multiple data (MIMD)**
       - multiple **instruction pointers (IP)**
- Java provided the `Thread` as an abstraction for MIMD:
  - each `Thread` object corresponds to one computation with its own IP
  - two ways to create threads:
    1. subclass `Thread` and override its `run` method
    2. define a class that implements the `Runnable` interface

Creating Java threads:

```
class Foo extends Thread { // classical inheritance
  void run() {...}
  ...
}

interface Runnable { void run(); }

class Bar implements Runnable { // duck typing with interfaces
  void run() {...}
  ...
}
```

- thread life cycle:
  - controlled by parent thread:
    - NEW state: initially, created via `new`
      - in this stage, it is not running, so its priorities and options can be modified
    - RUNNABLE state: triggered by `t.start()`
      - allocates OS resources to actually run the thread
      - calls `t.run()` in the new thread
      - note that being in the runnable state does not mean the thread is running, up to the OS
  - controlled by child thread:
    - RUNNABLE state: currently computing in child thread
    - RUNNABLE state: triggered by `yield()`
      - politely reschedule themselves, for performance
    - BLOCKED state: eg. blocked by I/O
    - WAITING state: triggered by `wait()`
    - TIMEDWAITING state: triggered by `sleep()`
    - TERMINATED state: return from `run()` method
- MIMD has a central problem of allowing multiple threads communicate with each other
  - Java uses a shared memory approach
    - a thread T can commmunicate to another thread U by modiying a shared object O
- Java memory model:
  - every thread has its own stack
    - variables are stored on stack
  - a *shared* heap is used to store *all* objects
    - objects can be referred to by thread local variables and other objects
    - if one object has multiple references, this can lead to **race conditions**
- techniques for dealing with common multithreading issues:
  - **cache invalidation**:
    - in modern processors, there are caches for each thread, but they may hold different values
    - a `volatile` instance variable tells the compiler that the variable is intended for communication
      - ie. assume that the variable will spontaneously change in value due to another thread
      - `volatile` is a constraint on the compiler, cannot optimize away access to the variable

- ensures that every time the variable is accessed, it is read from main memory
- however, does not solve all problems, since objects can still contain temporarily *inconsistent* state
- reliability over performance

- **race conditions**:
  1. use the `join` method to wait for threads to finish
     - simplest and easiest
     - works for "embarassingly parallel" problems
  2. use monitors and the `synchronized` keyword
     - when a thread executes any synchronized method, no other thread can execute any sychronized method on the same object
     - in shared objects, use `sychronized(this)` in a method to obtain a lock that is released at the closing brace
     - synchronized methods should be really fast, since all other synchronized methods must wait
     - reliability over performance
  3. waiting for events
     - sometimes, the computations in a synchronized method cannot be completed fast enough
     - `o.wait()` removes all locks (implicit from synchronized methods) held by this thread
       - this thread waits until `o` becomes unavailable
     - `o.notify()` wakes up one of the threads waiting on `o`
     - `o.notifyAll()` wakes up *all* threads waiting on `o`
       - more flexibility, allows threads to collaborate to decide who should run next
  4. use atomic operations instead
     - eg. `AtomicIntegerArray` has atomic operations like `getAndAdd`

- higher level techniques provided by Java:
  1. `Exchanger` :
     - ie. drop box objects
     - `u = x.exchange(v)` communicates `v` to partner thread and gets `u` from partner
  2. `CyclicBarrier` :
     - most computation is embarassingly parallel
     - resynchronize occasionally

Using `volatile` :

```java
volatile long x;
...
// require two loads from memory for x,
// instead of optimizing into one access if x were not volatile
y = x - x;
y = f(x) + g(x);

// if x were not volatile, could be optimized into while (true)
while (x == x)
  ...
```

When `volatile` is not enough:

```
class C {
  volatile int x, y; // condition: x < y always
  void set(int newx) {
    if (newx < newx+1) { // check no overflow
      x = newx; y = newx+1;
    }
  }
  int getx() { return x; }
  int gety() { return y; }
}

// volatile does not protect from the race condition
int a = getx();
int b = gety();
if (a >= b)
  meltdown()
```

Solved using `synchronized` :

```
class C {
  volatile int x, y; // condition: x < y always
  synchronized void set(int newx) {
    if (newx < newx+1) {
      x = newx; y = newx+1;
    }
  }

  // synchronized getx and gety is not enough since
  // set can occur between those calls
  int getx() { return x; }
  int gety() { return y; }

  // have to get both
  synchronized long getboth() { return ((long) x << 32) + y; }
}

long ab = getboth();
long a = ab >> 32;
long b = ab & 0xFFFFFFFF;
if (a >= b)
  meltdown()
```

Python

Generators

- a **generator** in Python uses the `yield` keyword
  - allows for *lazy* evaluation
  - similar to a lazily calculated list

Generator examples:

```
def myrange(l, r): # range in Python is a builtin generator
    print(f"i = {l}")
    i = l
    while i < r:
        yield i
        print("i += 1")
        i += 1

mygen = myrange(1, 5) # creates a generator
next(mygen) # prints "i = 1", gives 1
next(mygen) # prints "i += 1", gives 2
...
next(mygen) # throws StopIteration exception after 5
```

Multithreading

- Python uses garbage collection to clean up objects
  - maintains a *reference count* to every object
  - frees objects when their reference count drops to zero

Using `sys.getrefcount` :

```
import sys
a = []
b = [a, a, a]      # pointers, not copies
sys.getrefcount(a) # gives 5, counting function call
```

- similarly to Java, locks are needed to avoid conflicts
  - but the reference count will change *frequently*, especially between multiple threads

  - Python has a **global interpreter lock (GIL)** for updating the reference counts for shared object across threads
    - unlike Java, which *does not* use reference counts for garbage collection
      - only when provided memory becomes full, does Java copy over referenced objects into a new memory space and free all previous memory
    - but this *greatly* slows down the problem:
      - even with multiple threads, only one of them is running most of the time
      - threads are managed by OS with a high overhead, with call stacks, thread local memory, etc.
  - need alternatives for multithreading in Python
    - eg. in client-server model, need a non-blocking way to handle many requests at a time
- in a *multiprocessing* program:
  - independent processes
  - processes do not share a heap, *unlike* threads
    - no need for global reference locks
  - useful for multicore CPUs
  - *more* overhead to spawn processes than threads
  - *harder* to share objects between processes
- `select` is another method for handling multiple requests *without* having multiple threads
  - harder to read and maintain because the processing procedure is split into *slices*
    - must manually handle temporary variables
  - is it possible to:
    - keep the original structure of the program
    - return immediately when an I/O operation is executed *without* blocking
    - *re-enter* the program when the I/O operation is finished
      - *not* possible in C/C++
      - but this is *exactly* like using continuations in Scheme with `call/cc`

Continuations in Python using generators:

```
def coroutine_main(ch): # generator that prints ch
    while True:
        print(ch, end='')
        yield

coroutine_list = [ # list of generators
```

```
    coroutine_main('h'),
    coroutine_main('i'),
    coroutine_main('\n')
]

while coroutine_list:
    next_coroutine = coroutine_list.pop(0)
    next(next_coroutine)
    coroutine_list.append(next_coroutine)
    # prints hi\n infinitely
```

- note that the `coroutine_main` runs *concurrently* (in the same thread) but not in *parallel*
  - programmer can control when to switch using `yield`
  - compared to threads, there are less locks and less resources

Builtin coroutine example using `asyncio` :

```
import asyncio

async def coroutine_main(ch):
    while True:
        print(ch, end='')
        await asyncio.sleep(0.1) # await instead of yield

async def execute():
    coroutine_list = asyncio.gather(
        coroutine_main('h'),
        coroutine_main('i'),
        coroutine_main('\n')
    )
    try:
        await asyncio.wait_for(coroutine_list, timeout=0.5)
    except asyncio.TimeoutError:
        print('Done')
        coroutine_list.exception() # throw away exception

asyncio.run(execute())
```

Object-Oriented Programming

- although Java is an object-oriented language, there are other important aspects of object-oriented (OO) *programming* as a paradigm:
  - object-oriented programming is not the same as programming in an object-oriented language such as Java
    - eg. OO programs can be written in C, using `struct` for classes, and creating table of pointers to functions for virtual methods
  - object-oriented languages are not all like Java:
    - ie. OO is not a *monolith*, there are many variations
    - eg. Python uses dynamic checking, while Java uses static checking
      - static type checking complicates the language with interfaces, abstract classes, generics, etc.
  - nuances of OO:
    - some argue Java is *not* fully object-oriented, due to primitive types like `int`
    - some argue Scheme is object-oriented in some sense, because every Scheme value is an object
- *class-based* object-oriented languages eg. Python, Java, C++ share the following properties:
  - class *bundle* together fields ie. instance variables with methods
  - instantiation as the way to create objects
    - note that you can use `new` to create a new object, or `clone` to allocate a new object and copy an existing object's value into it
  - inheritance is class-based
  - classes are types
  - classes control namespaces
- alternatively, *prototype-based* OO languages eg. JavaScript share the following properties:
  - no `new` , just `clone`
    - gives more control and flexibility to the programmer, can easily modify and change prototypes
    - simpler model at the low level
  - typically works better with dynamic rather than static checking

Different approaches to OO instantiation:

```
// class-based instantiation
T x = new T();

T 0 = ...;
T x = o.clone();
```

```
// prototype-based instantiation
greenthread = t.clone();
greenthread.color = "green"; // greenthread is now a *prototype* for green threads
gt = greenthread.clone();
```

- single vs. multiple inheritance as another OO variant:
  - Java uses single inheritance, classes form a tree rooted at `Object`
    - for `o.m()` , look in `o` 's class, then its parent, then its parent's parent, etc.
  - Python allows for multiple inheritance, classes form a DAG
    - rules for inheritance are more complex
    - for `o.m()` , look in `o` 's class, then look at its first parent, then first parent's ancestors, then second parent, etc.
- consider OO programming in a functional language like ML:
  - objects will be implemented as *functions*
  - objects will take a message and return a response

Stack implementation using object-oriented programming in ML:

```
datatype message = IsNull | Add of string | HasMore
  | Remove | GetData | Get Link;
datetype response = Pred of bool | Data of string
  | Removed of (message -> response) * string
  | Object of message -> response;

fun root _ = Pred false;
fun null IsNull = Pred true
  | null message = root message;

fun node data link GetData = Data data
  | node data link GetLink = Object link
  | node _ _ message = root message;

fun stack top HasMore =
    let val Pred(p) = top IsNull in Pred(not p) end
  | stack top (Add data) = Object(stack (node data top))
  | stack top Remove = let
      val Object(next) = top GetLink
      val Data(data) = top GetData in
        Removed(stack next, data)
      end
  | stack _ message = root message;
```

```
(* a kind of inheritance *)
fun peekableStack top Peek = top GetData
  | peekableStack top message = stack top message;

val a = stack null;
val Object(b) = a (Add "the plow.");
val Object(c) = b (Add "forgives ");
val Object(d) = c (Add "The cut worm ");
val Removed(e, s1) = d Remove; (* s1 = "The cut worm " *)
val Removed(f, s2) = e Remove; (* s1 = "forgives " *)
val Removed(_, s3) = f Remove; (* s1 = "the plow." *)
```

Logic Programming

- the **logic programming** paradigm gives up on functions as well as side effects, and focuses on **predicates** and *gluing* them together using logical connectives such as `& | →`
  - a statement in logic, rather than involving any function calls
  - higher-level of thinking, with *declarative* rather than *procedural* or *functional* programs
    - don't focus on *how* to implement goals, focus on *specifying* the goals of the program ie. what you want
    - ie. declare the constraints on the solutions, and the system finds them
  - *mantra* of logic programming:
    - any running *algorithm* is made of logic and control
      - divide and conquer, separation of concerns
      - in other languages, impossible to separate these two aspects
    - **logic** is the specification of the problem
      - doesn't worry as much about efficiency, all about correctness
    - **control** is efficiency advice to the interpreter
      - entirely efficiency based, cannot affect the correctness of the program
  - logic programming was championed by Prolog
- for example, to sort a list with a logic program:
  - may consider different algorithms such as quicksort, merge sort, heap sort, etc.
    - but these are all different *efficiency* solutions to the same underlying problem
  - instead, in a logic program, we would write a *spec* for what it means to sort a list in terms of logical clauses
    - a **clause** is a logical statement that is the unit of a logic program, ordinarily assumed to be true

Prolog

Syntax

- Prolog programs are made up of a series of **clauses** or logical predicates that are made up of **terms**
  - declarative, each piece of the program corresponds to a simple mathematical abstraction
- a Prolog **term** is one of:
  - **number** eg. `12 1e-6 12.3`
  - **atom** eg. `abc a912 'Xyz' 's s/!' 'o''clock'`
    - `[a-z][A-Za-z_0-9]*` , or quoted atoms that can contain any characters
    - each atom is *unique* and equals only itself
  - **variable** eg. `X Y PL1`
    - `[A-Z_][A-Za-z_0-9]*`
      - `_` itself is special, nameless variable
      - better to avoid variables names starting with `_`
    - can become *instantiated* or *bound* to values as the program succeeds
      - a bound variable cannot become bound to a different value, unless the code fails (making it unbound), and then succeeds in a different way (making it bound again)
      - similar to iterative and functional variables in some ways
    - can become *uninstantiated* or *unbound* on failure
  - **structure** or compound term eg. `sort(X, Y) functor(arg1, ..., argN)`
    - **functor** or function symbol is an atom that *tags* the structure followed by `/` and number of arguments
    - arguments are terms:
      - N is the **arity** or number of terms of the structure, and also tags the structure
      - `append/3` is functor indicating the structure with function symbol `append` and arity 3
    - root of a tree / data structure representing function calls, but there is *no actual function*
- syntactic sugar, ie. extra shorthand for previously seen forms:
  - `[] = '[]'` , empty list is an atom
  - `[a,b,c] = '.'(a, '.'(b, '.'(c, '[]')))` , lists are just linked lists
    - `./2` is the fundamental functional symbol used to write lists in

Prolog

- - - - similar to construct operator
    - eg. `[X|Y] = '.'(X, Y)`
  - `3+4*5 = '+'(3, '*'(4, 5))`
- more syntactic sugar for building clauses:
  - `:-` is a **turnstile**, `A :- B` means “A is true if B is true”
    - *not* “if and only if”
    - eg. `a :- b, c, d.` is the same as using `:-/2` in `':-'(a, ','(b, ','(c, d)))`
  - `,` indicates “and” at the top level, `A, B` means “A and B”
    - order doesn’t matter for correctness, but it does matter for control
  - `.` indicates the end of a clause at the top level
- pattern matching using Prolog terms is called **unification**:
  - two terms are said to *unify* if there is some way of binding their variables that makes them identical
  - eg. `parent(adam, Child)` and `parent(adam, seth)` unify by binding the variable `Child` to the atom `seth`
    - ie. applying a *substitution* of variables to create a unifying instance of a term
  - eg. `f(g(Y), h(Z))` and `f(A, h(i(B)))` bind with the unification `A = g(y)` and `Z = i(B)`
  - Prolog wants to bind variables only when necessary to prove a query, so it chooses **most general unifiers (MGUs)**
    - eg. `parent(X, Y)` and `parent(fred, Y)` can be unified by binding `X` to `fred`
    - or by binding `X` to `fred` and `Y` to `mary`
    - Prolog chooses the MGU, the first simpler binding
  - variables in goals can be unified, as well as variables in clause heads
    - more powerful that matching in other languages, ie. Prolog can *build* values via unification
    - supply info from caller to callee, as well as callee to caller
  - unification in Prolog replaces parameter-passing, assignment, and data constructors from other languages
  - for a variable `X` and term `t`, an MGU is binding `X` to `t` *except* for a special case:
    - when `t` is a compound term in which the same variable appears
    - must perform a check known as the **occurs check** to ensure this is not the case, usually not done automatically
      - Prolog will misbehave on unifications that fail the occurs check
    - eg. binding `e(X, X)` and `e(Y, f(Y))` gives an infinite term `Y = f(f(f(...)))`
    - eg. `append([], X, [a|X])` *succeeds* with `X = [a|**]`, which has a cyclic term ie. circular linked list of `a`

- types of Prolog clauses:
  - a **fact**, ie. no conditions, has a term followed by a period
  - in a **rule**, the terms are conditional, ie. includes turnstiles
  - a **goal** is supplied to the Prolog interpreter
    - attempts to find a match of the goal in the heads of all the facts and rules
    - if it finds no match, it *fails*
    - often failure means false, but not always
    - ie. Prolog either has *success* which means true and failure which means it couldn't be proved either way
  - note that a clause can refer to a predicate that *hasn't* been defined yet, as long as the predicate is defined before the clause is used

Illustrating distinction of failure vs. false:

```prolog
prereq(cs31, cs32).
prereq(cs31, cs111).
prereq(cs31, cs131).
prereq(cs31, cs132).

% Can we conclude that Dance 101 is *not* a prereq for CS 131?
% Logically, no.

% But from the following additional crucial, but often unstated, assumption:
% Closed World Assumption (CWA): the provided list is the complete list
% Then, yes.
```

- the builtin predicate `is/2` is used to evaluate in Prolog:
  - `N is 2+2.` or `is(N, 2+2)` , gives `N = 4.`
  - but usually not used by Prolog programs, which use purely symbolic forms
- Prolog numeric comparisons:
  - `< > =< ≥ =:= =\=`
    - note that `≤` is not used here to allow it to be used as an arrow notation
  - `X is Y` evaluates `Y` and unifies the result with `X`
  - `X=Y` unifies `X` and `Y` only with respect to term structure
  - `X=:=Y` evaluates both `X` and `Y` and succeeds if and only if they are numerically equal
    - all variables need to be instantiated in this predicate

Using numeric comparisons:

```prolog
length([], 0).
length([_|Tail, Len]) :- length(Tail, TailLen), Len is TailLen + 1.
```

```
sum([], 0).
sum([Head|Tail, X]) :- sum(Tail, TailSum), X is TailSum + Head.

fact(X, 1) :- X =:= 1, !.
fact(X, Fact) :- X > 1, NewX is X - 1, fact(NewX, NF), Fact is X * NF.

num(0).
% num(X) :- num(X-1). % fails, X-1 is *not* evaluated
% num(X) :- X1 is X + 1, num(X1). % instantiation error, X1 doesn't have value
% num(X) :- X1 is X - 1, num(X1). % works only for number testing, not enumeration
% num(X) :- num(X1), X is X1 + 1. % works for testing and enumeration
                                  % but this leads to infinite loop for num(1) etc.
                                  % if first solution is rejected
num(X) :- num(X1), X is X1 + 1, !. % cut fixes infinite loop
```

- Prolog list predicates:
  - `append(X, Y, Z)` is provable if `Z` is the result of appending `Y` onto `X`
  - `member(X, Y)` is provable if `Y` contains `X`
  - `select(X, Y, Z)` is provable if `Y` contains `X` and `Z` is the same as `Y` but with one instance of `X` removed
  - `nth0(X, Y, Z)` is provable if `Z` is the `X` th element of `Y`
  - `length(X, Y)` is provable if `X` is a list of length `Y`
    - useful for *fixing* the length or shape of a list
  - some predicates are *inflexible*:
    - `reverse(X, Y)` can lead to infinite loops because of its implementation and Prolog's proof technique
    - `sort(X, Y)` is *not* permitted by Prolog to be a variable
    - these issues stem from using Prolog purely declaratively and ignoring its procedural proof aspect

Prolog list predicate implementations:

```
append([], B, B).
append([Hd|TlA], B, [Hd|TlC]) :- append(TlA, B, TlC)

% using select to remove an element
select(2, [1,2,3], Z). % gives Z = [1,3] ; false.
% using select to insert an element
select(2, Y, [1,3]).   % gives Z = [2,1,3] ; Z = [1,2,3] ; Z = [1,3,2] ; false.

reverse([], []).
reverse(Hd|Tl, X) :- reverse(Tl, Y), append(Y, [head], X).
```

- to run code in Prolog, an interpreter is used:
  - prompts for **queries** from the user
  - the interpreter maintains a database of facts and rules, and attempts to prove the query
    - queries can contain variable terms, and logical conjunction of terms
    - will try and unify the query term variables with other terms by procedurally trying bindings
  - thus queries are more *flexible* than function calls in other languages

Example Prolog queries:

```prolog
parent(kim, holly).
parent(margaret, kim).
parent(esther, margaret).
parent(herbert, jean).
greatgrandparent(GGP, GGC) :- parent(GGP, GP), parent(GP, P), parent(P, GGC).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(Z, Y), ancestor(X, Z).

% queries:
parent(kim, holly).    % gives true.
parent(fred, pebbles). % gives false.
parent(P, jean).       % gives P = herbert.
parent(P, esther).     % gives false.
parent(esther, Child). % gives Child = margaret.
parent(P, C). % gives P = kim, C = holly...
parent(P, P). % gives false
parent(margaret, X), parent(X, holly). % gives X = kim.
```

Examples

List operations:

```prolog
member(X, [X|_]).
member(X, [_|L]) :- member(X,L).
% member(a, [a,b,c,X,Y,d,e,Z]). gives four true unifications

% O(N)
append([], M, M).
append([X|L], M, [X|LM]) :- append(L, M, LM).
% append(X, Y, [a,b,c]). gives four true unifications for X and Y
% O(N^2)
reverse([], []).
reverse([X|L], RX) :- reverse(L, R), append(R, [X], RX).

% O(N) - using accumulation for more efficiency
revapp([], A, A).
revapp([X|L], A, RXA) :- revapp(L, [X|A], RXA).
reverse(L, R) :- revapp(L, [], R).

% reverse(L, [a,b,c,d]). gives [d,c,b,a], but then stack overflow if rejected
% Prolog continues matching L with longer and longer lengths (in both versions)
```

Sorting a list:

```
% L - unsorted list
% P - sorted list that is otherwise equivalent to L,
%     ie. P is a permutation of L and P is sorted.
sort(L, P) :- permute(L, P), sorted(P).

% use multiple clauses here, Prolog interpreter can use *any* of them
sorted([]).  % this is a fact for empty lists, alternatively, sorted([]) :- true
sorted([_]). % another fact for singleton lists
sorted([X, Y|L]) :- X =< Y, sorted([Y:L]). % rule for remaining lists

% sorted([X,Y|L]) :- X =< Y, sorted(L). % improper logic, need to relate L to Y!
  % matches any list with first two members X, Y and reamining list L
  % like OCaml's X::Y::L

% permute(L, P) :- L and P are permutations of each other
permute([], []).
permute([X|L], R) :-       % a permutation of [X|L] is defined by:
  permute(L, PL),          % taking some permutation of L,
  append(PL1, PL2, PL),    % splitting it into segments,
  append(PL1, [X|PL2], R). % appending X between the two segments

% a bunch of logic statements... but it will run!
sort([1,9,-3], R).              % gives R = [-3,1,9].
append([a,b],[c,d,e], R).       % gives R = [a,b,c,d,e].
append(X, [c|Y], [a,b,c,d,e]). % gives X = [a,b], Y = [d,e].
append(X, Y, [a,b,c,d,e]).      % gives multiple answers starting with
  % X = [], Y = [a,b,c,d,e] ...
  % means this call can succeed *multiple* times as it tries all iterations
% append is O(N) → permute is O(N!) → sort is O(N!), not optimal
% need to work on control
% note that sort (O(nlogn)) and append are builtin
```

- achilles heel:
  - Prolog is checking *all* logical iterations of the inputs / variables to see if they satisfy the clauses
    - the *logic* of the program may be right, but the *control* may not be good enough

Knapsack problem:

```
% weight(L, N) takes list L of food terms (Name,Weight,Calories)
weight([], 0).
weight([food(_,W,_)|Rest], X) :- weight(Rest, RestW), X is W + RestW.

% calories(L, N) takes list L of food terms (Name,Weight,Calories)
calories([], 0)
weight([food(_,_,C)|Rest], X) :- calories(Rest, RestC), X is C + RestC.

% subseq(X, Y) succeeds when list X is same Y but with zero+ elements omitted
% can be used to *generate* subsequences
subseq([], []).
subseq([Item|RestX], [Item|RestY]) :- subseq(RestX, RestY).
subseq(X, [_|RestY]) :- subseq(X, RestY).

% find some solution, not necessarily optimal
knapsackDecision(Pantry, Capacity, Goal, Knapsack) :-
  subseq(Knapsack, Pantry),
  weight(Knapsack, Weight),
  Weight =< Capacity,
  calories(Knapsack, Calories),
  Calories ≥ Goal.

% findall takes a variable, a goal containing the variable, and
% collects a list of all the instantiations of that variable in a list
findall(X, Goal, L).

legalKnapsack(Pantry, Capacity, Knapsack) :-
  subseq(Knapsack, Pantry),
  weight(Knapsack, W),
  W =< Capacity.
```

```prolog
% maxC helper takes remaining list of lists of food terms, best list so far,
% its total calories, and the final result
maxC([], Sofar, _, Sofar).
maxC([First|Rest], _, MC, Result) :-
  calories(First, FirstC),
  MC =< FirstC,
  maxC(Rest, First, FirstC, Result).
maxC([First|Rest], Sofar, MC, Result) :-
  calories(First, FirstC),
  MC > FirstC,
  maxC(Rest, Sofar, MC, Result).
maxCalories([First|Rest], Result) :-
  calories(First, FirstC),
  maxC(Rest, First, FirstC, Result).

knapsackOptimal(Pantry, Capacity, Knapsack) :-
  findall(K, legalKnapsack(Pantry, Capacity, K), L),
  maxCalories(L, Knapsack).
```

Eight-Queens problem:

```prolog
% nocheck(X/Y, L) takes queen X/Y and list of queens,
% succeeds only if the X/Y queen holds none of the others in check.
% using the division symbol just as a constructor
nocheck(_, []).
nocheck(X/Y, [X1/Y1 | Rest]) :-
  X =\= X1, % not same row
  Y =\= Y1, % not same col
  abs(Y1-Y) =\= abs(X1-X), % not same diagonal, horiz dist = vert dist
  nocheck(X/Y, Rest).

% legal if all coords in range and no queen in check.
% eg. X = [_,_,_], legal(X). finds legal placements for three queens
legal([]).
legal([X/Y | Rest]) :-
  % legal(Rest) *must* be called first in order to *instantiate* Rest
  % as a legal arrangement with one less queen.
  legal(Rest),
  member(X,[1,2,3,4,5,6,7,8]),
  member(Y,[1,2,3,4,5,6,7,8]),
  nocheck(X/Y, Rest).

eightqueens(X) :-
```

```
  % don't want permutations
  X = [1/_,2/_,3/_,4/_,5/_,6/_,7/_,8/_],
  legal(X).
```

Procedural View

- each rule in Prolog should give a **procedure** for proving a goal
  - eg. to prove `p :- q, r.` , unify with `p` , prove `q` , and then prove `r`
  - eg. to prove `s.` , just unify with `s`
- Prolog will use **backtracking** to explore all possible targets in the order given
  - until it finds as many successes as required or all possibilities are exhuasted
  - ie. creating a proof that is a tree ie. **proof tree**:
    - the root is the goal
    - subtrees represent subproofs or subgoals, worked on from left-to-right, depth-first
    - each branch corresponds to succesful rules
    - leaves are facts
    - failures are erased from the tree, to save time
    - at any point in time, Prolog has a partially-built proof tree
  - ie. goal oriented, using *backward chaining*
  - coupled with **substitutions** or matching that bind variables to terms
  - order is thus *very important* in terms of controlling how Prolog runs

Illustrating backtracking:

```
p :- q, r. % 1
q :- s.    % 2
q.         % 3
r.         % 4
s :- 0=1.  % 5

% tries 1, 2, 5 which fails
% backtracks to 3, which succeeds
```

- but there is another tree, representing the possible solutions Prolog is looking for ie. **search tree**:
  - leaves are answers (answers at top-level of interpreter)
  - internal nodes are **choice points**:
    - when a choice point is reached, an alternative is chosen via depth-first, left-to-right approach
    - this is where *control* comes into play, since Prolog may not always

        make the correct decision
    * eg. avoid large, uneccesary subtrees in the search tree
  - a mechanism to accomplish this is using the **cut** operator `!`
    * allows for *pruning* away alternatives ie. part of the search tree, that are known by the programmer to not work
    * always succeeds once, but if backtracked into, the *calling* predicate immediately fails

Considering search trees:

```
true.

% repeat is the while true of Prolog, can infinitely backtrack and reprove
% right-recursive search tree
repeat.
repeat :- repeat.

repeat, write('x'), fail. % prints infinite 'x'

% left-recursive search tree
loop :- loop.
loop.

loop. % attempting to prove loop will cause an infinite loop
```

Using the cut:

```
complicated(X, Y, Z) :- generate(X, Y, Z), test(X, Y, Z).
% eg. sorted :- permute, ordered.

% Suppose test/3 is slow, and we want to skip it.
% Suppose test(X,Y,Z) will eventually fail unless X is a member of Y.

% retune the predicate
complicated(X, Y, Z) :- generate(X, Y, Z), member(X, Y), test(X, Y, Z).

% Still have efficiency problem!
% Suppose X=a, Y=[a,b,r,a,c,a,d,a,b,r,a].
% Then member(X,Y) succeeds, and we'll call test/3.
% But if test/3 fails, member(X,Y) succeeds 5 times total.
% So test/3 is called 5 times!

% ! removes the ability to backtrack into later clauses
memberchk(X, [X|_]) :- !.
memberchk(X, [_|L]) :- memberchk(X, L).

complicated(X, Y, Z) :- generate(X, Y, Z), memberchk(X, Y), test(X, Y, Z).

% Suppose X=a, Y=[a,b,r,a,c,a,d,a,b,r,a].
% Then member(X,Y) succeeds, and we'll call test/3.
% But if test/3 fails, it'll backtrack into the cut.
% So test/3 is called just once!
```

- cuts are nicknamed the *goto* of Prolog, since there are some problems with them
- distinguishing truth vs. provability:
  - if P is provable but not true, the logic is *inconsistent*
  - if P is true but not provable, the logic is *incomplete*
  - logisticians have the goal of having truth equal probability:
    - eg. goal met for Euclidean geometry
    - eg. goal is impossible for integer arithmetic, due to Gödel's theorems

Problems with cuts:

```
% succeed if P fails, and fail if P succeeds
% looks like negation, aliased as 'not' in many Prolog systems
\+(P) :- P, !, fail.
\+(_).

% but it is *not* negation, \+ is not provable
X=1, \+(X=1), write(ok). % P && Q writes ok
\+(X=2), X=1, write(ok). % Q && P fails!

\+ prereq(dance101, cs131).
% succeeds, and this is OK if the CWA is valid

\+ (X=1).
% fails because CWA in Prolog is dicey if variables are involved
% argument to \+ should be a *ground term* without logical variables
```

Appendix

- `write/1` is a predefined predicate with a side effect
  - true for input, but prints out parameter when called

    - `nl/0` is a predicate that prints a new line

- `read/1` prompts for input, and reads in a term, unifying it with the parameter `X`

- `trace/0` allows for debugging
    - places *spies* at the four *ports* around each goal
    - ie. the **four-port debugging model**:
        * entrance ports: call goal, or backtrack (or redo) from another goal
        * exit ports: fail from goal, or succeed from goal

- `fail/0` always fails

- `assert` and `retract` are used to add and remove terms as facts in the current database
    - create self-modifying behavior

- the cut prdicate `!` takes no parameters and eliminates backtracking
    - used to improve efficiency, or change the found solutions
    - acts as a checkpoint that tells Prolog not to backgrack before the checkpoint

- `not` is unprovability (different from negation)
    - implemented as `not(Goal) :- call(Goal), !, fail.` followed by `not(Goal).`

Finite Domain Operations

- Prolog has a finite domain solver that allows for efficiently using **finite domain** operations ie. set operations
    - FD operators are set operators, so they do not require instantiation
    - `fd_labeling` draws numbers from the finite domain that satisfy all requirements
        * should be the last line

Generating permutations using finite domain operations:

```
sum_fd([], 0).
sum_fd([H|T], Sum) :- sum_fd(T, SumT), H + SumT #= Sum.

% get permutation of 1..5 that sums to 15
permute_fd(Lst) :-
  length(Lst, 5),
  fd_domain(Lst, 1, 5), % numbers in Lst from 1 to 5
```

```
fd_all_different(Lst),
sum_fd(Lst, 15),
fd_labeling(Lst).
```

Fixing Common Errors

- `Y = X-1` only does unification, not evaluation:
  - `rec(X,Y) :- rec(X-1,Y).` does not work
  - `rec(X,Y) :- X1 is X-1, rec(X1,Y).` works
- handling instantiation errors:
  - for `X is Y+1` , `Y` is known and `X` is assigned to `Y` + 1
  - for `X =:= Y+1` both variables need to be known
  - for `X #= Y+1` no prerequisites
- sometimes changing the order will work:
  - eg. draw out an element first
  - `zero_or_five(X) :- Y is X-1, member(Y, [-1,4]).` does not work
  - `zero_or_five(X) :- member(Y, [-1,4]), X is Y+1.` works
- order matters, and the base cases should come first
  - so that Prolog tries predicates in order, and backtracks correctly
- drawing a lot of variables before they are actually used leads to inefficiency:
  - `member(X,[1,2]), member(Y,[1,2]), member(Z,[1,2]), S is X+Y, S is Y+Z, S<3.` is slow
  - `member(X,[1,2]), member(Y,[1,2]), S is X+Y, S<3, member(Z,[1,2]), S is Y+Z.` is faster
- consider using cuts in order to accelerate code
  - but be wary of side effects

Appendix

Sample Midterm

1. a. Write an OCaml function `merge_sorted` that merges two sorted lists given a comparison function.

`merge_sorted` implementation:

```
(* non-tail recursive *)
let rec merge_sorted lt l1 l2 =
  match l1 with
    | [] -> l2
    | hd::tl -> match l2 with
        | [] -> l1
        | hd2::tl2 -> if lt hd hd2 then hd::(merge_sorted lt tl l2)
            else hd2::(merge_sorted lt l1 tl2)

(* tail recursive *)
let merge_sorted lt l1 l2 =
  let helper l1 l2 hds =
    match l1 with
      (* note rev_append is tail recursive, while append and @ are not *)
      | [] -> List.rev_append hds l2
      | hd::tl -> match l2 with
          | [] -> List.rev_append hds l1
          | hd2::tl2 -> if lt hd hd2 then helper lt tl l2 (hd::hds)
              else helper lt l1 tl2 (hd2::hds)
  in helper l1 l2 []

(* quick rev_append tail recursive implementation *)
let revapp (h::t) l2 = revapp t (h::l2)
```

1. b. What is the type of `merge_sorted` ?
   - `('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list`

   c. What does `merge_sorted (fun a b -> List.length a < List.length b)` yield, and what is its type?
   - A function that sorts list of lists by inner list length.
   - `'a list list -> 'a list list -> 'a list list`

   d. Is your implementation tail-recursive?

- Second implementation is tail-recursive. Tail-recursive functions do not overflow since the compiler can discard the stack and return the result of the next function call.

2. In the following OCaml definitions, give each identifier's scope and type, or any errors:

```
let f f = f 1 1;;
(* top-level f:
 *   type: (int → int → 'a) → 'a
 *   scope: all of top-level
 *)
(* inner f:
 *   type: int → int → 'a
 *   scope: within function body
 *)
let g g = g 0.0 g;;
(* error:
 *   type of inner g: (float → 'a → 'b), recursive type with no start
 *)
let h h = h f "x";;
(* top-level h:
 *   type: ((int → int → 'a) → 'a → string → 'b) → 'b
 *   scope: all of top-level
 *)
(* inner h:
 *   type: (int → int → 'a) → 'a → string → 'b
 *   scope: within function body
 *)
```

3. a. In Java, is the subtype relation transitive?
    - Yes, eg. consider inheritance.

   b. In Java, is the graph of the subtype relation a tree?
    - No, because multiple interfaces can be implemented by a class or extended by another interface.
    - Also, primitive types such as `int` and `float` are not related to one another.
    - Note that a subclass is a subtype, but subtypes are not always subclasses.

4. Consider the following grammar for C declarations (fragment from midterm).

```
declaration:
  declaration-specifiers init-declarator-list? ';'
```

```
declaration-specifiers:
  type-specifier declaration-specifiers?
  ...

init-declarator-list:
  ...

init-declarator:
  declarator
  declarator '=' initializer

declarator:
  pointer? direct-declarator

direct-declarator:
  ID
  '(' declarator ')'
  direct-declarator '[' INT ']'
  direct-declarator '(' void ')'

pointer:
  '*' type-qualifier-list? pointer?

type-qualifier-list:
  type-qualifier-list? type-qualifier
```

4. a. What makes the grammar EBNF and not BNF?
    - It uses extended syntax such as `?` in the grammar.

   b. Give an example declaration that is syntactically correct but semantically incorrect.
    - `char char foo;`
    - `void bar = 1;`

   c. Suppose the grammar was changed by replacing the rule for `type-qualifier-list` with `type-qualifier type-qualifier-list?` . Does this cause any problems?
    - No, results in the same supported syntax and no differences.

   d. Suppose the grammar was changed by merging the rulesets for `declarator` and `direct-declarator` as follows. Does this cause any problems?

Merging rulesets:

```
declarator:
  pointer? declarator
  ID
  '(' declarator ')'
  declarator '[' INT ']'
  declarator '(' void ')'
```

4. d. Yes. Although an identical syntax is supported, the semantics don't match up between the grammars.
   - eg. for `**a[1][1]` , in the original grammar, the pointers were always handled before the remainder of the declarator.
   - In the new grammar, the pointers can be parsed together with different declarator rules, leading to ambiguity.

5. Suppose we write Java in a purely functional style, in that we never assign to any variables except when initializing them. In these purely-functional Java programs, is the Java Memory Model still relevant?

   - No, because we never write to any shared variables on the heap after the initialization, so race conditions are avoided.
   - However, this requires avoiding functions that update state such as `list.append` , which is not exactly specified in the question.