SYNTAX

what is a syntax and how does it differ, from say, semantics? what are the different ways we can define a syntax? what is the difference between concrete and abstract syntax? and why do programming languages tend to use context-free grammars for their syntax?

DEFINITION

the syntax of a programming language is the set of rules that define which arrangements of symbols comprise structurally legal programs.

in the case of traditional textual languages, "arrangements of symbols" can be read as "strings of symbols".

STRUCTURE VS. MEANING

we say structurally legal because we want to distinguish between (1) errors of well formedness, and (2) errors of meaning.

for example, this java program:

```
class A { 2 / == }{;
```

is not well-formed, it does not have the structure of a java program. a compiler would report a syntax error. but the program:

```
class A { int x = y; }
```

is structurally well formed. according to the official syntax of the java language, this program is fine. the compilation unit consists of a type declaration that is a class declaration whose body consists of a field declaration with a type, a name, and an initializing expression which is an identifier. a compiler will not detect a syntax error; it will, however, detect a semantic error since the identifier y has not been declared.

the same goes for this C program:

```
/*
 * this c program has no syntax errors, but it does
 * have static semantic errors
 */
int main () {
    f(x) -> p = sqrt(3.3);
    return "dog";
}
```

non-structural errors detectable by a compiler are called static semantic errors. they are almost always related to problems of context: identifiers used but not declared,

identifiers redeclared, the wrong no. of arguments passed to a subroutine, assignment to a read-only variable, etc.
in practice, language designers almost always restrict syntax to aspects that are context-free.

DESIGNING THE SYNTAX OF A LANGUAGE

a designer will often start by writing down examples of both legal and illegal strings, then try to figure out the general rules from these examples

example: consider the language of integer arithmetic expressions with parentheses. lets think up examples and non-examples.

examples of strings in this language	non-examples
432	432)
24*(31/899+3-0)/(54/2+4+2*3)	24*(31////)/(5+---+))
(2)	[{we] 23re31124ef r$#%^@
8*(((3-6)))	--2--

from these examples, we infer the legal symbols, the different kinds of structural components (digits, numbers, expressions), and the rules that characterize all and only the legal examples. there are many ways to define the rules formally. here are a few.

CONTEXT-FREE GRAMMERS

formally, a context-free grammar, or CFG, is a quadruple (T, C, P, S) where

- T is a set of tokens
- C is a set of categories, such that $C \cap T = \phi$
- $P \subseteq C \times (C \cup T)^*$ is a set of production rules.
- $S \in C$ is the start symbol.

the language is defined by a grammar (T, C, P, S) is $\{ s \in T^* \mid S \Rightarrow^*_P s \}$

a CFG for the example language is:

```
(
  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (, )},
  {E, N, D},
  {
    (D,0), (D,1), (D,2), (D,3), (D,4), (D,5),
    (D,6), (D,7), (D,8), (D,9), (N,D), (N,ND),
    (E,N), (E, E+E), (E, E-E), (E, E*E),
    (E, E/E), (E, (E))
  },
  E
)
```

here
E stands for expression,
N for a number, and
D for a digit.
also, notice parantheses appear in both the little language we are describing (the object language) and the mathematical notation for the grammar (the meta language). these have to be kept distinct (using colors, fonts, or plain common sense).

we usually take C, T, S from context (no pun intended) and write the production rules (P) in a more readable form:

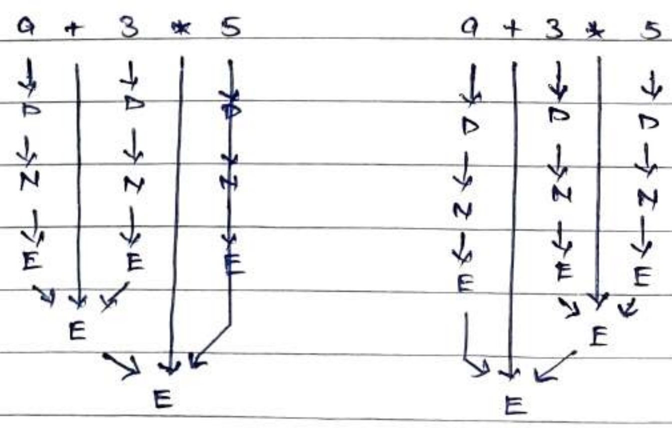
$E \rightarrow N$
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow E / E$
$E \rightarrow (E)$
$N \rightarrow D$
$N \rightarrow ND$
$D \rightarrow 0$
$D \rightarrow 1$
$D \rightarrow 2$
$D \rightarrow 3$
$D \rightarrow 4$
$D \rightarrow 5$
$D \rightarrow 6$
$D \rightarrow 7$ $D \rightarrow 8$ $D \rightarrow 9$

ex describe how C, T & S are inferred from context in the shorthand grammar above.

a little allowable notational shorthand allows us to combine rules with the same left hand sides (but of course be careful if your object language has a pipe in it!):

E → N | E+E | E-E | E*E | E/E | (E)
N → D | ND
D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

a CFG is ambiguous whenever there are multiple parse trees for atleast one string in the language. this is best seen by example. our grammar above is ambiguous because there are 2 ways to parse 9+3*5:



these 2 parse trees suggest different interpretations of the string. fortunately, there are usually many grammars that can be written for a single language. here is another CFG for our language, which is non-ambiguous.

$E \rightarrow T \mid E + T \mid E - T$

$T \rightarrow F \mid T * F \mid T / F$

$F \rightarrow N \mid (E)$

$N \rightarrow D \mid ND$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

ex generate possible parse trees for 9+3*5. convince yourself that the grammar is indeed non-ambiguous. if you feel up to it, prove the grammar is non-ambiguous.

BNF

BNF (backus-naur form) is a style of writing CFGs to make them look a little prettier. the original BNF looked something like this:

```
<expression> ::= <number>
               | <expression> + <expression>
               | <expression> - <expression>
               | <expression> * <expression>
               | <expression> / <expression>
               | ( <expression> )
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

but eventually the concern about having to deal with the possibility of languages that actually contain characters like <, >, | and so on made people realize that one should quote the tokens, not the categories, so modern BNF looks like:

```
expression ::= number
             | expression '+' expression
             | expression '-' expression
             | expression '*' expression
             | expression '/' expression
             | '(' expression ')'
number ::= digit | number digit
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

EBNF

EBNF (extended BNF) adds even more syntactic sugar. there are lots and lots of variants of EBNF, but generally the idea is that EBNF quotes tokens rather than categories and uses fancy markup on the right hand sides, such as:

- parentheses for grouping
- * to indicate zero-or-more (but wirth like curly braces)
- + to indicate one-or-more
- ? to indicate zero-or-one, aka "optional": (but wirth like square brackets)

the non-ambiguous example grammer from above looks like this in EBNF:

EXP -> TERM (('+' | '-') TERM)*
TERM -> FACTOR (('*' | '/') FACTOR)*
FACTOR -> NUMBER | '(' EXP ')'
NUMBER -> ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')+

you might even see

- min and max values in repetition.
  e.g. (A){2,5} for AA | AAA | AAAA | AAAAA

- other forms of selection
  e.g. A||B for (A|B|AB|BA), that is, one or more of A or B must occur, but in any order.

- a binary '-' operator (but watch out for paradoxes, e.g. A-> 'a' - A)

ex explain why the rule A -> 'a' - A leads to a paradox

- something for separators, like the commas in argument lists
  e.g. (A ^ B) for (A (BA)*

Some times you'll even see:

- a notation for a sequence of characters
  e.g. 'while' for 'w' 'h' 'i' 'l' 'e'

- an alternate quoting mechanism meaning exactly one character from this set
  e.g. [aei-kou] means ('a' | 'e' | 'i' | 'j' | 'k' | 'o' | 'u')

to express the characters '-' or ']' in this notation they need to be escaped somehow. we may as well use a backslash, so the backslash itself will need to be escaped.
e.g. [2\]\-p\\] means ('2' | ']' | '-' | 'p' | '\').
but if the '-' appears first or last, we don't need to escape it!

- a notation for representing a character given its codepoint
  e.g. \x followed by 2 hex digits, \u followed by 4 hex digits, and \U followed by 8 hex digits, is as good as any. in this scheme \x09 would be tab, \x0D carriage return, \xae for the registered trademark sign, \u03a3 for the greek capital letter sigma, and \U0001d15f for the musical symbol quarter note.

because this notation is a shorthand for a particular character it can only appear within quotes ('...' or [...]):
e.g. [xy\x200-9\u22c8Y_] means
('x' | 'y' | '\x20' | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '\u22c8' | 'Y' | '_').

- a notation for exactly one character from one of these character categories
  e.g. \p{Nd, Lo} for any character in category Nd or Lo.

because this notation is a short hand for a set of characters it can only appear within the character set quotes ([...]):
e.g. [\p{L}-\-\p{Nd}] means
('A' | 'B' | ... | '_' | '-' | '0' | '1' | ...)

- a notation for any character except the following:
e.g. [^aeiou>] for any character except a lowercase latin vowel or a greater-than sign

with this device, any time you want an actual caret character in a set, it cannot be the first one.
e.g. [^abc] is completely different from [a^bc]

our example unambiguous grammer is now a bit simpler:

EXP → TERM ^ [+-]
TERM → FACTOR ^ [*/]
FACTOR → NUMBER | '(' EXP ')'
NUMBER → [0-9]+

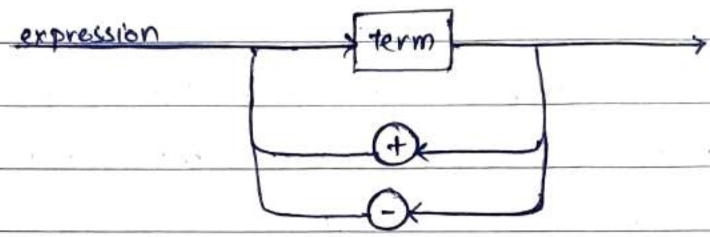
and the ambiguous one is really simple:

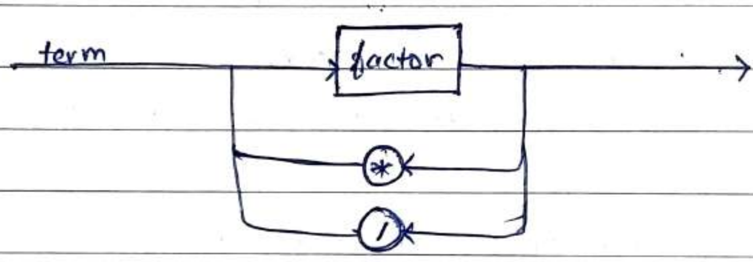
EXP → [0-9]+ | EXP [*/+-] EXP | '(' EXP ')'

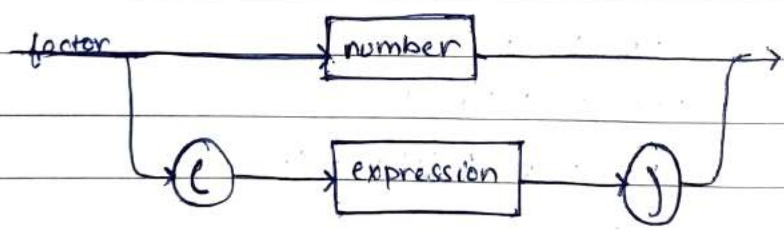
there is an international standard for EBNF if you are interested.

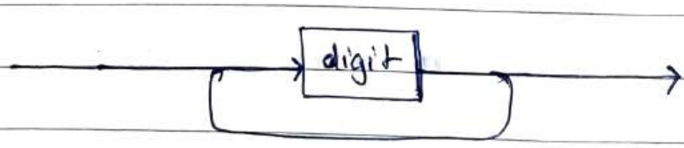
SYNTAX DIAGRAMS

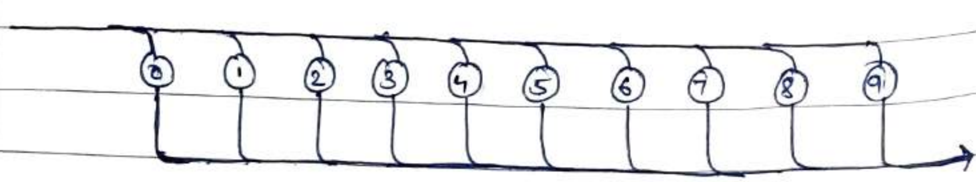
its pretty easy to generate diagrams from the EBNF. examples











PARSING EXPRESSION GRAMMARS (PEGS)

a PEG looks almost the same as an EBNF description but with a couple important differences.

- the choice operator, written as a slash (/) rather than a pipe, is an ordered operation, meaning that in the rule A ← B / C, if during a parse B matches, then B is taken and C is never tried.

- there is an and-predicate. in A ← &B C, while parsing A, we try B without consuming any input. if B can't match, we are done with a failure. if B does match, then we try C. this is a great way to describe "lookahead" in parsing.

- there is a not-predicate. we can write A ← B !C to say to parse A, you need a B not followed by C.

these features mean a PEG is never ambiguous.

here's our little example language expressed with a PEG:

exp ← term ([+-] term)*

term ← factor ([*/] factor)*

factor ← number / '(' exp ')'

number ← [0-9]+

and here's an ambiguous CFG (it's supposed to signify assignment statements and if-then-else statements in a programming language):

S → 'a' | 'i' 'x' 't' S ('e' S)?

which when given as a PEG is not ambiguous:

S ← 'a' / 'i' 'x' 't' S 'e' S / 'i' 'x' 't' S

ex: work out the ambiguity problem in the CFG and show how it is resolved in the PEG.

this is extremely cool & worth repeating. in a PEG, the so-called "dangling-else" problem does not exist.

WHAT ABOUT REAL PROGRAMMING LANGUAGES?

real languages have whitespace, comments, escape characters in string and character literals, multi-word identifiers and keywords. specifying the set of legal strings with a single set of rules is rather difficult and error-prone. here is an attempt to define the syntax of a C-like mini language.

```
PROGRAM → S* BLOCK S*
BLOCK → (DEC S* ';' S*)* (STMT S* ';' S*)+
DEC → 'var' S+ ID
    | 'fun' S+ ID S* '(' IDLIST? ')' S* '=' S* EXP
STMT → ID S* = S* EXP
    | 'read' S+ IDLIST
    | 'write' S+ IDLIST
    | 'while' S+ EXP S+ 'loop' S+ BLOCK 'end'
EXP → TERM (S* ADDOP S* TERM)
TERM → FACTOR (S* MULOP S* FACTOR)
FACTOR → INTLIT | ID | CALL | '(' S* EXP S* ')'
CALL → ID S* '(' S* EXPLIST? S* ')'
IDLIST → ID (S* ',' S* ID)
EXPLIST → EXP (S* ',' S* EXP)*
ADDOP → '+' | '-'
MULOP → '*' | '/'
INTLIT → DIGIT+
DIGIT → [\p{Nd}]
LETTER → [\p{L}]
ID → LETTER (LETTER | DIGIT | '_')* - KEYWORD
KEYWORD → 'var' | 'fun' | 'read' | 'write' | 'while' | 'loop' | 'end'
S → [\x20 \x09 \x0A \x0D] | COMMENT
COMMENT → '--' [^\x0A \x0D]* [\x0A \x0D]
```

how did we do?

- spaces and comments (together called skips) can appear almost anywhere; the grammar is really ugly.

- spaces are sometimes problematic. the above grammar requires spaces after the keyword 'while', which most people would say is too strict. the alternative is to add more rules to the grammar to distinguish expressions that begin with "c" and those that do not.

ex try it.

- there's an unpleasant mixing of low-level concerns like letters and digits from bigger chunks that matter, like identifiers and expressions. users of a language don't really think in terms of character strings, do they?

- when good engineers see a big mess, they break the mess up into simpler & neater parts. we need to break up the specification.

in other words, instead of worrying about the character string:

```
-- this doesn't write hello world

var x ;        var y; -- my first program
   -- uggh: lousy identation   and   SPACing

   while y-5 loop
   var y;
          read x, y;
x = 2 * (3+y);
  end;         -- ends a loop, i think
    write 5;
```

we should care about the token string

'var' 'ID(x)' ';' 'var' 'ID(y)' ';' 'while' 'ID(y)' '-' 'INTLIT(5)' 'loop' 'var' 'ID(y)' ';' 'read' 'ID(x)' ',' 'ID(y)' ';' 'ID(x)' '=' 'INTLIT(2)' '*' '(' 'INTLIT(3)' '+' 'ID(y)' ')' ';' 'end' ';' 'write' 'INTLIT(5)' ';'

notice that some of the tokens (ID and INTLIT to be specific) contain attributes. in a real compiler we could maintain other attributes, like line & column number, say.

MICROSYNTAX AND MACROSYNTAX

we generally define syntax in 2 parts:

- microsyntax: describes how characters are grouped into tokens, including, necessarily, how tokens are separated (eg. whitespace & comments). Key ideas: Keywords, identifiers, literals, operators, tokens, whitespace, comments.

- macrosyntax: a context-free grammar over sequences of tokens (not characters). key ideas: declarations, statements, expressions, blocks, subroutines, modules.

thus we would define the syntax above by saying:

a string is a syntactically valid program if it greedily matches $S^* T_0 S^* T_1 S^* T_2 \ldots S^* T_n S^*$ where

```
S → [x\20\x09\x0A\x0D] | '--' [^\x0A\x0D]* [\x0A\x0D]
T → INTLIT | ID | KEYWORD | SYMBOL
SYMBOL → [+\-*/=,;()]
ID → LETTER (LETTER | DIGIT | '_')* - KEYWORD
KEYWORD → 'var' | 'fun' | 'read' | 'write' | 'while' | 'loop' | 'end'
INTLIT → DIGIT+
DIGIT → [\p{Nd}]
LETTER → [\p{L}]
```

and $T_0 T_1 \ldots T_n$ is derivable from

```
PROGRAM -> BLOCK
BLOCK -> (DEC ';')* (STMT ';')+
DEC -> 'var' ID
     | 'fun' ID '(' IDLIST? ')' '=' EXP
STMT -> ID '=' EXP
     | 'read' IDLIST
     | 'write' EXPLIST
     | 'while' EXP 'loop' BLOCK 'end'
IDLIST -> ID (',' ID)*
EXPLIST -> EXP (',' EXP)*
EXP -> TERM ([+-] TERM)*
TERM -> FACTOR ([*/] FACTOR)*
FACTOR -> INTLIT | ID | CALL | '(' EXP ')'
CALL -> ID '(' EXPLIST? ')'
```

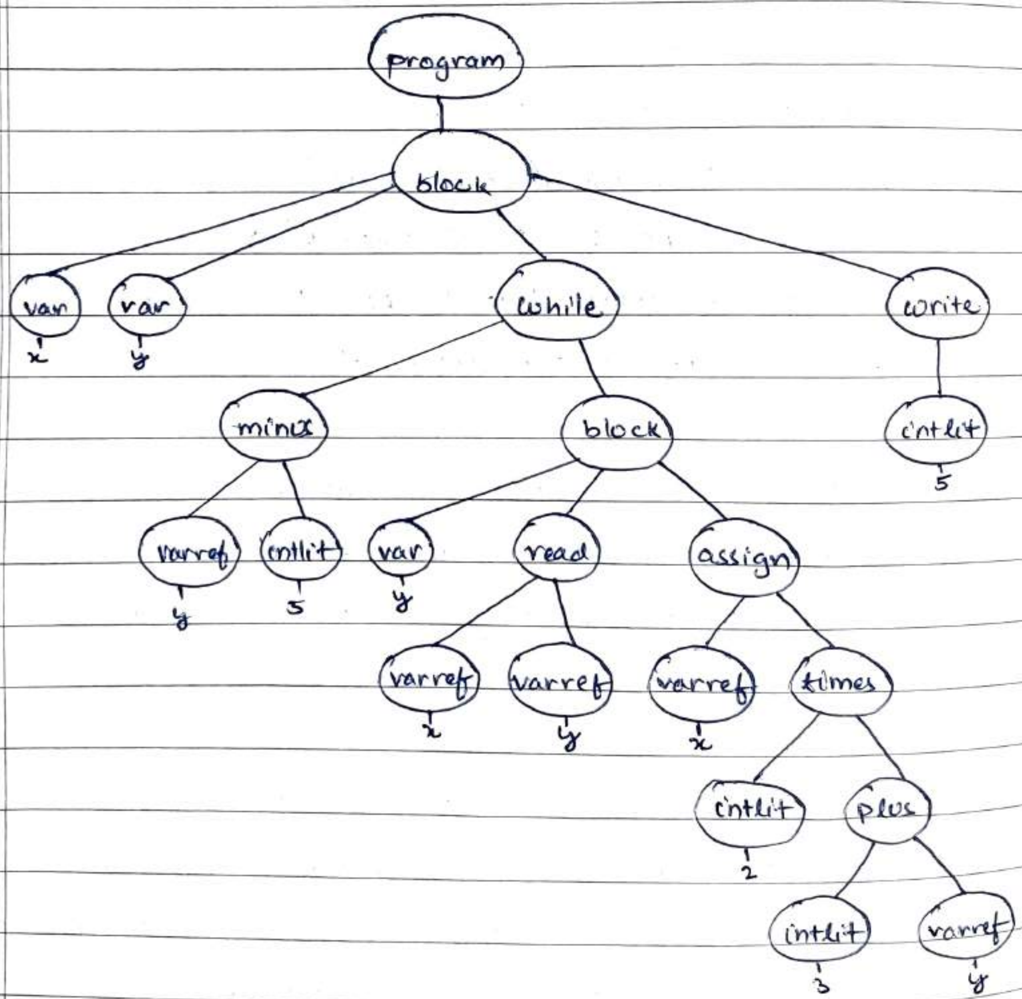
pulling out the skips from the macrosyntax makes the grammar much clearer. derivations & parse trees need only contain tokens & categories not characters, for example.
(,..) ->

ex rewrite the macro syntax for the above "little language" in BNF and as a plain context free grammar with undecorated right hand sides. also give a complete set of syntax diagrams for the language.

PEGs to the rescue... we are using ohm... >

ABSTRACT SYNTAX

you can go further and argue that certain punctuation like commas, semicolons, parentheses, are only artificial devices to make what should be hierarchical program be a flat one. the direct specification of the higher level structure is called abstract syntax. here is the abstract syntax tree corresponding to the parse tree we saw earlier:



what disappears from syntax when we move to abstract syntax?

- operator precedence
- paretheses and other grouping devices
- commas and other separators
- semicolons and other terminators
- multiple syntax categories for expressions
- keywords, like "begin", "end", "while", etc.

TREE GRAMMARS

abstract syntax is normally expressed with a tree grammer (recall that a tree grammar produces trees, whereas the chomsky grammars we saw above produce strings). for example the language above:

program → block
block → dec* stmt+
var: dec → id
fun: dec → id id* exp
assign: stmt → varexp exp
read: stmt → id*
write: stmt → exp*
while: stmt → exp block
plus: exp → exp exp
minus: exp → exp exp
times: exp → exp exp
divide: exp → exp exp
varexp: exp → id
intlit: exp → numeral
call: exp → id exp*

* exted the little language to allow non-value returning functions that are called as statements as well as if statements, break and continue statements, and floating point literals. given the EBNF, the abstract grammar, and attribute evaluation rules for the new language.

MULTIPLE CONCRETE SYNTAXES

it can be argued that abstract syntax captures the essence of a language's structure because multiple concrete syntaxes can be mapped to a single abstract syntax. here are some example programs in various languages that all have the same abstract syntax tree above:

```
program
  declare x, y
  // in this language, identation defines blocks
  while (y-5)
     declare y
     get (x)
     get (y)
     x := 2 * (3 + y)
  put (5)
```

```
new x, y
while (y -5) {
   new y;
   STDIN -> x;
   STDIN -> y;
   x <- 2* (3+y);
}
STDOUT <-5;
```

```
COMMENT THIS LOOKS LIKE OLD CODE
DECLARE INT X
DECLARE INT Y
WHILE DIFFERENCE OF Y AND 5 IS NOT ZERO LOOP:
    DECLARE INT Y
    READ FROM' STDIN INTO X
    READ FROM STDIN INTO Y
    MOVE PRODUCT O 2 AND (SUM OF X AND Y) INTO X
END WHILE
WRITE 5 TO STDOUT
```

```
(program
  (define x y)
  (while (- y 5)
     (seq (define y) (read x, y) (assign x (* 2 (+ 3 y))))
  )
  (write 5)
)
```

```
<program>
  <declare vars = "x, y"/>
  <while>
    <minus><varexp ref = "y" /> <intlit value = "5" /> </minus>
    <declare vars = "y"/>
    <read vars = "x y" />
    <assign varexp = "x" >
      <times>
        <intlit value = "2"/>
        <plus>
          <intlit value = "3" />
          <varexp ref = "y" />
        </plus>
      </times>
    </assign>
  </while>
  <write>
    <intlit value = "5"/>
  </write>
</program>
```

ex write the example in JSON like notation.

PARSING CONCERNS

a language's syntax is normally defined with a CFG or an equally powerful representation (BNF, EBNF, syntax diagram), or a PEG. a compiler uses the grammar to parse an input string, that is to determine if the string can be generated by the grammar, and produce the syntax tree for that string. some important facts:

- a grammar for a real programming language should be unambiguous.
- there is a theorm that says that every CFG can be parsed in $\Theta(n^3)$ time.
- LL grammars and LR grammars are (restricted) CFGs that can be parsed in $\Theta(n)$ time.
- PEGs are a little different from CFGs. they can be parsed in linear time with a packrat parser, but at the cost of a lot of additional space. Ohm adds some power to PEGs (left recursion) so you might encounter superlinear parse times.

what do LL and LR mean?

LL(k) grammar:

a grammar that can be used to drive a leftmost derivation by reading the input from left-to-right, examining only the next k symbols, and never backing up.

LR (K) grammar

a grammar that can be used to drive a right most derivation by reading the input from left-to-right, examining the next k symbols, and never backing up.

an. LL and LR fact sheet:

LL grammars	LR grammars
• subset of CFGs	• subset of CFGs
• never ambiguous	• never ambiguous
• parseable in linear time	• parseable in linear time
• left-to-right, leftmost derivation	• left-to-right, rightmost derivation
• top-down	• bottom-up
• predictive	• not predictive
• expand-match	• shift-reduce
• parsers can be hand-made	• nearly impossible to write parsers by hand
• can give nice error messages	• must work hard to give good errors
• cannot handle left-recursion	• left-recursion is desirable!
• less "powerful" than LR	• more "powerful" than LL

details of LL and LR: how to determine if grammar is LL:

1. locate all the "choice points" in the grammar (these are easy to find in the equivalent syntax diagram).

2. determine the maximum no. of tokens, k; required to know exactly which path to take over all choice points. if k is finite, the grammar is LL(k).

ex why is that a grammar with left-recursion cannot be LL(k) for any k?

WHAT ABOUT CONTEXT-SENSITIVE GRAMMARS?

can't we just write a context-sensitive grammar to define the set of legal programs? :c HAH!!! context-sensitive grammars are:

- often impossible to think up,
- nearly impossible for a human to read,
- horribly expensive even for a computer to parse.

try the following exercise if you're not convinced:

ex write context-sensitive grammars for

- $\{0^n 1^n 2^n \mid n \geq 0\}$
- $\{a^{2^n} \mid n \geq 0\}$
- $\{ww \mid w \in \{a, b\}^*\}$

ex do some research to see if there are any known lower complexity bounds on the parsing of context-sensitive grammars.

however, PEGs can sometimes be written for languages which are known to be context-sensitive and not context-free. here is an example (from wikipedia) for the non-context-free language $\{a^n b^n c^n \mid n \geq 0\}$

S ← &(A 'c') 'a'+ B !('a'/'b'/'c')

A ← 'a' A? 'b'

B ← 'b' B? 'c'