Syntax: Concrete & Abstract

- Language syntax describes the legal form and structure of programs
- Concrete syntax describes is what a program looks like on the page or screen
- Abstract syntax describes the essential contents of a program as it might be represented internally (e.g. by an interpreter or compiler)
- In this course, we won’t worry much about concrete syntax, but it is worth some brief discussion
- (And we will need to choose some concrete syntax for the programs we interpret)
- Syntax is specified by a grammar.
CONTEXT-FREE GRAMMARS

- Used for description, parsing, analysis, etc.
- Based on **recursive** definition of program structure.
- A grammar is defined by:
  - a set of **terminal** symbols (strings of characters)
  - a set **nonterminal** variables, which represent sets of terminals
  - a set of **production** rules that map nonterminals to strings of terminals and non-terminals
- The **language** defined by a grammar is the **set of strings of terminals** that can be derived by applying production rules, starting from a specified nonterminal.
- Grammars have rich theory with connections to automatic parser generation, push-down automata, etc.
- Many possible representations, including **BNF** (Backus-Naur Form), **EBNF** (Extended BNF), syntax charts, etc.
BNF was invented ca. 1960 and used in the formal description of Algol-60. It is just a particular notation for grammars, in which

- Nonterminals are represented by names inside angle brackets, e.g., `<program>`, `<expression>`, `<S>`.

- Terminals are represented by themselves, e.g., WHILE, `,`, 3. The empty string is written as `<empty>`.

**BNF example...**
\[\textbf{<program>} ::= \text{BEGIN} \textbf{<statement-seq>} \] 
\[\quad \text{END} \]
\[\textbf{<statement-seq>} ::= \textbf{<statement>} \]
\[\textbf{<statement-seq>} ::= \textbf{<statement>} ; \]
\[\quad \textbf{<statement-seq>} \]
\[\textbf{<statement>} ::= \textbf{<while-statement>} \]
\[\textbf{<statement>} ::= \textbf{<for-statement>} \]
\[\textbf{<statement>} ::= \textbf{empty} \]
\[\textbf{<while-statement>} ::= \text{WHILE} \textbf{<expression>} \]
\[\quad \text{DO} \textbf{<statement-seq>} \text{END} \]
\[\textbf{<expression>} ::= \textbf{<factor>} \]
\[\textbf{<expression>} ::= \textbf{<factor>} \text{AND} \textbf{<factor>} \]
\[\textbf{<expression>} ::= \textbf{<factor>} \text{OR} \textbf{<factor>} \]
\[\textbf{<factor>} ::= ( \textbf{<expression>} ) \]
\[\textbf{<factor>} ::= \textbf{<variable>} \]
\[\textbf{<for-statement>} ::= \ldots \]
\[\textbf{<variable>} ::= \ldots \]
EBNF is (any) extension of BNF, usually with these features:

- A vertical bar, |, represents a choice,
- Parentheses, ( and ), represent grouping,
- Square brackets, [ and ], represent an optional construct,
- Curly braces, { and }, represent zero or more repetitions,
- Nonterminals begin with upper-case letters.
- Non-alphabetic terminal symbols are quoted, at least when necessary to avoid confusion with the meta-symbols above.

EBNF is often used in language reference manuals and standards.
EBNF Example

Program ::= BEGIN Statement-seq END
Statement-seq ::= Statement [ ';' Statement-seq ]
Statement ::= [ While-statement | For-statement ]
While-statement ::= WHILE Expression DO Statement-seq END
Expression ::= Factor { (AND | OR) Factor }
Factor ::= '(' Expression ')' | Variable
For-statement ::= ...
Variable ::= ...
Syntax Analysis (Parsing)

Parser recognizes syntactically legal programs (as defined by a grammar) and rejects illegal ones.

- Successful parse also captures hierarchical structure of programs (expressions, blocks, etc.).
- Convenient representation for further semantic checking (e.g., typechecking) and for code generation.
- Failed parse provides error feedback to the user indicating where and why the input was illegal.

Any context-free language can be parsed by a computer program, but only some can be parsed efficiently. Modern programming languages are usually designed to be parsed efficiently.
Lexical Analysis

Programming language grammars usually take simple tokens rather than characters as terminals. Converting raw program text into token stream is job of the lexical analyzer, which

- Detects and identifies keywords and identifiers.
- Converts multi-character symbols into single tokens.
- Handles numeric and string literals.
- Removes whitespace and comments.
Graphical representation of a derivation.

Given this grammar:

\[
expr \rightarrow \ expr + \ expr \mid \ expr \ast \ expr \mid (\ expr \ ) \mid -\ expr \mid \ id
\]

Example tree for derivation of sentence \(-(x + y)\):

Each application of a production corresponds to an internal node, labeled with a non-terminal.

Leaves are labeled with terminals, which can have attributes (in this case the specific identifier name).

The derived sentence is found by reading leaves (or “fringe”) left-to-right.
A given string in the language defined by a grammar may have more than one parse tree. Grammars for which this is true are called **ambiguous**.

Example: given the grammar on the last slide, the sentence

\[ a + b * c \]

has two parse trees:

We may think of the left tree as being the “correct” one, but nothing in the grammar says this.

To avoid the problems of ambiguity, we can:

- Rewrite grammar; or
- Use “disambiguating rules” when we implement parser for grammar.
Ambiguity in Arithmetic Expressions

To disambiguate a grammar like

\[ E \rightarrow E + E \mid E - E \mid E \times E \mid E / E \mid (E) \mid \text{id} \]

we need to make choices about the desired order of operations.

For any expression of the form \((X \ op_1 Y \ op_2 Z)\) we must define:

- **Precedence** - which operation \((op_1\ or \ op_2)\) is done first?

- **Associativity** - if \(op_1\ and \ op_2\) have the same precedence, then does \(Y\) “associate” with the operator on the left or on the right?

In other words, we need rules to tell us whether the expression is equivalent to \(((X \ op_1 Y) \ op_2 Z)\) or to \((X \ op_1 (Y \ op_2 Z))\).

The “usual” rules (based on common usage in written math) give \(*\ and \ /\) higher precedence than \(+\ and \ -\), and make all the operators left-associative.

So, for example, \(a - b - c \times d\) is equivalent to \((a - b) - (c \times d)\).

But this is a matter of **choice** when defining the language.
Rewriting Arithmetic Grammars

One way to enforce precedence/associativity is to build them into the grammar using extra non-terminals, e.g.:

\[
\begin{align*}
\text{factor} & \rightarrow (\text{expr}) \mid \text{id} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \mid \text{term} \div \text{factor} \mid \text{factor} \\
\text{expr} & \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}
\end{align*}
\]

Example: \( a \ast b - c + d \ast e \)
LIMITATIONS OF CONTEXT-FREE GRAMMARS

Context-free grammars are very useful for describing the structure of programming languages and identifying legal programs.

But there are many useful characteristics of legal programs that cannot be captured in a grammar (no matter how clever we are).

For example, in many programming languages, every variable in a legal program must be declared before it is used. But this property cannot be captured in a grammar.

So checking legality of programs typically requires more than syntax analysis. Most compilers use a secondary “semantic” analysis phase to check non-syntactic properties, such as type-correctness. Of course, sometimes illegal programs cannot be detected until runtime.
**Parse Trees vs. Abstract Syntax Trees**

Parse trees reflect details of the **concrete** syntax of a program, which is typically designed for easy parsing.

For processing a language, we usually want a **simpler**, more **abstract** view of the program. (No firm rules about AST design: matter of taste, convenience.)

**Simple concrete grammar:**

\[
S \rightarrow \text{while} \; '(' \; E \; ')' \; \text{do} \; S \mid ID \; '=' \; E \\
E \rightarrow E \; '+' \; T \mid E \; '-' \; T \mid T \\
T \rightarrow ID \mid NUM \mid '(' \; E \; ')'
\]
while (n) do n = n - (b + 1)
Possible abstract syntax tree for while (n) do n = n - (b + 1)

```
While
   /
  Id  Assgn
   n   n
    |
   Sub
      /
     Id  Add
      n   1
      /
     Id  Num
      b   1
```

Note that tree nodes may have attributes (such as the name of an Id) and/or sub-trees.
Abstract Syntax Captures the Essence

Concrete syntax is important for usability, but fundamentally superficial. The same abstract syntax can be used to represent many different concrete syntaxes.

Examples:

- C-like:

  ```
  while (n) do n = n - (b + 1);
  ```

- Fortran-like:

  ```
  do while(n .NE. 0)
      n = n - (b + 1)
  end do
  ```
CONCRETE SYNTAX EXAMPLES (2)

- COBOL-like:

```plaintext
PERFORM 100-LOOP-BODY
   WITH TEST BEFORE
   WHILE N IS NOT EQUAL TO 0
100_LOOP-BODY.
   ADD B TO 1 GIVING T
   SUBTRACT T FROM N GIVING N
```

- Use Chinese keywords in place of `while` and `do`.

- Use a graphical notation.
INTERNAL REPRESENTATION OF AST’S IN SCHEME

AST’s have recursive structure and irregular shape and size, so it makes sense to store them as a linked data structures in the heap, using one record for each tree node.

(define-type stmt
    [While (test expr?) (body stmt?)])
    [Assgn (lhs symbol?) (rhs expr?)])

(define-type expr
    [Add (left expr?) (right expr?)])
    [Sub (left expr?) (right expr?)]
    [Id (name symbol?)]
    [Num (value integer?)])

Note that, in Scheme, symbols are immutable literal strings that can be cheaply compared for equality. In program text, the symbol foo is written (quote foo) or simply ’foo.
In Java, heap records are **objects**. We define **classes** corresponding to the various kinds and a subclass for each label, e.g.

```java
abstract class Stmt { }
class While extends Stmt {
    Expr test;
    Stmt body;
}
class Assgn extends Stmt {
    String lhs; Expr rhs;
}
...
abstract class Expr { }
class Add extends Expr {
    Expr left; Expr right;
}
...
class Num extends Expr {
    int value;
}
```
INTERNAL REPRESENTATION OF ASTs IN C

In C, Pascal, Ada, etc., we might use unions or variant records for the different node labels of each node kind. E.g., in C:

```c
struct stmt {
    enum { While, Assgn, ... } label;
    union {
        struct {
            struct expr *test;
            struct stmt *body;
        } while_s;
        struct {
            char *lhs; struct expr *rhs;
        } assgn_s;
        ...
    } u;
}
```

```c
struct expr {
    enum { Add, ..., Num } label;
    union {
        struct {
            struct expr *left;
            struct expr *right;
        } add_e;
        ...
        struct {
            int value;
        } num_e;
    } u;
}
```
All these approaches generate roughly the **same** heap structures, e.g. for

```
while (n) do  n = n - (b + 1)
```

```
WHILE
  Stmt
    ASSGN
      Stmt
        SUB
          Exp
            ID
              Exp
                ID
                  Exp
                    ID
                      Exp
                        ID
                          Exp
                            ID
                              Exp
                                ID
                                  Exp
                                    LD
                                      Exp
                                        ID
                                          Exp
                                            LD
                                              Exp
                                                ID
                                                  Exp
                                                    LD
                                                      Exp
                                                        ID
                                                          Exp
                                                            LD
                                                              Exp
                                                                LD
                                                                  Exp
                                                                      ID
                                                                        Exp
                                                                          ID
                                                                            Exp
                                                                                     ```
Although ASTs are designed as an *internal* program representation, it can be useful to give them an *external* concrete form too that can be read or written by other programs or by humans.

Any external representation of ASTs must accurately reflect the internal tree structure as well as the “fringe” of the tree.

One well-established approach is to use *parenthesized prefix notation* to represent trees.

Each node in the tree is represented by the expression

\[
( \text{label} \ \text{attr}_1 \ldots \ \text{attr}_m \ \text{child}_1 \ \text{child}_n )
\]

where *label* is the node label, the *attr*\(_i\) are the node’s attributes (if any), and the *child*\(_i\) are the node’s sub-trees (if any), each of which is itself a node expression. To make things more readable, we might abbreviate for some labels, e.g., use + for Add. We may also represent simple leaf nodes by their bare attributes, e.g., use 3 for (Num 3), as long as no confusion can arise.
So the representation of our AST example could be

(While n
  (Assign n (- n
    (+ b
      1)))))

where the indentation is optional, but makes the representation easier for humans to read.

The BNF for this syntax can be given as:

<stmt> ::= (<While> <expr> <stmt>)
  | (<Assign> <name> <expr>)

<expr> ::= (+ <expr> <expr>)
  | (- <expr> <expr>)
  | <name>
  | <int>

Concrete and abstract syntax are isomorphic.
PARSING PARANTHESIZED PREFIX NOTATION

Note that this BNF description contains essentially the same information as our AST internal datatype declarations (except for the descriptive field names).

Parsing this representation is easy! Why?

- Everything is either an atom (keyword, symbol, numeric literal, etc.) or a parenthesized list of atoms
- Each node in the AST corresponds to a list or atom.
- The first item in each list is always a symbol that identifies the node type and implies the kind and number of the remaining things in the list.
- The lexical analyzer just needs to identify the atoms and list delimiters.

Does this sound familiar?
In fact, parenthesized prefix notation (usually called S-expressions) is what Scheme uses as a concrete syntax for its programs!

So Scheme’s concrete and abstract syntax are essentially identical. (What is good and bad about this?)

Because S-expressions are central to Scheme, the language has good built-in support for analyzing them, via the interactive Read-Eval-Print Loop (REPL) or using \texttt{read} primitive function, which can be applied to standard input or to a file. These facilities work even if we want to use S-expressions to represent programs in the “toy” language we are interpreting rather than to represent Scheme programs.
The following all define the same value for \( x \):

\[
\text{(define } x \text{ (quote (While } n \text{ (Assgn } n (\text{- } n (\text{+ } b \text{ 1})))))}
\]

\[
\text{(define } x \text{ '}(\text{While } n \text{ (Assgn } n (\text{- } n (\text{+ } b \text{ 1})))))
\]

\[
\text{(define } x \text{ (read))}
\]

where we type \( (\text{While } n \text{ (Assgn } n (\text{- } n (\text{+ } b \text{ 1})))) \) into the input box

\[
\text{(define } x \text{ (list 'While 'n}
\text{ (list 'Assgn 'n (list '– 'n}
\text{ (list '+ 'b '1)))))}
\]

In all cases, \( x \) prints simply as:

\[
(\text{While } n \text{ (Assgn } n (\text{- } n (\text{+ } b \text{ 1})))}
\]
Here's a parser from S-expression lists into our `stmt` and `exp` types:

```
(define (parse-stmt sexp)
  (cond
    [(list? sexp)
      (case (first sexp)
        [(While) (While (parse-expr (second sexp))
                       (parse-stmt (third sexp)))]
        [(Assgn) (Assgn (second sexp)
                        (parse-expr (third sexp)))]))

(define (parse-expr sexp)
  (cond
    [(number? sexp) (Num sexp)]
    [(symbol? sexp) (Id sexp)]
    [(list? sexp)
      (case (first sexp)
        [(+) (Add (parse-expr (second sexp))
                 (parse-expr (third sexp)))]
        [(-) (Sub (parse-expr (second sexp))
                 (parse-expr (third sexp)))]))

> (parse-stmt x)
#(struct:While #(struct:Id n) #(struct:Assgn n
                #(struct:Sub #(struct:Id n) #(struct:Add #(struct:Id b) #(struct:Num 1))))))
```