Syntax (Easy)
What do programs look like?
- Grammars; BNF and Syntax Charts

Semantics (Hard)
What do programs do?
- Informal: Reference manuals, user guides, examples
- Formal: Operational, Denotational, Axiomatic
- Experimental: Try running the program and see what happens!

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 actually quite important to the style of a language
- Concrete syntax is defined by a context-free grammar

Simple Formal Example Grammar
Character set: \{ (, ) \}
Terminals: \{ (, ) \}
Non-terminals: \{ S \}
Productions:
\[ S ::= ( S ) \]
\[ S ::= SS \]
\[ S ::= \epsilon \] (the empty string)
Starting non-terminal: S
Sample derivation:
\[ S \to ( S ) \to ( SS ) \to (( S ) S) \to ( ) S \to ( ) ( S ) \to ( ) ( ) \]
This grammar generates the language of strings of properly matched parentheses.
It is often useful to think of a derivation as a tree (more shortly).
CONTEXT-FREE GRAMMARS

- Used for description, parsing, analysis, etc.
- Especially good for recursive definition of program structure
- A grammar is defined by:
  - a set of terminal symbols (strings of characters)
  - a set non-terminal variables, which represent sets of strings of terminals
  - a set of production rules that map non-terminals to strings of terminals and non-terminals
- The language $L(G)$ defined by a grammar $G$ is the set of strings of terminals that can be derived by applying production rules beginning from a specified start symbol (a non-terminal).
- 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
- Non-terminals 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...

```
<program> ::= BEGIN <statement-seq> END
<statement-seq> ::= <statement>
<statement-seq> ::= <statement> ; <statement-seq>
<statement> ::= <while-statement>
<statement> ::= <for-statement>
<statement> ::= <empty>
<while-statement> ::= WHILE <expression> DO <statement-seq> END
<expression> ::= <factor>
<expression> ::= <factor> AND <factor>
<expression> ::= <factor> OR <factor>
<factor> ::= ( <expression> )
<factor> ::= <variable>
<for-statement> ::= ...
<variable> ::= ...
```

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,
- Non-terminals begin with upper-case letters.
- Non-alphabetic terminal symbols are quoted, at least when necessary to avoid confusion with the meta-symbols above.
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 ::= ...

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.

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 can usually be parsed efficiently.

Parse Trees

Graphical representation of a derivation.

Given this grammar:

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

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

```
expr
  /  
expr expr
    / 
  expr expr
    / 
  id id
```

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 sentence in $L(G)$ can have more than one parse tree. Grammars $G$ 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.

To disambiguate a grammar like

$$E \rightarrow E + E | E - E | E * E | E / E | (E) | id$$

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, $a - b - c * d$ is equivalent to $(a - b) - (c * d)$. But this is a matter of choice when defining the language.

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

- $factor \rightarrow (expr) | id$
- $term \rightarrow term * factor \ | \ term / factor \ | \ factor$
- $expr \rightarrow expr + term \ | \ expr - term \ | \ term$

Example: $a * b - c + d * e$
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:

\[
S \rightarrow \text{while } (' E ') \text{ do } S \mid ID \ ' = ' E \\
E \rightarrow E \ '+' T \mid E \ '-' T \mid T \\
T \rightarrow ID \mid \text{NUM} \mid (' E ')'
\]

PARSE TREES vs. ABSTRACT SYNTAX TREES (2)

Possible abstract syntax tree for **while (n) do n = n - (b + 1)**

```
While
   |   Assign n
   |   Subtract
   |   Id n
   |   Add
   |   Id b
   |   Numeric 1
```

Note that tree nodes may have **attributes** (such as the name of an \textit{Id}) and/or **sub-trees**.

TREE GRAMMARS

AST's obey a **tree grammar**. Rules have form

\[
\text{label} : \text{kind} \rightarrow (\text{attr}_1 \ldots \text{attr}_m) \ \text{kind}_1 \ldots \text{kind}_n
\]

where the LHS classifies the possible node **labels** into **kinds**, and the RHS describes the label's atomic **attributes** (if any, in parentheses) and the kinds of its subtrees (if any).

Example:

\[
\text{While} : \text{Stmt} \rightarrow \text{Exp} \ \text{Stmt} \\
\text{Assign} : \text{Stmt} \rightarrow (\text{string}) \ \text{Exp} \\
\text{Add} : \text{Exp} \rightarrow \text{Exp} \ \text{Exp} \\
\text{Sub} : \text{Exp} \rightarrow \text{Exp} \ \text{Exp} \\
\text{Id} : \text{Exp} \rightarrow (\text{string}) \\
\text{Num} : \text{Exp} \rightarrow (\text{int})
\]
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:
  ```c
  while (n) do n = n - (b + 1);
  ```

- Fortran-like:
  ```fortran
  do while(n .NE. 0)
  n = n - (b + 1)
  end do
  ```

- COBOL-like:
  ```cobol
  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.

AST's in Scala

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

In Scala, heap records are objects. We define classes corresponding to the various kinds and a case class for each label, e.g.

```
sealed abstract class Stmt
case class While(test:Exp,body:Stmt) extends Stmt
case class Assign(lhs:String,rhs:Exp) extends Stmt
sealed abstract class Exp
case class Add(left:Exp,right:Exp) extends Exp
case class Num(value:Int) extends Exp
```

The case class declarations also define constructors, so we can say, e.g., `val e:Exp = Num(42)`.

Heap Structure

Using these constructors, we generate a heap structure that is isomorphic to the AST tree, e.g. for `while (n) do n = n - (b + 1)`
Although ASTs are designed as an internal program representation, it can be useful to give them an external 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. Can’t use tree grammar to parse, since it is typically ambiguous!

We will represent trees using parenthesized prefix notation, also called s-expressions (the name comes from the programming language LISP). Each node in the tree is represented by the expression

\[( \text{label} \ attr_1 \ldots \ attr_m \ child_1 \ child_n \)\]

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

So the representation of our AST example could be

\[(\text{While} (\text{Id} n)\)
\n\n\n(\text{Assign} n (- (\text{Id} n)\)
\n\n\n(+ (\text{Id} b)\)
\n\n\n(\text{Num} 1))))\]

where the indentation is optional, but makes the representation easier for humans to read. The BNF for this syntax can be given as:

\[<\text{stmt}> ::= (<\text{While}> <\text{expr}> <\text{stmt}>)
| (\text{Assign} <\text{name}> <\text{expr}>)
\]

\[<\text{expr}> ::= (+ <\text{expr}> <\text{expr}>)
| (- <\text{expr}> <\text{expr}>)
| <\text{name}>
| <\text{int}>
\]

Concrete and abstract syntax are isomorphic.