Paul Graham: "Beating the Averages"

Website

- **Mailing list**
- **Reading assignments**

Syntax

- **How to get from the textual representation of a program to a logical representation?**
  - Example: Piece of code with a loop
    - Textually, it's just a sequence of words and symbols
    - Logically, there is a tree structure: Some code belongs inside the loop
  - Usually divided into two phases (not strictly necessary):
    - Lexical analysis: identify tokens (words, numbers, symbols, etc.)
    - Parsing: construct tree from sequence of tokens
  - Text -> Tokens by lexical analysis, using lexical rules)
  - Tokens -> Parse tree by parsing, using syntax rules (a grammar)
- **In this class we're not much concerned with algorithms for parsing**
- **But we are concerned with lexical rules and grammar, for together they descibe:**
  - The textual representations of programs in a language
  - The logical representations of programs in a language
  - The set of valid programs in a language
    - Only to some extent
    - Most languages have additional rules
- **Lexical analysis**
  - Example epsilon program (see slide)
  - Example
    - Source code: n2 := 100
    - Result: The following tokens (lexemes):
      - IDENT("n2")
      - ASSIGN
      - VALUE("100")
  - Tasks of the lexical analyzer ("lexer"):
    - Remove whitespace (space, newline, tab)
    - Remove comments
    - Recognize and assemble symbols, classify as tokens
      - keywords (begin, end, program, procedure, etc.)
      - operators (:=, etc.), parantheses
      - constants (numbers)
      - identifiers (variable names, etc.)
      - compute line numbers for error messages
      - give error messages, e.g., invalid characters (e.g., '?'), unterminated comments, lines that are too long
    - Techniques

        - Specify tokens by regular expressions
        - Recognize by a deterministic finite automaton (DFA, see CS136)
    - Some lexical rules for epsilon
        - if return(IF)
        - := return(ASSIGN)
        - [a-zA-Z][a-zA-Z0-9]* return(IDENT(text))
- **Parsing (syntactic analysis)**
    - Show epsilon program and its parse tree (see slide)
    - Parsing used to be difficult (Fortran I)
    - Now: Rule-driven parser generators (CS160)
        - Best known (for C): YACC: Yet Another Compiler Compiler
    - Grammar
        - Fortran: English text
        - ALGOL 60: Context-free grammar developed by John Backus
            - Backus-Neur Form (BNF)
        - EBNF: Extended BNF
    - Partial grammar for epsilon (see slide)
        - terminals, non-terminals

Semantics

- **Purpose**
    - Further define the set of valid programs
        - Not all syntactically correct programs have meaning. Examples:
            - Cannot use an undeclared variable
            - Type checking
    - Define what the program is supposed to do
- **Natural language (English text)**
    - Example: C, Common Lisp
    - Imprecise
    - "Language lawyers"
    - Hard to describe what should be done without describing how
- **Reference implementation**
    - Perl, Python
    - Hard to make an alternative implementation
        - Someone is bound to rely on even the accidental behavior of the implementation
        - Has to be "bug-compatible" with the reference implementation
    - Hard to modify the reference implementation
- **Operational semantics**
    - Describe what the program construct should do in terms of a machine
        - Machine can be virtual or real
        - Virtual makes it independent of the ideosyncracies of a real machine

- Attempt to abstract away from the implementation details of a reference implementation
- Efficiency is not a concern
- Formal operational semantics: Vienna Definition Language (VDL)
  - Used to described the semantics of PL/I (1972)
  - The VDL description is so complex it serves no practical purpose

- **Axiomatic semantics**
  - View program as a state machine. Constructs modify the state.
  - Describe state by first-order logic predicate (see CS40)
  - The meaning of each construct is defined as a function "asem" that relates the state before the construct to the state after the construct
    - Precondition: State before the construct
    - Postcondition: State after the construct
  - Example: Assignment
    - Construct: x := y + 1
    - Postcondition: x > 0
    - What is a valid precondition?
      - y = 3
      - y = 42
    - What is the weakest precondition?
      - y ≥ 0
      - Weakest because it is implied by all the other conditions
        - If y = 42 then y ≥ 0
  - The function asem is called a predicate transformer.
    - Computes the weakest precondition for any statement and postcondition
  - Can we find the function asem for any simple assignment?
    - Simple assignment: VAR := EXPRESSION
    - Replace VAR in the postcondition with EXPRESSION
      - x > 0
      - y + 1 > 0
      - y > -1
      - y ≥ 0
  - Composition
    - Statements S1 and S2
    - Compound statement C: "S1; S2" (first do S1, then do S2)
    - How can we find the weakest precondition of C?
      - Let P be the postcondition of C
      - Use asem to compute R, the precondition of S2
      - Now, let R be the postcondition of S1
      - Use asem to compute W, the weakest precondition of S1
    - Example: two assignments
      - S1: y = x * x

        - S2: z = y + 4
        - C: y = x * x; z = y + 4
        - P: z > 8
        - R: y + 4 > 8 => y > 4
        - W: x * x > 4 => x > 2
    - Loops are more complex
        - while B do L
        - What if we know how many times L will execute?
            - Same as composition
        - But in general, we don't know how many times L will execute
            - Don't know that it will terminate at all
            - If we did, we could solve the halting problem (see CS186)
        - If we cannot find the weakest precondition, can we find another precondition that is not the weakes?
            - Come up with a loop invariant
                - Something that is true before and after each loop iteration.
            - Example: power function (see slide)
                - Postcondition: p = a^n
                - What about after the next-to-last iteration?
                    - p = a^(n - 1)
                - Is there something that is n after n iterations and n - 1 after n - 1 iterations?
                    - i
                - So after each loop iteration, p = a^i
                - That is our loop invariant
                - It is also our precondition for the loop
    - Finding asem for a specific loop in a specific program equires some creativity. (It helps to keep it in mind when you write the code.)
        - Cannot specify an asem function that works for any loop written in the language.
        - Axiomatic semantics have been useful for proving specific programs correct (see CS266)
            - ASLAN, a system for formally specifying programs (Kemmerer)
            - A formally verified Unix kernel (Kemmerer)
        - Not as good for defining a programming language
- **Denotational semantics**
    - Also considers the program as a state machine
    - Function dsem gives the value of each variable
        - M(X): value of variable X in state M (the book uses mem(x))
        - Different from axiomatic semantics, where the states are described by predicates
    - Example: assignment
        - b = a+ 1

    - Let M be the state before and M' the state after
    - M' defined in terms of the function dsem:
        - M' = dsem(b = a +1, M)
    - What is M'?
        - M'(b) = M(a + 1)
        - M'(X) = M(X) for all X ≠ b
    - What is the function dsem for all simple assignments?
        - Let Y = EXPR be the assignment
        - dsem(Y = EXPR, M) = M' where
            - M'(Y) = M(EXPR)
            - M'(X) = M(X) for all X ≠ Y
- Loops
    - while B do L
    - Example: power (see slide)
        - while i < n begin i := i + 1; p := p * a; end
        - dsem(begin i := i + 1; p := p * a; end, M) defined by
            - M'(i) = M(i) + 1
            - M'(p) = M(p) * a
            - For brevity, call this dsem(L, M)
        - dsem(while i < n L, M) defined by
            - M if i ≥ n
            - dsem(while i < n L, dsem(L, M)) if i < n
- This describes what the effect of the loop should be on the state without actually prescribing a certain way to do it
    - Compiler is free, for instance, to reorder the assignments, or to optimize them away entirely