CS 242

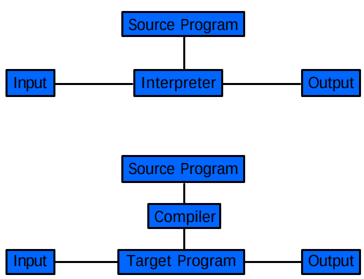
Fundamentals

John Mitchell

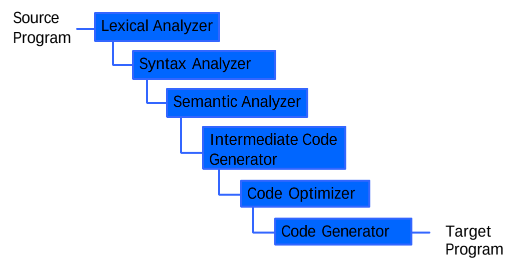
Syntax and Semantics of Programs

- Syntax
  - The symbols used to write a program
- Semantics
  - The actions that occur when a program is executed
- Programming language implementation
  - Syntax → Semantics
  - Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

Interpreter vs Compiler



Typical Compiler



See summary in course text, compiler books

Brief look at syntax

- Grammar

  e ::= n | e+e | e–e

  n ::= d | nd

  d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- Expressions in language

  e → e–e → e–e+e → n–n+n → nd–d+d → dd–d+d

  → … → 27 – 4 + 3

Grammar defines a language

Expressions in language derived by sequence of productions

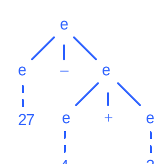
Most of you are probably familiar with this already

Parse tree

- Derivation represented by tree

  e → e–e → e–e+e → n–n+n → nd–d+d → dd–d+d

  → … → 27 – 4 + 3



Tree shows parenthesization of expression

Parsing

- Given expression find tree
- Ambiguity
  - Expression 27 – 4 + 3 can be parsed two ways
  - Problem: 27 – (4 + 3) ≠ (27 – 4) + 3
- Ways to resolve ambiguity
  - Precedence
    - Group * before +
    - Parse 3*4 + 2 as (3*4) + 2
  - Associativity
    - Parenthesize operators of equal precedence to left (or right)
    - Parse 3 – 4 + 5 as (3 – 4) + 5

See book for more info

Theoretical Foundations

- Many foundational systems
  - Computability Theory
  - Program Logics
  - Lambda Calculus
  - Denotational Semantics
  - Operational Semantics
  - Type Theory
- Consider two of these methods
  - Lambda calculus (syntax, operational semantics)
  - Denotational semantics

Plan for next 1.5 lectures

- Lambda calculus
- Denotational semantics (shorten this year)
- Functional vs imperative programming

Lambda Calculus

- Formal system with three parts
  - Notation for function expressions
  - Proof system for equations
  - Calculation rules called *reduction*
- Additional topics in lambda calculus
  - Mathematical semantics (=model theory)
  - Type systems

We will look at syntax, equations and reduction

There is more detail in book than we will cover in class

History

- Original intention
  - Formal theory of substitution (for FOL, etc.)
- More successful for computable functions
  - Substitution --> symbolic computation
  - Church/Turing thesis
- Influenced design of Lisp, ML, other languages
- Important part of CS history and theory

Why study this now?

- Basic syntactic notions
  - Free and bound variables
  - Functions
  - Declarations
- Calculation rule
  - Symbolic evaluation useful for discussing programs
  - Used in optimization (in-lining), macro expansion
  - Illustrates some ideas about scope of binding

Expressions and Functions

- Expressions
  $x + y$ $\quad$ $x + 2*y + z$
- Functions
  $\lambda x.\ (x+y)$ $\quad$ $\lambda z.\ (x + 2*y + z)$
- Application
  $(\lambda x.\ (x+y))\ 3 \quad = \ 3 + y$
  $(\lambda z.\ (x + 2*y + z))\ 5 \quad = \ x + 2*y + 5$

Parsing: $\lambda x.\ f\ (f\ x) = \lambda x.\ (\ f\ (f\ (x))\ )$

Higher-Order Functions

- Given function f, return function f ∘ f
  $\lambda f.\ \ \lambda x.\ f\ (f\ x)$
- How does this work?
  $(\lambda f.\ \ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)$
  $= \ \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x)$
  $= \ \lambda x.\ (\lambda y.\ y+1)\ (x+1)$
  $= \ \lambda x.\ (x+1)+1$

Same result if step 2 is altered.

Same procedure, Lisp syntax

- Given function f, return function f ∘ f
  (lambda (f) (lambda (x) (f (f x))))
- How does this work?
  ((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ y 1))

  = (lambda (x) ((lambda (y) (+ y 1))
  ((lambda (y) (+ y 1)) x))))

  = (lambda (x) ((lambda (y) (+ y 1)) (+ x 1))))

  = (lambda (x) (+ (+ x 1) 1))

Declarations as "Syntactic Sugar"

```
function f(x)
    return x+2
end;
f(5);
```

$\underbrace{(\lambda f.\ \ f(5))}_{\text{block body}}\ \ \underbrace{(\lambda x.\ \ x+2)}_{\text{declared function}}$

let $x = e_1$ in $e_2 \quad = \quad (\lambda x.\ \ e_2)\ \ e_1$

Free and Bound Variables

- Bound variable is "placeholder"
  - Variable x is bound in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is same function as $\lambda z.\ (z+y)$
- Compare
  $\int x+y\ dx \ = \ \int z+y\ dz \qquad \forall x\ \ P(x) = \forall z\ \ P(z)$
- Name of free (=unbound) variable does matter
  - Variable y is free in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is *not* same as $\lambda x.\ (x+z)$
- Occurrences
  - y is free and bound in $\lambda x.\ ((\lambda y.\ y+2)\ x) + y$

Reduction

- Basic computation rule is β-reduction
  $(\lambda x.\ e_1)\ e_2 \quad \to \quad [e_2/x]e_1$
  where substitution involves renaming as needed
  (next slide)
- Reduction:
  - Apply basic computation rule to any subexpression
  - Repeat
- Confluence:
  - Final result (if there is one) is uniquely determined

Rename Bound Variables

- Function application

  $\underbrace{(\lambda f.\ \lambda x.\ f\ (f\ x))}_{\text{apply twice}}\ \ \underbrace{(\lambda y.\ y+x)}_{\text{add x to argument}}$

- Substitute "blindly"

  $\lambda x.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ x)] \ = \ \lambda x.\ x+x+x$

- Rename bound variables

  $(\lambda f.\ \lambda z.\ f\ (f\ z))\ \ (\lambda y.\ y+x)$

  $= \ \lambda z.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ z))] \ = \ \lambda z.\ z+x+x$

Easy rule: always rename variables to be distinct

1066 and all that

- *1066 And All That*, Sellar & Yeatman, 1930
  - *1066* is a lovely parody of English history books, "Comprising all the parts you can remember including one hundred and three good things, five bad kings and two genuine dates."
- Battle of Hastings Oct. 14, 1066
  - Battle that ended in the defeat of Harold II of England by William, duke of Normandy, and established the Normans as the rulers of England

Main Points about Lambda Calculus

- λ captures "essence" of variable binding
  - Function parameters
  - Declarations
  - Bound variables can be renamed
- Succinct function expressions
- Simple symbolic evaluator via substitution
- Can be extended with
  - Types
  - Various functions
  - Stores and side-effects

  (But we didn't cover these)

Denotational Semantics

- Describe meaning of programs by specifying the mathematical
  - Function
  - Function on functions
  - Value, such as natural numbers or strings

  defined by each construct

Original Motivation for Topic

- Precision
  - Use mathematics instead of English
- Avoid details of specific machines
  - Aim to capture "pure meaning" apart from implementation details
- Basis for program analysis
  - Justify program proof methods
    - Soundness of type system, control flow analysis
  - Proof of compiler correctness
  - Language comparisons

Why study this in CS 242 ?

- Look at programs in a different way
- Program analysis
  - Initialize before use, …
- Introduce historical debate: functional versus imperative programming
  - Program expressiveness: what does this mean?
  - Theory versus practice: we don't have a good theoretical understanding of programming language "usefulness"

Basic Principle of Denotational Sem.

- Compositionality
  - The meaning of a compound program must be defined from the meanings of its parts (*not* the syntax of its parts).
- Examples
  - P; Q

    composition of two functions, state → state
  - letrec f(x) = e_1 in e_2

    meaning of e_2 where f denotes function ...

Trivial Example: Binary Numbers

- Syntax

  b ::= 0 | 1

  n ::= b | nb

  e ::= n | e+e
- Semantics value function E : exp -> numbers

  E [[0]] = 0 E [[1]] = 1

  E [[nb]] = 2* E[[n]] + E[[b]]

  E [[e_1+e_2]] = E[[e_1]] + E[[e_2]]

  Obvious, but different from compiler evaluation using registers, etc.
  This is a simple machine-independent characterization ...

Second Example: Expressions w/vars

- Syntax

  d ::= 0 | 1 | 2 | ... | 9

  n ::= d | nd

  e ::= x | n | e + e
- Semantics value E : exp x state -> numbers

  state s : vars -> numbers

  E [[x]] s = s(x)

  E [[0]] s = 0 E [[1]] s = 1 ...

  E [[nd]] s = 10* E[[n]] s + E[[d]] s

  E [[e_1 + e_2]] s = E[[e_1]] s + E[[e_2]] s

Semantics of Imperative Programs

- Syntax

  P ::= x:=e | if B then P else P | P;P | while B do P
- Semantics
  - C : Programs → (State → State)
  - State = Variables → Values

    would be locations → values if we wanted to model aliasing

Every imperative program can be translated into a functional program in a relatively simple, syntax-directed way.

Semantics of Assignment

C[[x:= e]]

is a function states → states

C[[x:= e]] s = s'

where s' : variables → values is identical to s except

s'(x) = E [[e]] s gives the value of e in state s

Semantics of Conditional

C[[if B then P else Q]]

is a function states → states

C[[if B then P else Q]] s =

C[[P]] s if E [[B]] s is *true*

C[[Q]] s if E [[B]] s is *false*

Simplification: assume B cannot diverge or have side effects

Semantics of Iteration

C[[while B do P]]

is a function states → states

C[[while B do P]] = the function f such that

f(s) = s if E [[B]] s is *false*

f(s) = f(C[[P]](s)) if E [[B]] s is *true*

Mathematics of denotational semantics: prove that there is such a function and that it is uniquely determined. "Beyond scope of this course."

Perspective

- Denotational semantics
  - Assign mathematical meanings to programs in a structured, principled way
  - Imperative programs define mathematical functions
  - Can write semantics using lambda calculus, extended with operators like

    modify : (state × var × value) → state
- Impact
  - Influential theory
  - Indirect applications via

    abstract interpretation, type theory, ...

Functional vs Imperative Programs

- Denotational semantics shows
  - Every imperative program can be written as a functional program, using a data structure to represent machine states
- This is a theoretical result
  - I guess "theoretical" means "it's really true" (?)
- What are the practical implications?
  - Can we use functional programming languages for practical applications?

    Compilers, graphical user interfaces, network routers, ...

What is a *functional* language ?

- "No side effects"
- OK, we have side effects, but we also have higher-order functions...

We will use *pure functional language* to mean "a language with functions, but without side effects or other imperative features"

No-side-effects language test

Within the scope of specific declarations of $x_1, x_2, \ldots, x_n$, all occurrences of an expression e containing only variables $x_1, x_2, \ldots, x_n$, must have the same value.

- Example

```
begin
    integer x=3; integer y=4;
    5*(x+y)-3
    ...    =?          // no new declaration of x or y //
    4*(x+y)+1
end
```

Example languages

- Pure Lisp

  atom, eq, car, cdr, cons, lambda, define
- Impure Lisp: rplaca, rplacd

  lambda (x) (cons

  (car x)

  (... (rplaca (... x ...) ...) ... (car x) ...)

  ))

  Cannot just evaluate (car x) once
- Common procedural languages are not functional
  - Pascal, C, Ada, C++, Java, Modula, ...

    Example functional programs in a couple of slides

Backus' Turing Award

- John Backus was designer of Fortran, BNF, etc.
- Turing Award in 1977
- Turing Award Lecture
  - Functional prog better than imperative programming
  - Easier to reason about functional programs
  - More efficient due to parallelism
  - Algebraic laws
    - Reason about programs
    - Optimizing compilers

Reasoning about programs

- To prove a program correct,
  - must consider everything a program depends on
- In functional programs,
  - dependence on any data structure is *explicit*
- Therefore,
  - easier to reason about functional programs
- Do you believe this?
  - This thesis must be tested in practice
  - Many who prove properties of programs believe this
  - Not many people really prove their code correct

Functional programming: Example 1

- Devise a representation for stacks and implementations for functions
  - push (elt, stk) returns stack with elt on top of stk
  - top (stk) returns top element of stk
  - pop (stk) returns stk with top element removed
- Solution
  - Represent stack by a list
    - push = cons
    - top = car
    - pop = cdr

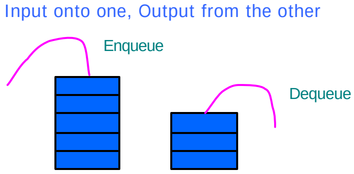
This ignores test for empty stack, but can be added …

Functional programming: Example 2

- Devise a representation for *queues* and implementations for functions
  - enq (elt, q) returns queue with elt at back of q
  - front (q) returns front element of q
  - deq (q) returns q with front element removed
- Solution
  - Can do this with explicit pointer manipulation in C
  - Can we do this efficiently in a functional language?

Functional implementation

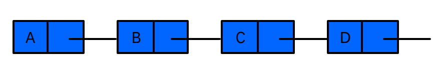
- Represent queue by two stacks
  - Input onto one, Output from the other



  - Flip stack when empty; constant amortized time.
- Simple algorithm
  - Can be proved correct relatively easily

Disadvantages of Functional Prog

Functional programs often less efficient. Why?



Change 3rd element of list x to y

- (cons (car x) (cons (cadr x) (cons y (cdddr x))))
  - Build new cells for first three elements of list
- (rplaca (cddr x) y)
  - Change contents of third cell of list directly

However, many optimizations are possible

Von Neumann bottleneck

- Von Neumann
  - Mathematician responsible for idea of stored program
- Von Neumann Bottleneck
  - Backus' term for limitation in CPU-memory transfer
- Related to sequentiality of imperative languages
  - Code must be executed in specific order

    function f(x) { if x<y then y:=x else x:=y };

    g(f(i), f(j));

Eliminating VN Bottleneck

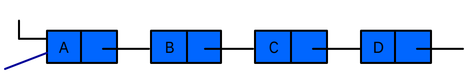
- No side effects
  - Evaluate subexpressions independently
  - Example
    - function f(x) { if x<y then 1 else 2 };
    - g(f(i), f(j), f(k), ...);
- Does this work in practice? Good idea but …
  - Too much parallelism
  - Little help in allocation of processors to processes
  - …
  - David Shaw promised to build the non-Von …
- Effective, easy concurrency is a *hard* problem

Optional extra topic

- Interesting optimizations in functional languages
  - Experience suggests that optimizing functional languages is related to parallelizing code
  - Why? Both involve understanding *interference* between parts of a program
- FP is more efficient than you might think
  - But efficient functional programming involves complicated operational reasoning

Sample Optimization: Update in Place

Function uses updated list



(lambda (x)

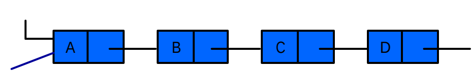
  (... (list-update x 3 y) ... (cons 'E (cdr x)) ...)

Can we implement list-update as assignment to cell?

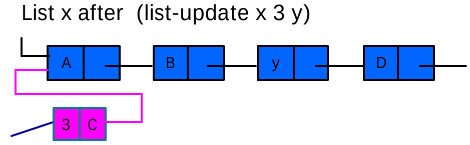
May not improve efficiency if there are multiple pointers to list, but should help if there is only one.

Sample Optimization: Update in Place

Initial list x



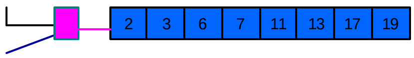
List x after (list-update x 3 y)



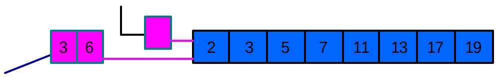
This works better for arrays than lists.

Sample Optimization: Update in Place

Array A



Update(A, 3, 5)



- Approximates efficiency of imperative languages
- Preserves functional semantics (old value persists)