What is Syntax?

- The form of its expressions, statements, and program units
- e.g. it describes what a "while loop" looks like
- But it does not describe what the program means
- e.g. `c = a + b;` is a syntactically valid statement. What does it mean?

Syntax Description

- Programming languages are defined by formal rules
- Lexemes are small units of characters that are usually considered indivisible
- e.g. `varname`, `1234`
- Tokens are categories of lexemes (e.g. identifier, semicolon, plus operator, string literals)
- A lexical analyzer tokenizes the program
- Different types of tokens are often described by regular expressions (not covered in this course)

Syntax Description

- The syntax of the language is usually described by a grammar
- Backus-Naur Form (BNF)
- A "metalanguage" to describe other languages

Backus-Naur Form

- Abstractions are enclosed in `< >`. e.g. `<var>`
- Each abstraction is defined by a number of rules of the form:

$$< \text{abs} > \rightarrow \ldots$$

  For example,

$$< \text{assign} > \rightarrow < \text{var} > = < \text{expression} >$$

$$< \text{expression} > \rightarrow < \text{term} > + < \text{expression} >$$

- The definition is sometimes called rules or productions.
- The left-hand side (LHS) is defined by the right-hand side (RHS).

Backus-Naur Form

- Nonterminals are abstractions: they must be defined by rules
- Terminals are lexemes and tokens: they are not further defined
- Many nonterminals have multiple different rules
- It is common (necessary?) to have recursion
- Parser translates tokenized programs into parse trees

BNF Examples

- If-then and If-then-else statements
- Lists of items
- Arithmetic expressions

Derivations

- A grammar usually has a start symbol (e.g. `<program>`)
- A derivation starts with the start symbol, and repeatedly chooses a rule whose LHS matches a nonterminal and replaces it by the RHS.
- A derivation is finished when the current string has only terminals.
- At any point, there may be multiple nonterminals that can be replaced: leftmost derivations always chooses the leftmost one
- There may also be multiple rules that can be applied to a given nonterminal.
- The parser has to choose the correct rule so the final string of terminals is the given source code.

Parse Trees

- Describes syntactic structure of a program
- The root is the start symbol
- Each time a nonterminal is replaced by its RHS, children nodes are added corresponding to RHS
- Nonterminals are internal nodes, terminals are leave nodes
- An inorder traversal of the leave nodes should give the original source code

Ambiguity

- A grammar is ambiguous if there can be two different parse trees for the same expressions.
- We do not want ambiguity generally: what does it mean?
- It may depend on the context (need extra information).

Operator Precedence and Associativity

- When writing grammars, we want operators with higher precedence to be "deeper down" in the parse trees
- Operator precedence describes which operators are applied first when multiple types of operators are involved
- Associativity describes which operators are applied first when operators with the same precedence are involved
- Example: arithmetic operators

Extended BNF

- More convenient form of BNF
- Optional part: use square brackets
- Optional repeated part: use braces
- Multiple choice: use parentheses and vertical bars.
- Example: arithmetic expressions

Attribute Grammar (Advanced)

- Context-free grammars do not completely specify the language
- Without context, the parser does not know:
  - if variable has been declared
  - data types are compatible
  - ...
- In attribute grammar, every nonterminal has a set of attributes to keep track of context (e.g. data type)
- Rules are used to pass attributes up and down the parse tree
- Predicates (boolean functions) are used to check the attribute at each node

Semantics

- Semantics refer to the meaning of the program: what does it do?
- Many alternative ways to describe semantics:
  - Operational
  - Denotational
  - Axiomatic

Operational Semantics

- Describe the meaning of each statement by specifying the effect of running it on a machine
- Need to define machine states:
  - memory content
  - registers (data and program counter)
  - etc.
- Often use an informal "abstract machine" for platform independence
- Not always very precise
- Each statement defines how the state of the machine changes.

Denotational Semantics

- Mathematical way to specify the meaning of each statement
- Each language construct is associated with a mathematical object
- Each language construct has a function to map it to the appropriate mathematical objects and how they are manipulated
- The state of the machine can be denoted as a set of ordered pairs (variable, value).
- It is very precise mathematically and can be used to reason about the program.
- It is also very complex to describe and understand.

Axiomatic Semantics

- Uses predicate logic
- Can be used to prove program correctness
- For each statement, assertions (boolean expressions) are used to describe what are true before and after the statement
- Precondition: an assertion before a statement, usually describes relationship and constraints of input to the statement
- Postcondition: an assertion after a statement, usually describes the result of the statement
- Typically put in braces before and after a statement

Axiomatic Semantics

- One way to prove program correctness:
  - Write a postcondition for the final statement that implies correctness of the program
  - Find the weakest precondition for the final statement that makes the postcondition true
  - Continue working backwards until we arrive at the weakest precondition for the first statement
  - Check that input specification implies the precondition of the first statement

Weakest Precondition

- If Q is a given postcondition, then the weakest precondition P:
  - guarantees Q after the statement
  - for any other precondition P' that guarantees Q after the statement, P' implies P.
- That is, P is what is needed to guarantee Q, nothing more.

Program Proofs

- The proof of a program is simply a list of the statements with their preconditions and postconditions.
- We have a sequence of statements $S_1, \ldots, S_n$ and associated preconditions $P_1, \ldots, P_n$ and postconditions $Q_1, \ldots, Q_n$:
  - Input specification implies P_1
  - P_i guarantees Q_i
  - Q_i implies P_{i+1}
  - Q_n implies output specification

Program Proof Example

- Assignment: `{P} x = 2 * y - 3 {x > 25}`

Program Proof for Selection

- For the statement:

```
{P}
if B then
  S1
else
  S2
{Q}
```

- Need to consider both branches:
  - `{P and B} S1 {Q}`
  - `{P and not B} S2 {Q}`

Program Proof for Loops

- Consider only while loops:

```
while B do
  S
end
```

- Idea: use mathematical induction
- Need a loop invariant I that is true before, during each iteration, and after the loop.
- Need `{I and B} S {I}`
- Loop precondition is `{I}`
- Loop postcondition is `{I and not B}`

Program Proof for Loops Example

- Linear search in array

Program Proof for Loops

- Does the loop terminate?
- Need a loop variant: a non-negative integer quantity that decreases after each loop iteration.
- For example, number of array elements not yet examined.
- Limitations: very difficult to obtain useful variants and invariants in general

Axiomatic Semantics

- Can be used to prove program correctness
- Can be very difficult to use for large programs (tedious)