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 programming book
  - created TeX, a language for typsetting 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 has 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 fo 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 influences 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 using assignment and iteration
      · variables have current values, and order of execution is critical
    * **functional** languages like ML focus on using recursion and single-valued variables
    * **logic programming** languages express programs in terms of rules about logical inference
    * **object oriented** languages like Java focus on using objects, or a bundle of data and methods
    * language types can overlap
  - programming languages will also *evolve* and change over time
- *core* of programming languages:
  - principal 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
    * also 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 that its functions cannot return an array type
  - efficiency
    * eg. CPU, RAM, energy, network access
  - simplicity, ie. ease of *implementation*
  - convenience, ie. easy 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

Syntax and Grammar

- *translation* stages of a C source file:
  - 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
  - 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, and parentheses, brackets, and semicolons
    * eg. the children of a function call in the tree would include subtrees for the expression and parameters, and 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
  - the parse tree is *checked* for types, names, etc.
  - *intermediate* code is generated from the parse tree
    * convenient for the compiler writer
  - intermediate code is turned into assembly code
  - the assembly code is turned into object code by the assembler
  - the object code is turned into an executable by a linker
  - the executable is run by the loader
- 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 match declaration to use
  - 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`, $\neq$
  - 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
  - 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, 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 use CFGs:
  - Fortran, predates CFGs
  - typedefs in C allow for change 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 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 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 + 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
- **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 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 a subset of the grammar for the C standard:
    - <stmt> ::= ; | break ; | continue ; | return ;
    - <stmt> ::= return <expr> ; | <expr> ; | { <stmts> }
    - <stmt> ::= while (<expr>) <stmt> | do <stmt> while (<expr>) ; | 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
    - <stmt> ::= if (<expr>) <stmt1> else <stmt>
        * <stmt1> is just like <stmt>, except that it doesn't allow the elseless if
        * reorganize the grammra as follows:
            · <stmt> ::= <stmt1> | if (<expr>) <stmt>
            · <stmt1> ::= ... all previous <stmt> rules except the elseless if
- 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 **abstract syntax tree (AST)**
        * AST corresponds to ambiguous grammars where the *compiler* always does the "correct" parse
        * AST is simpler, and takes less memory

            * can 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:

```c
int g(void) {
  return (a = 1) + (a = 2); // This statement has undefined behavior!

  // This is a runtime problem that has nothing to do with syntax.
  // The problem stems from competing side effects within expressions.
}
```

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 <-> 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
    * $\sum_{0 \leq i < n} f(i) = f(0) + f(1) + ... + f(n-1) = \sum(0, n, f)$
    * $\sum$ is a function that takes a function `f` as its argument
    * 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
  - 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

ML

- **ML** is a popular functional language
  - has a *standard* dialect (SML) and an *object-oriented* dialect (OCaml)
  - this chapter uses the SML-NJ dialect
- properties of ML:
  - **functional**, functions are *first-class objects*, ie. they can be passed into functions and treated as any other variabels
  - **immutable**, variables (including lists) cannot be modified
  - uses **type inference** to automatically choose types
    * thus functions and operators can't have *overloaded* definitions
  - never does **implicit** casts
  - 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
    * `f a+1` is the same as `(f a) + 1`, `f g 1` is not the same as `f(g(1))`
- 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 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 (*cons* or construct operator)
    - `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 the 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 the 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 the 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

- 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* 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 Syntax

- 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`
  - `[1; 2; 3];;` has type `int 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 ...`

- 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`
  - the `rec` is required for a recursive variable definition

`fib` in SML:

```
fun fib 0 = 0
  | fib 1 = 1
  | n = fib (n-1) + fib (n-2)
```

`fib` in OCaml with full `fun`:

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

`fib` in OCaml with `function` syntactic sugar for matching:

```
let rec fib = function
    0 -> 0
```

```
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2)
;;
```

`fib` in OCaml with fully abbreviated syntactic sugar:

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

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