CSE 3302 Notes 2: Four-and-a-Half New Friends

(Last updated 11/11/15 8:44 AM)

References:

Crockford: 1, 10
Dybvig: 2
Wirth: 1-3

New?

Pascal - 1970, *Algorithms + Data Structures = Programs* - 1975
(`http://en.wikipedia.org/wiki/Algorithms_%2B_Data_Structures_%3D_Programs`
`http://www.ethoberon.ethz.ch/WirthPubl/AD.pdf`)

Scheme - 1975, *The Structure & Interpretation of Computer Programs* - 1985
(`http://mitpress.mit.edu/sicp/`)

JavaScript - 1994, *JavaScript: The Definitive Guide* - 1996 (in 6th edition)
(`http://site.ebrary.com.ezproxy.uta.edu/lib/utarlington/detail.action?docID=10763621`)

C++ - 1983, *The C++ Programming Language* - 1985 (in 4th edition)

2.1. Pascal and PL/0

Superficially, many similarities to C

ACM Computing Surveys 14 (1), 1982, `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=356869.356872`

Biggest philosophical difference - Pascal has stronger typing

Pascal	C
`integer`	`int`
`boolean`	`int` (or `char`)
`set` (bit vector)	`int`(s) + array + bit ops + code
array bounds are constant	0 ... ? (and row-pointer)
`new/dispose` (language)	`malloc/free` (library/casts/`&` operator)
pointer *equality* test	pointer *order* tests (and arithmetic)
1 based	0 based

Nested functions/procedures/scopes	No nesting of functions, but blocks may have declarations (gcc supports function nesting, LLVM does not)

Other differences:

`;` is a separator	`;` is a terminator
`:=  =`	`=  ==`
Unnatural (?) precedence with few levels (expressed in formal syntax)	Many (natural?) precedence levels (expressed by arity/associativity/precedence table)
Exhaustive evaluation of logical expressions?	“Short-circuit” evaluation
`function`/`procedure`	Not distinguished beyond `void` type for `return` value
Arrays may be passed by value or by reference	Arrays always passed by reference (every passed argument is “small”)
Return at exit using value assigned to name of function	`return` anywhere
Limited `for` loop (always ±1 step)	`for` and `while` are syntactic variants
No `break`/`continue`	
No preprocessor	
No separate compilation(?)	abstract data types and alternate implementations easily achieved using header files (`https://en.wikipedia.org/wiki/Opaque_pointer#C`)

Examples: `http://ranger.uta.edu/~weems/NOTES3302/GCD/gcd.pas`
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/table.pas`
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/table.ok.pas`

(Also, stable marriages code in C, Pascal, and Pascal-S.
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/STABLE/`
`http://en.wikipedia.org/wiki/Stable_marriage_problem`
Aside: `http://www.nrmp.org/`)

So, why learn this?

“Perhaps the most important single point is that the structure of (a) language is mirrored in the structure of its compiler and that its complexity - or simplicity - intimately determines the complexity of its compiler”. N. Wirth, *Algorithms + Data Structures = Programs*, p. 280. (Also see his Turing Award Lecture `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2786.2789`)

Wirth has embraced *recursive-descent* as a form of top-down parsing:

Straightforward to continue parsing in presence of errors
Leads to simple one-pass compilers (for appropriate languages)
Most easily implemented in a PL with nested functions

The Pascal-S parser/interpreter (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/pascals.pas`) is written in Pascal: 2000 lines

The Pascal-S subset is suitable for an introductory programming course.

Understanding `pascals.pas` will get you up-to-speed
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/Wirth-PascalS.pdf`
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/Wirth-PascalS.txt`

Input is appended to the end of the source file:
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/gcd.s.pas`
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/matmult.s.pas`

The simpler, but similarly implemented PL/0 (
`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/plzero.pas`) 450 lines

The base language (i.e. from *Algs + DSs = Progs*) is only large enough to demonstrate translation and stack machine concepts.

No arrays, functions, arguments, minimal control structures, I/O, only integer variables

Allows nesting of procedures and recursion.

Examples: `simple.plz`, `ex.plz`, `ex.err.plz`, `factorial.plz`, `recurse.plz`, `recurse2.plz`

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/adsprgs.pl0.pdf`
contains pages from Wirth's book

Baseline (`http://ranger.uta.edu/~weems/NOTES3302/BASELINE/`) 1900 lines of JavaScript

Extended with: comments, else option on if statement, simple integer I/O, passing of integers by value to procedures, C-like 1-d arrays, limited canvas facilities, instruction stepping, breakpoints, and IDE

`http://ranger.uta.edu/~weems/NOTES3302/LAB4SPR13/` code in Pascal and JavaScript does not have arrays and canvas (last version done in both languages)

2.2. SCHEME (`http://racket-lang.org/` `http://www.scheme.com/tspl4/start.html#./start:h0`)

1. Dybvig, 2.1 and 2.2, use the bottom (interactions panel, REPL) area in Racket to calculate various expressions. Look up `sqrt` in Racket documentation and compute with numbers of different types.

2. Dybvig, 2.1 and 2.2, cut some of your expressions from the interactions panel, paste into the definitions panel, save as a file, then "Run". Also, hit "Debug" and use "Step" (the Racket alternative to Chez Scheme's `trace` described in 2.8).

3. Lists are to Scheme what arrays are to other languages . . . `cons`, `car`, `cdr`, `null?`, `pair?` are the start. `list`, `list*`, `append`, `flatten`, `sort`, `reverse` are very convenient, but are also good coding exercises.

4. Be functional (expression-oriented, not state-oriented). **`let`** (for local values, 2.4) is OK, **`set!`** (changing a value, 2.9) can be trouble (and a bad habit). `let` creates a new scope, `define` introduces a name into the current scope. (Notes 4 includes other variations of `let`. Chapter 4 of Dybvig is intense on relating these variations to `lambda`).

5. Get experience with the three basic recursive "eaters": a number, a list of atoms (lat), and a general S-expression.

```
(define (gaussSum num)
  (cond
    ((zero? num) 0)
    (else (+ num (gaussSum (- num 1))))))

(define (gaussSum num)
  (if (zero? num)
      0
      (+ num (gaussSum (- num 1)))))

(gaussSum 0)
(gaussSum 5)

(define (countAtomsLat lat)
  (cond
    ((null? lat) 0)
    (else (+ 1 (countAtomsLat (cdr lat))))))

(define (countAtomsLat lat)
  (if (null? lat)
      0
      (+ 1 (countAtomsLat (cdr lat)))))

(countAtomsLat '())
(countAtomsLat '(a b c))

(define (countAtomsSexp l)
  (cond
    ((null? l) 0)
    ((pair? l)
     (+ (countAtomsSexp (car l)) (countAtomsSexp (cdr l))))
    (else 1)))
```

```
(countAtomsSexp 'a)
(countAtomsSexp '((())))
(countAtomsSexp '(a b c))
(countAtomsSexp '(a b (c d)))
(countAtomsSexp '(((a b (c (d)) (e f) () () (g)))))
```

6. **`quote`** ⇒ `'` (to avoid having Scheme attempt to evaluate)

7. Carefully defined helper functions are invaluable, especially to avoid passing an unchanging argument.

8. Get familiar with Racket facilities for formatting and "parenthesis management".

9. You haven't lived until you return a function . . .

10. **`lambda`** is usually unnecessary when defining functions:

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))

(define (atom? x)
  (and (not (pair? x)) (not (null? x))))
```

11. **`lambda`** gives you an anonymous function as a first-class object.

```
((lambda (x) (and (not (pair? x)) (not (null? x)))) '(a b c d))
```

returns `#f`

12. Using `{ }` and `[ ]` improve readability (a little):

```
([lambda (x) {and [not (pair? x)] [not (null? x)]}] '(a b c d))
```

13. Respect *The Law of Cons*:

```
> (cons (cons (cons (cons 1 2) 3) 4) 5)
'((((1 . 2) . 3) . 4) . 5)
```

Use:

```
> (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 '())))))
'(1 2 3 4 5)
```

instead (or use `(list 1 2 3 4 5)` or the obscure `((lambda x x) 1 2 3 4 5)` from `http://www.scheme.com/tspl4/start.html#./start:s70`)

14. An identity function that uses `display` or `displayln`, but returns its argument, is useful for debugging or tracing:

```
(define (print x)
  (display "My latest bug is: ")
  (displayln x)
  x)
```

16. 2.7, predicates and conditionals.

   a. Only `#f` (and aliases, `#F #false`) is falsy (in C, only `0` is falsy), everything else is truthy. Leads to:
```
> (and null null)
'()
> (and null '(a b c))
'(a b c)
> (or '(a b c) null)
'(a b c)
```
   b. `cond` is the multiway branch conditional. It is a special form that "delays" evaluation unlike the usual Scheme "inside-out" strategy.
   c. `(if x y z)` is like `x ? y : z` in the C family. It is equivalent to:
```
(cond
  (x y)
  (else z))
```
   d. Writing your own `length` function is a classic exercise.
   e. Short-circuit evaluation for `and` and `or` is also part of C family `&&` and `||`. This is connected to b. Try `(+ 1 (/ 1 0))` and `(or 1 (/ 1 0))` in the REPL.
   f. `eq?` is a shallow test (in the sense of Java comparing references).
   `equal?` returns `#f` if types are different (deep comparison). See 6.2

The core of Scheme is given at: `http://www.scheme.com/tspl4/further.html#./further:h1`

Give Scheme code for a function `summa` to compute the summation below. j and k are positive integers. p is a non-negative integer. (If $j > k$, then the result is 0. Helper functions are allowed! Do NOT use math library functions)

$$\sum_{i=j}^{k} i^p = \texttt{(summa j k p)}$$

```
(define (power i p)
  (cond
    ((= p 0) 1)
    (else
     (* i (power i (- p 1))))))

(define (summa j k p)
  (cond
    ((<= j k) (+ (power j p) (summa (+ j 1) k p)))
    (else 0)))
```

Give Scheme code to test if a single-level unordered list (i.e. just atoms, you may designate a single type for these) has duplicate elements. `#t` or `#f` will be returned. (Quadratic time is fine.)

```
(define (member? a lat)
  (cond
    ((null? lat) #f)
    (else (or (eq? (car lat) a)
              (member? a (cdr lat))))))

(define (duplicate l)
  (cond
    ((empty? l) #f)
    (else (or (member? (car l) (cdr l))
              (duplicate (cdr l))))))
```

Give Scheme code to insert into an ordered list of numbers.

```
(define (insertOrdered num lat)
  (define (help lat)
    (if (null? lat)
        (list num)
        (let ((head (car lat))
              (tail (cdr lat)))
          (cond
            ((= head num) lat)
            ((< head num) (cons head (help tail)))
            (else (cons num lat))))))
  (help lat))
```

Give Scheme code for the `mirror` function. This function will reverse a list, but will also *recursively reverse* any nested sub-lists. If its input is an atom, then the atom is to be returned. (Helper functions are allowed!)

```
(define (mirror l)
  (define (help l tail)
    (cond
      ((null? l) tail)
      ((pair? l) (help (cdr l) (cons (mirror (car l)) tail)))
      (else
       l)))
  (help l '()))
```

Give Scheme code to replace each atom in an S-expression by the number of atoms that *precede* it when the S-expression is printed. The argument will always be a list - no error checking is required.

```
> (replace '(1 2 3 4))
'(0 1 2 3)
> (replace '(1 (2 3) 4 5))
'(0 (1 2) 3 4)
> (replace '((((1 (2 3) 4 5)))))
'((((0 (1 2) 3 4))))
> (replace '(3 2 1))
'(0 1 2)
```

```
> (replace '((((1 (2 3) 4 5)) ((((1 (2 3) 4 5)) (3 2 1))))))
'((((0 (1 2) 3 4)) ((((5 (6 7) 8 9)) (10 11 12)))))
> (replace '(((x (y (z) (c (b (a))))))))
'(((0 (1 (2) (3 (4 (5)))))))
```

Avoiding `set!` but taking quadratic time:

```
(define (atomCount l)
  (cond
    ((null? l) 0)
    ((pair? l) (+ (atomCount (car l))
                  (atomCount (cdr l))))
    (else 1)))

(define (replace l)
  (define (replacer l prevCount)
    (cond
      ((null? l) '())
      ((pair? l) (cons
                  (replacer (car l) prevCount)
                  (replacer (cdr l) (+ prevCount
                                      (atomCount (car l))))))
      (else prevCount)))
  (replacer l 0))
```

Using `set!` to take linear time:

```
(define (replace l)
  (let
      ((prevCount 0))
    (define (replacer l)
      (cond
        ((null? l) '())
        ((pair? l) (cons
                    (replacer (car l))
                    (replacer (cdr l))))
        (else
         (set! prevCount (+ 1 prevCount))
         (- prevCount 1))))
    (replacer l)))
```

2.3. JAVASCRIPT

"It is Lisp in C's clothing." (to the level of http://www.amazon.com/dp/0262560992/, but see http://matt.might.net/articles/)

The superficial: http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/SEATING/seating.html
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/emoji.scroll.html

http://www.w3schools.com/js/ http://eloquentjavascript.net

https://developer.mozilla.org/en-US/docs/Web/JavaScript

1. DOM/HTML is the shark . . .

2. Browser issues can be irrational.
   (see `https://developer.mozilla.org/en-US/docs/Mozilla_event_reference/keypress`)

3. JavaScript applications are event-driven ("reactive") with one thread.
   Non-preemptive, run-to-completion (which leads to
   `https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers`).
   (Ajax, jQuery, Node.js are outside our scope.)

   Simple *synchronous* input may be achieved using `prompt()` or `confirm()`.

4. Prototypal inheritance/delegation approach to objects will be covered in Notes 10.

   `http://ranger.uta.edu/~weems/NOTES3302/LAB/15SUM/LAB4/`

5. Values have types, variables do not. Function parameter lists are very flexible.

6. Five types: number, string, boolean, object, `function`, (and `undefined/null)`.

   JavaScript numbers are 64-bit IEEE 754. Push these and they will push back.

   Strings are immutable. + is the concatenation operation.

   "Falsy": `false` `undefined` `null` `0` `-0` `NaN` `""`
   "Truthy": `true` everything else not on previous line

7. Functions as first-class objects are useful for controlling code complexity.

8. JavaScript arrays are imposters. They are hashed maps for strings as property names.
   When subscripting, it is like the subscript value gets converted to a string.

9. An object may be created with properties, but more may be added by using assignments.

10. Exceptions can be of any type. Uncaught exceptions end up at the "console".

11. Many errors fail silently, including misspelled names (except when calling a function).

`http://ranger.uta.edu/~weems/NOTES3302/CANVAS/mouseFollow.html`

`http://ranger.uta.edu/~weems/NOTES3302/CANVAS/mouseText.html`

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/events.html`

HTML elements (and JavaScript one-liners):

Canvas Radio Buttons Text Areas Coordinates

JavaScript elements

Globals - by intent . . . or accident? (like misspelled variable names)
Anonymous functions for event handlers
Getting a `static` variable - closures and iifes (Notes 5)
Property names can really be getters/setters `http://site.ebrary.com.ezproxy.uta.edu/lib/utarlington/reader.action?ppg=146&docID=10763621&tm=1434745663024`

2.4. C++

“There are only two kinds of languages: the ones people complain about and the ones nobody uses.”

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”

`http://www.amazon.com/Effective-Modern-Specific-Ways-Improve/dp/1491903996/`

`http://www.amazon.com/C-Programming-Language-4th/dp/0321563840/`

`http://isocpp.org/wiki/faq/wg21`

2.4.1 Suggestions for C Programmers

[1] Don’t think of C++ as C with a few features added.

[2] Don’t write C in C++.

[3] Use the C++ standard library as a teacher of new techniques (observe parallel features to C).

[4] Macro substitution is almost never necessary in C++. (Aside on demacrofication: `http://www.stroustrup.com/icsm-2012-demacro.pdf`)

[5] Declare variables where they may be initialized immediately.

[6] Don’t use `malloc`, BUT also avoid “naked” `new` and `delete` to respect RAII to avoid situations such as exceptions that lead to resource leaks.

[7] Avoid type system workarounds (`void*`, unions, casts)

[8] Use the C++ library versions of arrays, strings, and vectors.

2.4.2 Suggestions for Java Programmers

[1] Don’t mimic Java style in C++. Often functions are outside classes ([14] “binary methods” or freestanding functions in namespaces) and overloading is used.

[2] Don’t fall back to C.

[3] As above for C.

[4] Don’t use a unique base class for all classes.

[5] Use local and member variables rather than reference and pointer variables.

[6/7] Beware of the differences between Java references and C++ pointers and references (which may only be intialized and are never NULL)

[10] Much of the garbage collection should correspond to RAII scoping.

[13] Avoid "naked" `new` and `delete`, often by using STL containers

2.4.3 Streams for I/O (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/notes02.stream.cpp`)

With minimal effort:

1. ostreams replace `stdout` through `printf` by `cout` with <<
2. istreams replace `stdin` through `scanf` by `cin` with >> (but avoiding much of the pain)
3. Overloading of << and >> (binary operators whose result is always the stream)

2.4.4 Parameters (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/notes02.parameter.cpp`)

C - mechanism is specified by the language with some control on the calling side (e.g. use `&` to get pointer)

1. Scalars (including pointers) are passed by value (i.e. copied).
2. Arrays are passed by reference (through a pointer to element of 0, passed by value)
3. Structs are passed by value.

C++ - can specify `&` for parameter on the function side

1. Simplies situations like passing an `int` for modification. (Can't confuse with array or use pointer arithmetic.)
2. `const` may be used to prevent modification.
3. No notion of NULL for a reference's value.
4. No notion of modification ("reseating") for the reference.

2.4.5 Strings (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/notes02.string.cpp`)

C strings (`<string.h>`, `char*`) with NULL terminator are (have to be?) available

1. CACM, Sept. 2011, "The Most Expensive One-Byte Mistake", `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=1995376.1995391`
2. Are unlikely to be significantly faster, with potential for memory leaks.
3. Are not as general (and will cram your cranium with low-level library calls).

C++ strings (`<string>`, `std::string`)

1. Very flexible in terms of being a sequence, rather than just `chars`.
2. Literal strings are C strings.
3. Many useful overloaded operators (with many connections to JavaScript).
4. When low-level access is needed, code may be written in an iterator style.

2.4.6 Macros vs. Templates

```
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/Qmacro.c
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/Qtemplate.cpp
```

1. C/C++ macros are handled by preprocessor before compilation.
2. # indicates "stringify", ## indicates concatenate into one token.
3. C/C++ macros are processed without language considerations (unlike Scheme).
4. Templates bring general computation to compile time.
5. Simple templates are used to create instances of function with different parameter signatures.
6. More complicated templates generate class structures.

2.4.7 Operator Overloading

```
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/complex.cpp
```

1. Input and output stream overloading.
2. Tracks number of instances.
3. Multiple constructors.
4. Basic arithmetic and equality comparisons have been overloaded.

```
http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/rational.cpp
```

1. Represents rational number in reduced form.
2. Makes explicit conversions to `double` and `int` available.
3. Includes ordering comparisons.

2.4.8 Resource Allocation is Initialization (RAII)

1. Design principle to avoid resource leaks, especially for exception handling.

2. Avoids overhead of general garbage collection.

3. Explicit destructor calls are avoided by scoping discipline.

4. STL containers are important option for objects outside simple scoping.