Syntax and Parsing

- ***Syntax***: the form of a program
- ***Semantics***: the meaning of a program
- **Two parts to syntax analysis:**
  - → **lexical rules: define legal characters and how they can be combined to form symbols**

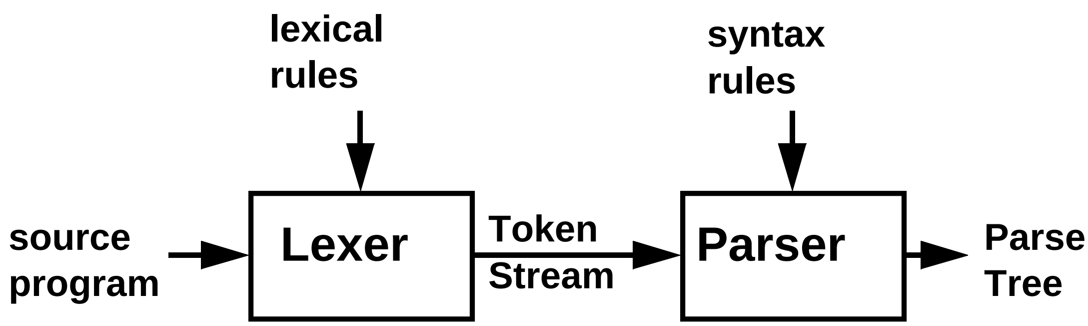
    ("lexemes").
  - → **syntax rules: define how categories of lexemes ("tokens") can be combined to form legal programs.**

Syntax and Parsing

- **More formal definition of Syntax:**
  - → **Syntax defines the symbols of a language, and the spatial relationships between symbols in that language**

- **Semantics**
  - → **Semantics decribes the meanings to be ascribed to symbols, and structures of symbols, in a language.**

Compiler Front End



Syntax Analysis

- **We won't make a strict distinction, but will generally deal with syntax rules.**

How to Describe the Syntax of a Language?

- **English description**
  - → **lengthy, tedious, ambiguous**
- **Formal description**
  - → **recognizer: given a string, a recognizer for a language L tells whether or not the string is in L (ex: Compiler)**
  - → **generator: a generator for L will produce an arbitrary string in L on demand. (ex: Grammar)**
- **Recognition and generation are useful for different things, but are <u>closely</u> related.**
- **First, we'll talk about an important generation tool: BNF.**

BNF - A Generative Tool

- **Backus-Naur Form (BNF) is a *metalanguage* for describing the syntax of programming languages.**
  - → **developed by John Backus and Peter Naur**
  - → **first used to describe ALGOL60**
- **A language description in BNF is called a *grammar*.**

Note: In formal linguistics a grammar is described as the syntax of a language together with an knowledge of the allowable constructs that provide “sensible” meanings. Obviously the syntax of a programming language does not contain “knowledge”, but we persist in this misuse of the term “grammar”.

Grammars

- **A grammar is made up of productions, or rules, e.g.,**

  <sentence> --> <subject><verb><object>

  <verb> --> see | love

  <subject> --> I

  <object> --> him | her

- **Five components:**

  The rules using:

  --> : "is defined as"

  | : "or"

  implied comcatenation

  terminals: see, love, I, him, her.

  non-terminals: <sentence>, <verb>, <subject>, <object>

  root symbol: <sentence>

  the algorithmsmof generation or analysis

Note: Except where specifically required, spaces are assumed to be inserted between “lexemes” in natural languages.

Recursion

- **Need recursion to define strings of indefinite length:**

<ones> --> 1 | 1<ones>

==> 1, 11, 111, 1111, . . .

<ablist> --> ab | a<ablist>b

==> ab

OR   a<ablist>b

==> aabb

OR   aa<ablist>bb

==> aaabbb

OR   aaa<ablist>bbb

. . .

Describe the language resulting from this syntax!

Write a syntax that describes a language composed of strings of arbitrary numbers of “a”s followed by arbitrary numbers of “b”s

Derivations

- **The steps in generating a string using a grammar are called a *derivation*.**

<exp> --> <id> | <exp> + <exp> | <exp> * <exp> | (<exp>)

<id> --> A | B | C

Note: " --> "
" ==> "

<exp> ==> <exp> + <exp>

==> <exp> * <exp> + <exp>

==> <id> * <exp> + <exp>

==> A * <exp> + <exp>

==> A * <id> + <exp>

==> A * B + <exp>

==> A * B + <id>

==> A * B + C

- **Each step produces a *sentential form*. It is a string of terminals and nonterminals.**

- **A *sentence* is a sentential form containing only terminals.**

Language

- **We are now ready to define a LANGUAGE:**
  - → **A language is a set of strings of terminals.**
- **Note**
  - → **A language may be finite or infinite. The empty set is a language.**
- **So now define the LANGUAGE GENERATED BY A GRAMMAR:**
  - → **The language generated by a grammar is the set of sentences derivable using the grammar.**

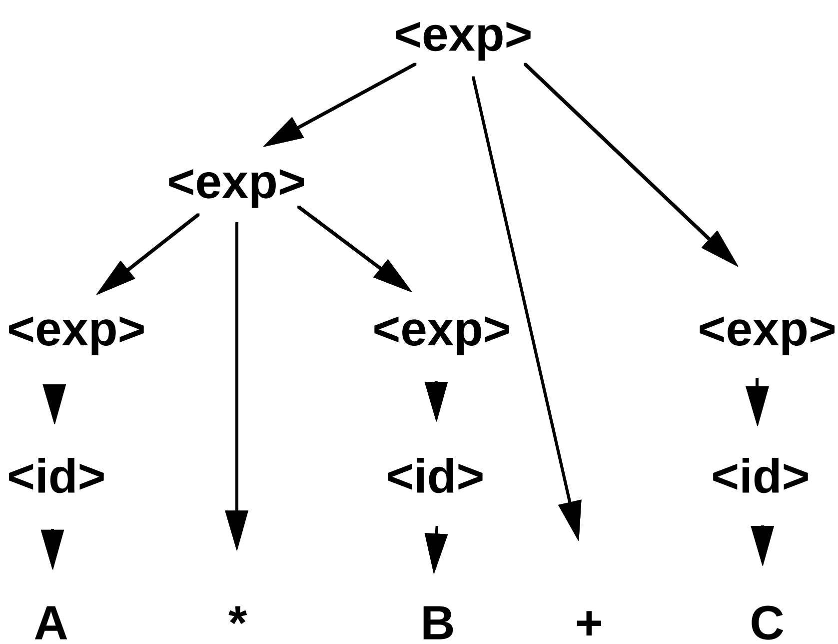
Left-Most and Right-Most Derivations

- **How to choose which non-terminal to replace next:**
  - → **left-most derivation: replace left-most NT first**
  - → **right-most derivation: replace right-most NT first**
- **Need not be either of these**
  - → **random replacement OK**

    can't affect **language** generated

Parse Trees

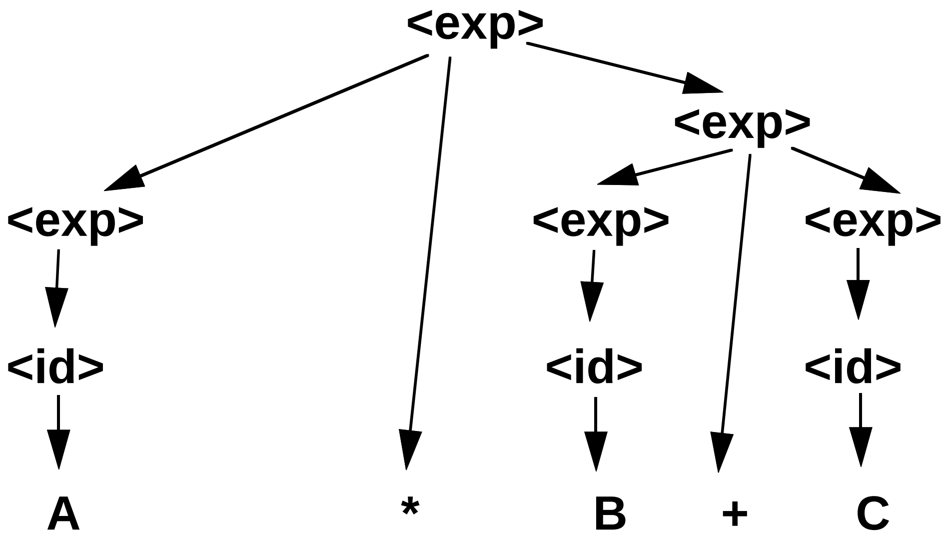
- **Show the <u>syntactic</u> structure of sentences.**



Ambiguous Grammars

- **A grammar is ambiguous if it generates the same sentence for which there are two or more parse trees.**

- **Another parse tree for A * B + C:**



Disambiguating the Grammar

- **To disambiguate this grammar, change to:**

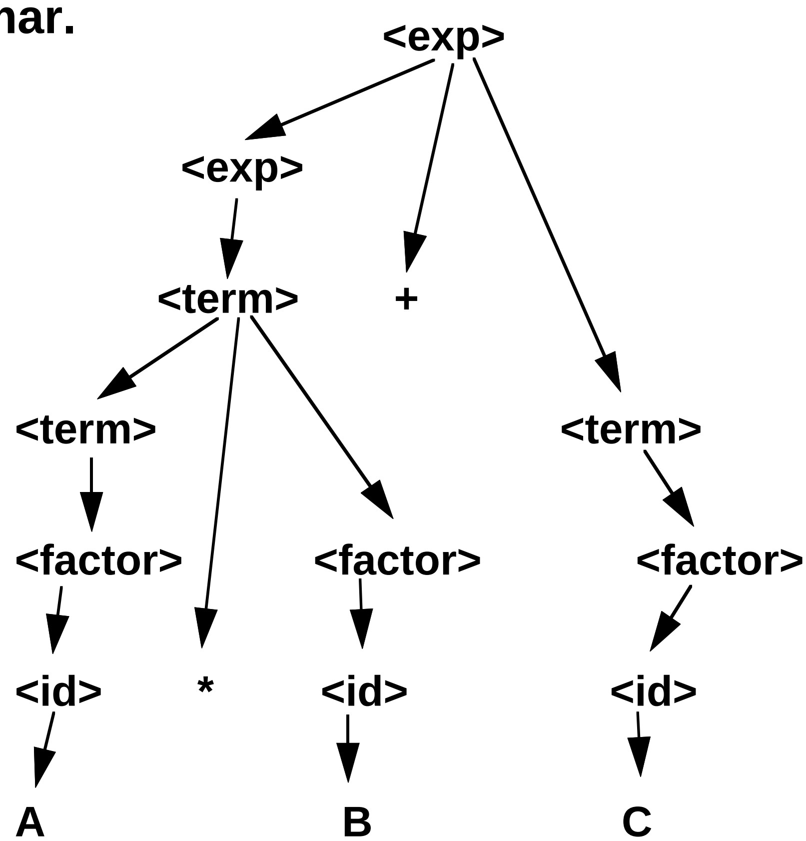
  <exp> --> <exp> + <term> | <term>

  <term> --> <term> * <factor> | <factor>

  <factor> --> (<exp>) | <id>

  <id> --> A | B | C

- **This forces * to have a higher precedence than +, even though it generates the same language as the first grammar.**



Chomsky Hierarchy of Syntax

- **0: Unrestricted grammar rules**
  - **string $\rightarrow$ string**
- **1: Context Sensitive**
  - **ABC $\rightarrow$ AγC**
- **2: Context Free**
  - **A $\rightarrow$ string**
- **3: Linear**
  - **A $\rightarrow$ aB | b**

where
N - **set of non-terminals**
T - **set of terminals**
A,B,C,γ ε N
a,b ε T
string ε (N U T)*

Chomsky Hierarchy of Syntax

- **0: Unrestricted grammar**
  - **string $\rightarrow$ string (arbitrary word processing change)**
- **1: Context Sensitive**
  - **ABC $\rightarrow$ AγC (change B to γ in the context of A..C)**
- **2: Context Free $\rightarrow$ syntactic analyzer**
  - **A $\rightarrow$ string (change A to string anywhere)**
- **3: Linear $\rightarrow$ lexical analyzer**
  - **A $\rightarrow$ aB | b (there is a one-for-one replacement in N)**

Chomsky Hierarchy of Syntax

$$G_0 \supset G_1 \supset G_2 \supset G_3$$

G_3 - linear, one-dimensional

G_2 - two-dimensional

G_1 - three-dimensional

Another Ambiguous Grammar

<stmt> --> <assign> | <if_stmt>

<assign> --> <id> := <exp>

<if_stmt> --> IF <bool> THEN <stmt> |

IF <bool> THEN <stmt> ELSE <stmt>

- **Exercise:**
  - → **Prove that this is ambiguous.**
  - → **Write a grammar for the <u>same</u> language that is not ambiguous. (Use any published reference material)**

Note: This grammar (syntax) develops a language that demonstrates the “DANGLING ELSE” problem that occurred in ALGOL 60.

Limitations of Context Free Grammars

- **Productions must always apply, regardless of context in which string appears.**

- **Can't handle some things:**

```
var x : integer;
    y : boolean;
begin
    x := 3;          OK
    x := y;          types wrong
    z := 5;          var z undeclared
```

- **Need "static semantics" . . .**

Recognizers

- **How to generate a recognizer from a grammar?**
  - **automatically (YACC)**
  - **by hand**
- **There are many types of parsers:**
  - **LL(0)**
  - **LL(k)**
  - **LR**
  - **LALR**
  - **recursive descent**

Extended BNF & Regular Expressions

Bracket notations

- **[. . .]   optional**

  <if> --> IF <bool> THEN <stmt> [ELSE <stmt>]

- **{. . .}   zero or more times**

  <ones> --> 1 {<ones>}

- **(. . . | . . .) or (: : :) local choice**

  <exp> --> <id> | <exp> (+ | *) <exp>

Regular Expressions

- **a^* equivalent to {a} - zero or more times**
- **a^+ equivalent to a{a} - one or more times**

Syntax and Parsing Summary

- **Recognizer vs. generator**
- **BNF**
  - → **Four components**
  - → **Recursion**
- **Derivation**
- **Ambiguous grammars**
- **Extended BNF**

Semantics of Programming Languages

- **How to define the meaning of programs?**
- **Three approaches:**
  - → **Operational**
  - → **Axiomatic**
  - → **Denotational**

Operational Semantics

- **Gives a program's meaning in terms of its implementation on a real or virtual machine**

- **Define two parts:**

- **machine**
  - → **high level**
  - → **low level**

- **translation from source code to "machine" code**

- **Let S be the *state,* having components mem, input, and output --- where mem consists of register values, memory values, etc.**

S { instruction } S'

Diff (S, S') => "meaning of instruction"

Example

Pascal	Operational Semantics

for i := x to y do
begin
.
.
.
end
	i := x
	loop: if i>y goto out
	.
	.
	.
	i := i + 1
	goto loop
	out: . . .

- **Operational semantics could be much lower level, e.g.,**

```
        mov i,r1
        mov y,r2
        jmpifless(r2,r1,out)
        ...
out:    ...
```

Advantages and Disadvantages of Operational Semantics

- **Advantages:**
  - → **May be simple, intuitive for small examples**
  - → **Useful for implementation**
- **Disadvantages**
  - → **Very complex for large programs**
  - → **Lacks mathematical rigor**
- **Uses:**
  - → **Vienna Definition Language (VDL) used to define PL/I (Wegner 1972)**
  - → **Compiler work**

Axiomatic Semantics

- **Based on predicate calculus. Use assertions to show certain properties of programs.**

  {P} statement {Q}

- **Compute precondition from postcondition:**

  {P} $x := y + 1$ {x>0}

  → **Possible Ps:**

  $y > 5$

  $y = 37$ **Weakest Precondition (WP)**

  $y \geq 0$

  etc.

- ***WP* -> identifies *all possible* cases for which postcondition holds!**

  WP => Least Restrictive

Finding the Weakest Precondition

- **Define function:**

  wp: Stmt x Postcondition --> weakest precondition

  stmt   post condition

  wp (x := e, P) = P $_{x \text{ -> } e}$ "substitute e for every x in P"

- **So:**

  wp (x := y+1, x > 0)

  = x > 0 $_{x \text{ -> } y+1}$

  = y+1 > 0

  = y $\geq$ 0

  → **basically, “undoing” the assignment and solving for y**

  y > -1 is WP, too

  y > -1 iy >= 0 when y is an integer

Sequences of Statements

{P} S1; S2 {Q}

- **Just apply wp twice**

wp (x := y + 1; z := x + y, z > 5)

wp (z := x + y, P1)

$= z > 5_{z \to x + y}$

$= x + y > 5$

What can we say about x+y knowing that after the assignment is made z > 5?

wp (x := y + 1, x + y > 5)

$= x + y > 5_{x \to y + 1}$

$= y + 1 + y > 5$

$= y > 2$

Loops

- **{P} while B do S end {Q}**
- **Need *loop invariant* I such that:**
  - **P ==> I**
  - **{I} B {I}**
  - **{I & B} S {I}**
  - **(I & (not B)) ==> Q**
  - **and the loop terminates**

Finding Loop Invariants

- **Work backwards through a few iterations and look for a pattern.**

  while y <> x do y:= y+1   {y = x}

  wp (y := y + 1, {y = x}) = {y = x} $_{y \to y+1}$

  = y = x - 1 -- last time through

  wp (y := y + 1, {y = x - 1}) = {y = x-1} $_{y \to y+1}$

  = y = x - 2 -- next to last time

  Now, by extension we get I = { y < x }

  When we factor in that the loop may not be executed even once (implying wp could be y = x) then we get

  I = { y ≤ x }

- **This also satisfies loop termination, so**

  P = I = {y ≤ x}

Finding Loop Invariants (cont.)

- **Does {y <= x} satisfy conditions (1) - (5)?**

(1) y = x ==> y <= x ?

(2) if y <= x and y <> x is then evaluated, is it still the case that y <= x ?

(3) if y <= x and y <> x are both true and then y := y+1 is executed, then is y <= x ?

(4) does y <= x and y = x ==> y = x ?

(5) Can you argue convincingly that the program segment terminates?

Finding Loop Invariants (a harder example)

- **However, finding the WP is not always so easy! Consider:**

  {P} while y < x + 1 do y := y + 1 {y>5}

  $y>5_{\ y \to y+1} \quad \Rightarrow y>4$

  $y>4_{\ y \to y+1} \quad \Rightarrow y>3$

  etc.

- **really tells us *nothing* relative to x because x is not in Q ≡ {y > 5}**

- **Try Using Boolean**

  I & (not B) => Q

  ? & $y \geq x + 1$ => $y > 5$

  ? & $y > x$ => $y > 5$

  any $x \geq 5$ satisfies implication

  so . . . let $I \equiv x \geq 5$

- **Do the 4 Axioms hold?**

Advantages, Disadvantages, and Uses of Axiomatic Semantics

- **Advantages**
  - → **Can be very abstract**
  - → **May be useful in proofs of correctness**
  - → **Solid theoretical foundations**
- **Disadvantages**
  - → **Predicate transformers are hard to define**
  - → **Hard to give complete meaning**
  - → **Does not suggest implementation**
- **Uses of Axiomatic Semantics**
  - → **Semantics of Pascal**
  - → **Reasoning about correctness**

HOMEWORK FOR AXIOMATIC SEMANTICS

Consider

$$\{P\}\ \ X := X * 3 \quad \{X^2 = 36\}$$

Determine <u>Weakest Precondition</u> P

Denotational Semantics

- **Define a function that maps a program (a syntactic object) to its meaning (a semantic object).**
- **Sort of like a high-level operational semantics, except**
  - → **machine is gone**
  - → **language is λ-calculus**
- **More abstract.**

Example: Decimal Numbers

Valuation function: V: Number --> Integers

syntax meaning

- **Syntax:**

  <num> --> <num> <digit> **|** <digit>

  <digit> --> 0 **|** 1 **|** 2 **|** . . . **|** 9

- **Semantics:**

  → **Let n** $\in$ **<num>, d** $\in$ **<digit>**

Applying the Function **V** to Syntax

$V\llbracket nd \rrbracket = 10 * V\llbracket n \rrbracket + V\llbracket d \rrbracket$

$V\llbracket 0 \rrbracket = 0$

$V\llbracket 1 \rrbracket = 1$

integers as we know them

- **Consider** $\mathbf{V\llbracket 237 \rrbracket}$ **:**

$V\llbracket 237 \rrbracket = 10 * V\llbracket 23 \rrbracket + V\llbracket 7 \rrbracket$

$= 10 * (10 * V\llbracket 2 \rrbracket + V\llbracket 3 \rrbracket) + V\llbracket 7 \rrbracket$

$= 10 * (10 * 2 + 3) + 7$

$= 10 * (20 + 3) + 7$

$= 10 * (23) + 7$

$= 230 + 7$

$= 237$

Expressions

- **But for *real* programming languages we need more info:**

  E: Expression --> Integer

  $E\llbracket x \rrbracket = ?$ where x is a variable

- **Depends on the current *state***

  → **STATE = <mem, input, output>**

  mem: Identifier --> Integer

  input: Integer *

  output: Integer *

  Streams of integers

- **Now**

  E: Expression x STATE --> Integer

  $E(\llbracket x \rrbracket, s) = mem(\llbracket x \rrbracket)$ where s = <mem,i,o>

  $E(\llbracket e_1 + e_2 \rrbracket, s) = E(\llbracket e_1 \rrbracket, s) + E(\llbracket e_2 \rrbracket, s)$

Statements

- **Expressions denote a value, but statements denote a state.**

ST: (Stmt x STATE) --> STATE

ST ($[\![$x := e$]\!]$, s) = <mem',i,o> where

s = <mem,i,o>

mem'$[\![$x$]\!]$ = E($[\![$e$]\!]$,s)

mem'$[\![$y$]\!]$ = mem$[\![$y$]\!]$ for all y $\neq$ x

ST ($[\![$write(e)$]\!]$, s) = <mem,i,o'> where

s = <mem,i,o>

o' = o • (E ($[\![$e$]\!]$, s))

Sequences of Statements

- **Basic (sequential statement evaluation)**

  $ST(\llbracket stmt_1; stmt_2 \rrbracket, s) =$

  $ST(\llbracket stmt_2 \rrbracket, s')$ where

  $s' = ST(\llbracket stmt_1 \rrbracket, s)$

- **Parallel statement evaluation**

  $ST(\llbracket stmt_1; stmt_2;stmt_3 \rrbracket, s) = \{s_1, s_2, s_3\}$ where

  $s_1 = ST(\llbracket stmt_1;stmt_2;stmt_3 \rrbracket, s)$

  $s_2 = ST(\llbracket stmt_1;stmt_3;stmt_2 \rrbracket, s)$

  $s_3 = ST(\llbracket stmt_3;stmt_1;stmt_2 \rrbracket, s)$

Example

```
P:  {          x := 5;
    {   P': {  y := x + 1;
    {       {  write(x * y);   }  P"
```

- **Initial state s = <mem,i,o>**

$$ST([\![P]\!], s) = ST([\![P']\!], \underbrace{(ST([\![x := 5]\!], s))}_{s'})$$

s' = <mem',i',o'> where

$mem'([\![x]\!]) = 5$

$mem'([\![z]\!]) = mem([\![z]\!])$ for all $z \neq x$

$i' = i$, $o' = o$

Example (continued)

- **$ST(\llbracket P' \rrbracket, s') = ST(\llbracket P'' \rrbracket, \underbrace{(ST(\llbracket y := x + 1 \rrbracket, s'))}_{s''})$**

  $s'' = \langle mem'', i'', o'' \rangle$ where

  $mem''(\llbracket y \rrbracket) = E(\llbracket x + 1 \rrbracket, s') = 6$

  $mem''(\llbracket z \rrbracket) = mem'(\llbracket z \rrbracket)$ for all $z \neq y$

  $i'' = i', o'' = o'$

- **$ST(\llbracket P'' \rrbracket, s'') = ST(\llbracket \text{write } (x * y) \rrbracket, s'') = s'''$**

  $s''' = \langle mem''', i''', o''' \rangle$ where

  $mem''' = mem'', i''' = i''$

  $o''' = o'' \bullet E(\llbracket x * y \rrbracket, s'') = o'' \bullet 30$

- **So,**

  $ST(\llbracket P \rrbracket, s) = \langle mem''', i''', o''' \rangle$ where

  $mem'''(\llbracket y \rrbracket) = 6$

  $mem'''(\llbracket x \rrbracket) = 5$

  $mem'''(\llbracket z \rrbracket) = mem(\llbracket z \rrbracket)$ for all $z \neq x,y$

  $i''' = i$

  $o''' = o \bullet 30$

Advantages, Disadvantages, and Uses of Denotational Semantics

- **Advantages (of denotational semantics)**
  - → **compact and precise**
  - → **may help with implementation**
  - → **solid mathematical foundations**
- **Disadvantages**
  - → **Hard for programmer to use**
- **Uses**
  - → **Semantics for Algol-60, Pascal, etc.**
  - → **Compiler generation and optimization**

HOMEWORK FOR DENOTATIONAL SEMANTICS

Prefatory Consideration:

Prog. Langs. have conditional statements, e.g.

1) if b then stmt1, else stmt2

2) if b then exp1, else exp2

Assuming that conditionals only support expression evaluation and have no side effects, let's give meaning to 1) above:

ST(⟦ if b then stmt, else stmt2⟧ , s) =

if E(⟦b⟧ , s) then

ST(⟦stmt1⟧ , s) else

ST(⟦stmt2⟧ , s).

Note: use of previous defns T/F Assessment like in 'C' introduction of"IF THEN ELSE" in denotational language

UNDERSTAND/STUDY

1) ST (⟦ if b then stmt1 else stmt2⟧ , s)

definition and elaboration

2) Give denotational semantics for repeat until stmt

<u>REPEAT stmt UNTIL b</u>

You will need conditional statement like that specified above

HINT:

on RHS you might find <u>recursive defn</u>

Summary

- **Each form of semantic description has its place:**
  - → **Operational**

    informal descriptions

    compiler work
  - → **Axiomatic**

    reasoning about particular properties

    proofs of correctness

- **Denotational**

  formal definitions

  provably correct implementations