CS 326 Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Midterm Review

Midterm review

- Midterm exam structure
  - Theory questions
    - True/false
    - Multiple choice
    - “Regular” questions (justify the answer)
  - Problems
    - Given a language, write a regular expression to describe it
    - Given a language, write a context-free grammar to describe it
    - Given a grammar and a string, show a parse tree / derivation
    - Write a recursive function in Scheme
    - Given a program, what does it print with static / dynamic scoping?
    - Write a tail-recursive function in Scheme

Midterm review

- Midterm exam content
  - Chapter 1 – Introduction
  - Chapter 2 – Programming language syntax
  - The Scheme programming language
  - Chapter 3 – Names, scopes and bindings
  - Chapter 6 – Control flow

Introduction

- Chapter 1 – Introduction
- Programming languages characterized by:
  - Syntax – what a program looks like
  - Semantics – what a program means
  - Implementation – how a program executes
- Spectrum of languages
- Machine language vs. assembly language vs. high-level language
- Compilation vs. interpretation
- Phases of compilation

Spectrum of Languages

- Imperative ("how should the computer do it?")
  - Von Neumann: Fortran, Basic, Pascal, C
    - Computing via "side-effects" (modification of variables)
  - Object-oriented: Smalltalk, Eiffel, C++, Java
    - Interactions between objects, each having an *internal state* and *functions* which manage that state

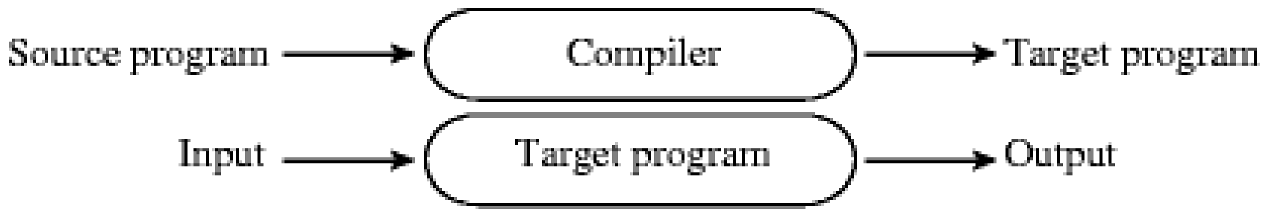
- Declarative ("what should the computer do?")
  - Functional: Lisp, Scheme, ML, Haskell
    - Program ↔ application of functions from inputs to outputs
    - Inspired from *lambda-calculus* (Alonzo Church)
  - Logic, constraint-based: Prolog
    - Specify constraints / relationships, find values that satisfy them
    - Based on *propositional logic*

Spectrum of Languages

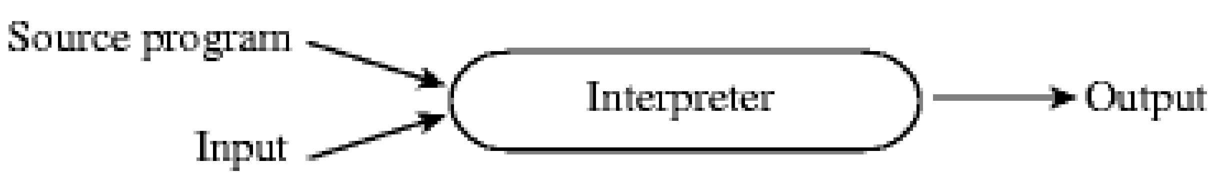
- Machine language
  - Sequence of bits that directly controls the processor
  - Add, compare, move data, etc.
- Assembly language
  - Mnemonic abbreviations
  - Translated by an *assembler*
  - Still machine-dependent
- High-level languages (the first: Fortran, Lisp)
  - Machine-independent language
  - No more 1-to-1 correspondence to machine language
  - Translated by a *compiler* or *interpreter*

Compilation and Interpretation

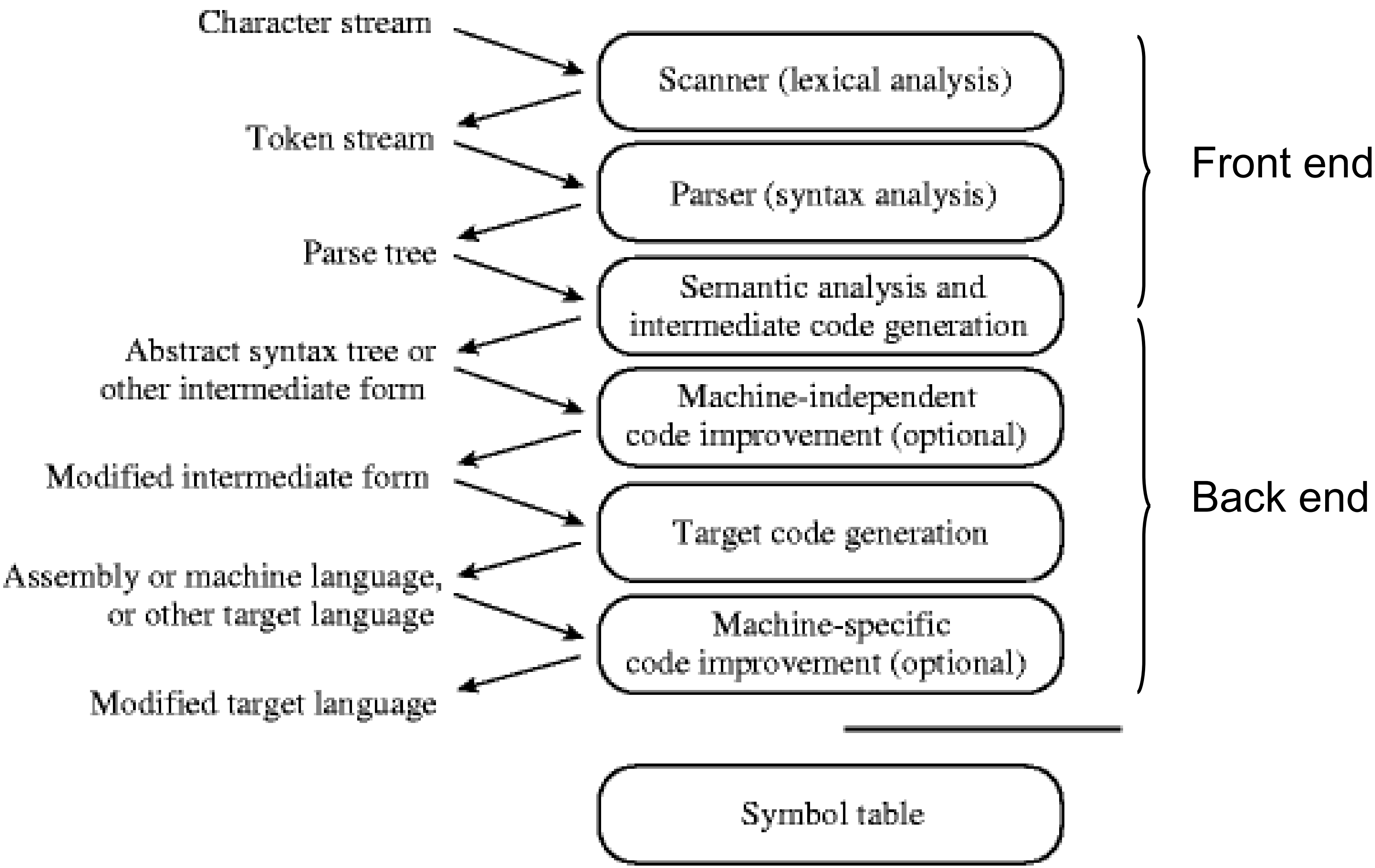
- *Compiler* translates into target language (machine language), then goes away
- The target program is the locus of control at execution time



- *Interpreter* stays around at execution time
- Implements a virtual machine whose machine language is the high-level language



Phases of Compilation



Programming Language Syntax

- Chapter 2 – Programming language syntax
- Regular expressions, context-free grammars
- Derivations, parse trees
- Scanners, parsers

Classification

- Chomsky hierarchy (incomplete):

Language	Generator	Acceptor	Compile phase
Regular	Regular grammar	Finite automaton	Lexical analysis (scanning)
Context-free	Context-free grammar	Push-down automaton	Syntax analysis (parsing)

- Regular languages are a subset of context-free ones

Grammars

- Rules (productions): A finite set of replacement rules:

  <sentence> → <subject> <predicate>

  <subject> → <article> <noun>

  <predicate> → <verb> <article> <noun>

  <verb> → ran | ate

  <article> → the

  <noun> → boy | girl | cake

- Nonterminals: a finite set of symbols:

  <sentence> <subject> <predicate> <verb> <article> <noun>

- Terminals: a finite set of symbols:

  the, boy, girl, ran, ate, cake

- Start symbol: one of the nonterminals:

  <sentence>

Parse Trees and Derivations

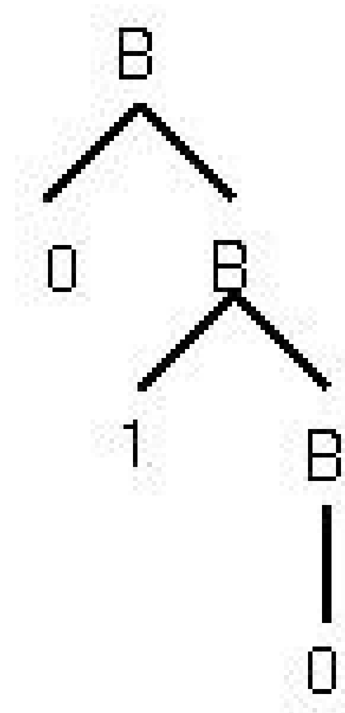
- Grammar:

  $B \rightarrow 0B \mid 1B \mid 0 \mid 1$

- Generate 010

- Derivation:

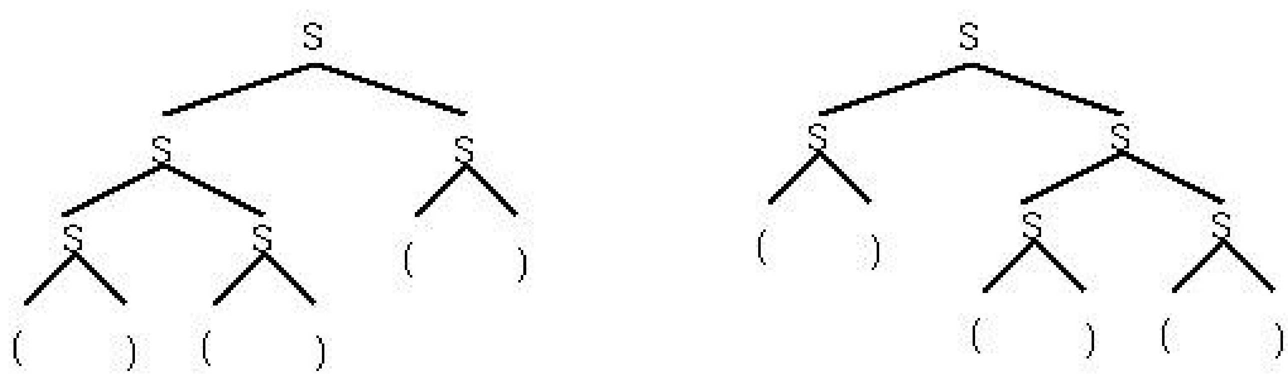
  $\underline{B} \Rightarrow 0\underline{B} \Rightarrow 01\underline{B} \Rightarrow 010$



Parse tree

Ambiguous Grammars

- Grammar: S -> SS | (S) | ()
- Generate ()()()
- Ambiguous grammar ⇔ one string has multiple parse trees



Regular Expressions

- Definition:

  - A regular expression R is either:
    - a character
    - the empty string ε
    - R_1 R_2 (concatenation)
    - R_1 | R_2 (alternation)
    - R_1* (repetition zero or more times - Kleene closure)

- Also used: R+ (repetition one or more times) ↔ R R*

Regular Expressions

- Language:

  set of strings over alphabet {a,b} that begin with at least two a's, and end with at least two b's

- Regular expression:

  $aa\ (a \mid b)^*\ bb$

- Language:

  set of strings over alphabet {a} that contain an odd number of a's

- Regular expression:

  $(aa)^*\ a$

Grammars

- Language:

  set of strings over alphabet {*a*,*b*} that begin with at least two *a*'s, and end with at least two *b*'s

- Grammar:

  $S \rightarrow aa\ P\ bb$

  $P \rightarrow P\ a \mid P\ b \mid \varepsilon$

- Language:

  set of strings over alphabet {*a*} that contain an odd number of *a*'s

- Grammar:

  $S \rightarrow P\ a$

  $P \rightarrow P\ aa \mid \varepsilon$

Grammars

- Language:

  $\{a^m b^n a^{m+n} \mid m \geq 0 \text{ and } n \geq 1\}$

  Hint: first rewrite as $a^m b^n a^n a^m$

- Grammar:

  $S \rightarrow aSa \mid R$

  $R \rightarrow bRa \mid ba$

Scanning and Parsing

- Scanner
  - ignores white space (blanks, tabs, new-lines)
  - ignores comments
  - recognizes tokens
  - implemented as a function that returns next token every time it is called

- Parser
  - calls the scanner to obtain tokens
  - builds parse tree
  - passes it to the later phases (semantic analysis, code generation and improvement)

The Scheme Programming Language

- The Scheme programming language
- Expressions, atoms, lists
- Evaluation, preventing evaluation, forcing evaluation
- List operations
- Boolean and conditional expressions
- Functions (lambda expressions)

Evaluation

- Expressions can be atoms or lists
- Atom: number, string, identifier, character, boolean
- List: sequence of expressions separated by spaces, between parentheses

- Constant atoms - evaluate to themselves
- Identifiers (symbols) - evaluate to the value bound to them
- Lists - evaluate as "function calls"

- Preventing evaluation – use quote
- Forcing evaluation – use eval

Functions

- Bind a name to a function:

```
(define square (lambda (x) (* x x)))
```

- Equivalent short-hand notation (typical way to use it):

```
(define (square x) (* x x))
```

- Now call the function:

```
(square 3)        => 9
```

Recursion

- How do you solve a problem recursively?

- Do not rush to implement it
- Think of a recursive way to describe the problem:
  - Show how to solve the problem in the general case, by decomposing it into similar, but smaller problems
  - Show how to solve the smallest version of the problem (the base case)
- Now the implementation should be straightforward (in ANY language)

Recursion

- Check membership in a list:

```
(define (member x L)
   (cond
         ((null? L)                    #f)
         ((equal? x (car L))           #t)
         (else                         (member x (cdr L)))))
```

- Boolean operators in Scheme use short-circuit evaluation. Rewrite member without using if or cond:

```
(define (member x L)
   (and (not (null? L))
        (or (equal? x (car L))
                   (member x (cdr L)))))
```

Recursion

- (deep-delete V L) – return a list similar to L, but having all occurrences of V in L or in any sublist of L deleted

```
(define (deep-delete V L)
  (cond
    ((null? L)              '())
    ((list? (car L))        (cons (deep-delete V (car L))
                          (deep-delete V (cdr L))))
    ((equal? V (car L))  (deep-delete V (cdr L)))
    (else                   (cons (car L)
    (deep-delete V (cdr L))))))
```

Recursion

- General approach:
  - When recurring on a list lst, ask two questions about it: (null? lst) and else
  - When recurring on a number n, ask two questions about it: (= n 0) and else

  - When recurring on a list lst, make your recursive call on (cdr lst)
  - When recurring on a number n, make your recursive call on (- n 1)

Other Expressions

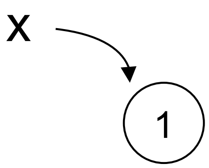
- Local definitions – let, let*, letrec
- Higher order functions – map, apply
- Sequencing – begin

- Input-output – read and display
- Assignment – set!, set-car!, set-cdr!

Internal Structure of Expressions

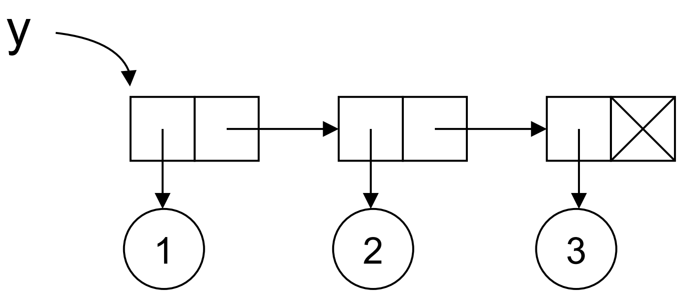
- Implicitly, all variables are pointers that are bound to values
  - Atom values:

    (define x 1)



  - List values:

    (define y '(1 2 3))



- Each element in the list is a cons cell, which contains:
  - a pointer to a value
  - a pointer to the next cons cell

Names, Scopes and Bindings

- Chapter 3 - Names, Scopes and Bindings
- Binding time
- Early vs. late binding
- Object vs. binding lifetime
- Storage allocation mechanisms
- Scope rules (static vs. dynamic scoping)
- Binding rules (deep vs. shallow binding)

Binding Time

- Early binding
  - associated with greater efficiency
  - compiled languages tend to have early binding times
  - the compiler analyzes the syntax and semantics of global variable declarations only once, decides on a data layout in memory, generates efficient code to access them

- Late binding
  - associated with greater flexibility
  - interpreted languages tend to have late binding times

Object Lifetime and Binding Lifetime

- Lifetime - the time interval between creation and destruction
- Both objects and bindings have their own, possibly distinct lifetimes

- If an object outlives its (only) binding it's garbage

```
p = new int ;
p = NULL ;
```

- If a binding outlives its object it's a dangling reference

```
p = new int ;
r = p ;
delete r ;
```

Storage Management

- Storage allocation mechanisms:
  - Static - each object is given an address before execution begins and retains that address throughout execution
  - Dynamic
    - Stack - objects are allocated (on a stack), and bindings are made when entering a subroutine
    - Heap
      - Explicit - allocated and deallocated by explicit directives at arbitrary times, specified by the programmer
      - Implicit - allocation and deallocation are implicit (transparent for the programmer); requires garbage collection

Storage Management

- Dynamic heap allocation

- Maintain a single linked list of heap blocks that are not currently used (the free list)
  - Strategies:
    - First fit – select the first block in the list that is large enough to satisfy the allocation request
    - Best fit – select the smallest block in the list that is large enough to satisfy the allocation request

- Maintain separate lists for blocks of different sizes
  - Strategies:
    - Buddy system
    - Fibonacci heap

Scope Rules

- Languages can be statically or dynamically scoped
- Statically (also called lexically) scoped
  - Bindings are determined by examining the program text
  - Can be determined at compile time
  - Examples: C, Pascal, Scheme
- Dynamically scoped
  - Bindings depend on the flow of control at run time (on the dynamic sequences of calls)
  - Choose the most recent active binding (at run time)
  - Examples: APL, Snobol, early Lisp

Scope Rules

- Referencing environment
  - represents the set of active bindings at a given point in program execution
  - determined by static or dynamic scope rules

- Deep vs shallow binding
  - assume a function is passed/stored/returned, and later called
  - when the function is called, what referencing environment will it use?
    - deep binding - use the environment from the moment when the function was passed/stored/returned
    - shallow binding - use the environment from the moment of function call

Static vs. Dynamic Scoping

- Example - static vs. dynamic scope rules

```
a : integer
procedure first
    a := 1
procedure second
    a : integer
    first()
// main program
a := 2
second()
write(a)
```

- What is written if the scoping rules are:
  - static? 1
  - dynamic? 2
- If static scoping - a in procedure first refers to the global variable a (as there is no local declaration of a in first). Therefore, the global a is changed to 1
- If dynamic scoping - a in procedure first refers to the local variable a declared in procedure second (this is the most recent binding for a encountered at run time, as first is called from second). Therefore, the local a is changed to 1, and then destroyed when returning from second

Symbol Tables

- Statically scoped languages
  - LeBlanc-Cook symbol table
    - uses a hash table for symbols, and a scope stack
- Dynamically scoped languages
  - Association list
    - uses a stack of pairs name / information about it
    - the current binding is the highest in the stack (most recent at run-time)
  - Central reference table
    - keeps a central table (dictionary) with a slot for each name
    - at each slot keeps an association list (stack) for that name

Control Flow

- Chapter 6 – Control flow
- Expression evaluation
- Structured vs. unstructured flow
- Sequencing
- Selection
- Iteration
- Procedural abstraction
- Recursion

Control Flow

- Sequencing – relevant only with side-effects
- Expressions and statements
  - Expression-oriented languages

    a := if b < c then d else e;
  - Statement-oriented languages
- Assignments
  - L-values
  - R-values
- Combination assignments (b.c[3].d += 3)
- Variables
  - Value model
  - Reference model

Control Flow

- Order of applying operators
  - Associativity and precedence
- Order of evaluating operands
  - Usually not specified
  - Allow compiler to reorder for code improvement
- Short-circuit evaluation
  - Boolean expressions
  - Evaluate only as much as needed
- Unstructured flow - goto
- Structured flow – lexically nested blocks, selection if…then…else, iterations for, while

Case/Switch Statements

- Alternative syntax for a special case of selection (from a set of discrete constants)
- Example (Modula-2):

```
CASE expr of
    1:        clause_A
|   2, 7:     clause_B
|   3..5:     clause_C
|   10:       clause_D
    ELSE      clause_E
END
```

- Specify values on each arm - they must be discrete and disjoint:
  - constants (1)
  - enumerations of constants (2, 7)
  - ranges of constants (3..5)

- Implementation
  - sequential testing (similar to if...then...else)
  - jump table (compute an address to jump in a single instruction)

Iteration

- Mechanisms
  - Enumeration-controlled loops (for) - executed once for every value in a given finite set
  - Logically-controlled loops (while) - executed until some Boolean condition changes

- Enumeration-controlled loops (for):
  - Changes to the loop index (i), step or bounds (first and last)
  - Empty bounds
  - Direction of step
  - Jumps in and out of the loop

Recursion

- Equally powerful to iteration
- Efficient implementation – tail recursion
- Compute greatest common divisor:

```
(define gcd (lambda (a b)
  (cond ((= a b) a)
        ((< a b) (gcd a (- b a)))
        ((> a b) (gcd (- a b) b)))))
```

- The function is tail recursive
  - no additional computation follows the recursive call
  - returns what the recursive call returns

- The compiler will "rewrite" as:

```
gcd (a b)
start:
  if a = b
    return a
  if a < b
    b := b - a
    goto start
  if a > b
    a := a - b
    goto start
```

Evaluation of Function Arguments

- When are the arguments evaluated?
  - Before being passed to the function (applicative-order evaluation)
    - in most languages
    - safer, more clear
  - Pass a representation of unevaluated parameters to the function; evaluate them only when needed (normal-order evaluation)
    - typical for macros
    - can be faster

Announcements

- Midterm on March 12