Syntax

A programming language consists of syntax, semantics, and pragmatics. We formalize syntax first, because only syntactically correct programs have semantics. A syntax definition of a language lists the symbols for building words, the word structure, the structure of well formed phrases, and the sentence structure. Here are two examples:

1. *Arithmetic*: The symbols include the digits from 0 to 9, the arithmetic operators +, −, ×, and /, and parentheses. The numerals built from the digits and the operators are the words. The phrases are the usual arithmetic expressions, and the sentences are just the phrases.
2. *A Pascal-like programming language*: The symbols are the letters, digits, operators, brackets, and the like, and the words are the identifiers, numerals, and operators. There are several kinds of phrases: identifiers and numerals can be combined with operators to form expressions, and expressions can be combined with identifiers and other operators to form statements such as assignments, conditionals, and declarations. Statements are combined to form programs, the ‘‘sentences’’ of Pascal.

These examples point out that languages have internal structure. A notation known as *Backus-Naur form* (BNF) is used to precisely specify this structure.

A BNF definition consists of a set of equations. The left-hand side of an equation is called a *nonterminal* and gives the name of a structural type in the language. The right-hand side lists the forms which belong to the structural type. These forms are built from symbols (called *terminal symbols*) and other nonterminals. The best introduction is through an example.

Consider a description of arithmetic. It includes two equations that define the structural types of *digit* and *operator:*

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<operator> ::= + | − | × | /

Each equation defines a group of objects with common structure. To be a digit, an object must be a 0 or a 1 or a 2 or a 3 . . . or a 9. The name to the left of the equals sign (::=) is the nonterminal name <digit>, the name of the structural type. Symbols such as 0, 1, and +

are terminal symbols. Read the vertical bar (|) as ‘‘or.’’

Another equation defines the numerals, the words of the language:

<numeral> ::= <digit> | <digit> <numeral>

The name <digit> comes in handy, for we can succinctly state that an object with numeral structure must either have digit structure or . . . or what? The second option says that a numeral may have the structure of a digit grouped (concatenated) with something that has a known numeral structure. This clever use of recursion permits us to define a structural type that has an infinite number of members.

The final rule is:

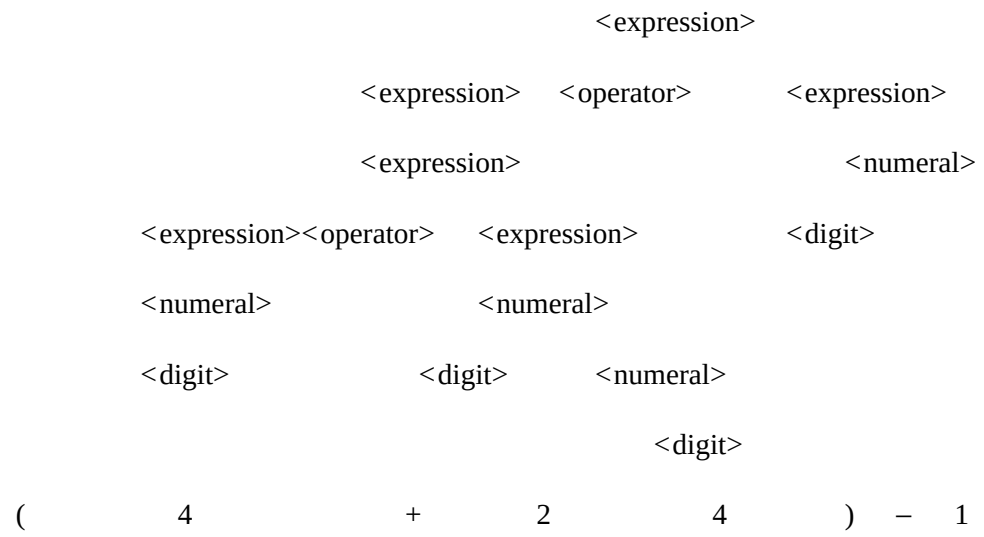
<expression> ::= <numeral> | (<expression>)
| <expression> <operator> <expression>

An expression can have one of three possible forms: it can be a numeral, or an expression enclosed in parentheses, or two expressions grouped around an operator.

The BNF definition of arithmetic consists of these four equations. The definition gives a complete and precise description of the syntax. An arithmetic expression such as $(4+24)-1$ is drawn as a *derivation tree*, so that the structure is apparent. Figure 1.1 shows the derivation tree for the expression just mentioned.

We won’t cover further the details of BNF; this information can be found in many other

Figure 1.1



texts. But there is one more notion that merits discussion. Consider the derivation trees in Figures 1.2 and 1.3 for the expression $4 \times 2 + 1$. Both are acceptable derivation trees. It is puzzling that one expression should possess *two* trees. A BNF definition that allows this phenomenon is called *ambiguous*. Since there are two allowable structures for $4 \times 2 + 1$, which one is proper? The choice is important, for real life compilers (and semantic definitions) assign meanings based on the structure. In this example, the two trees suggest a choice between multiplying four by two and then adding one versus adding two and one and then

Figure 1.2

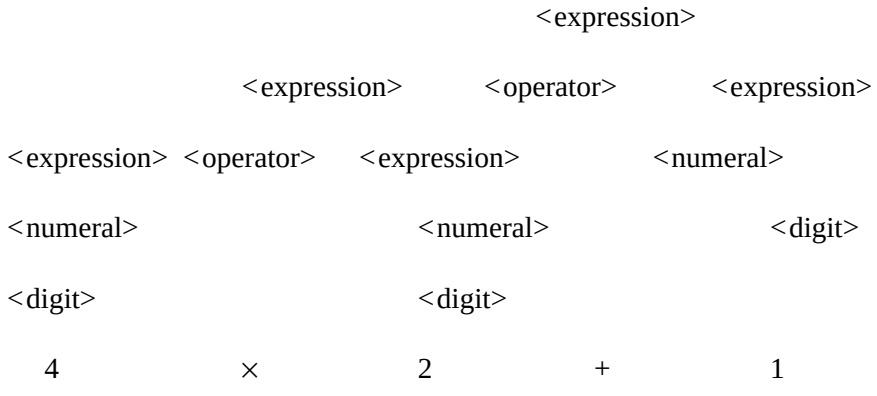
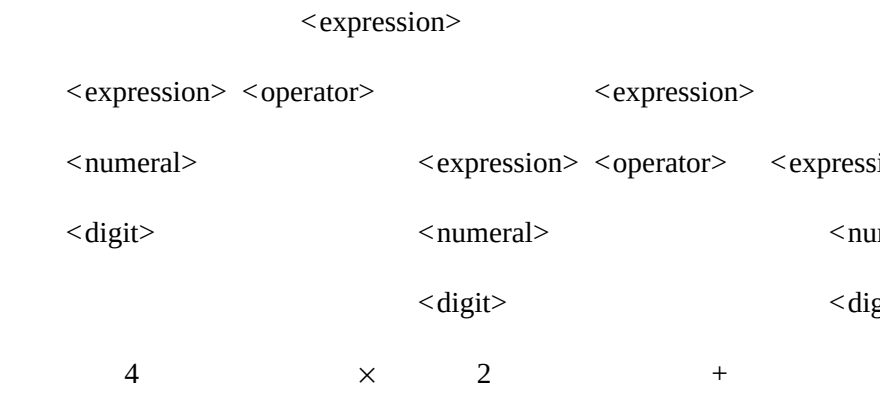


Figure 1.3



multiplying by four.

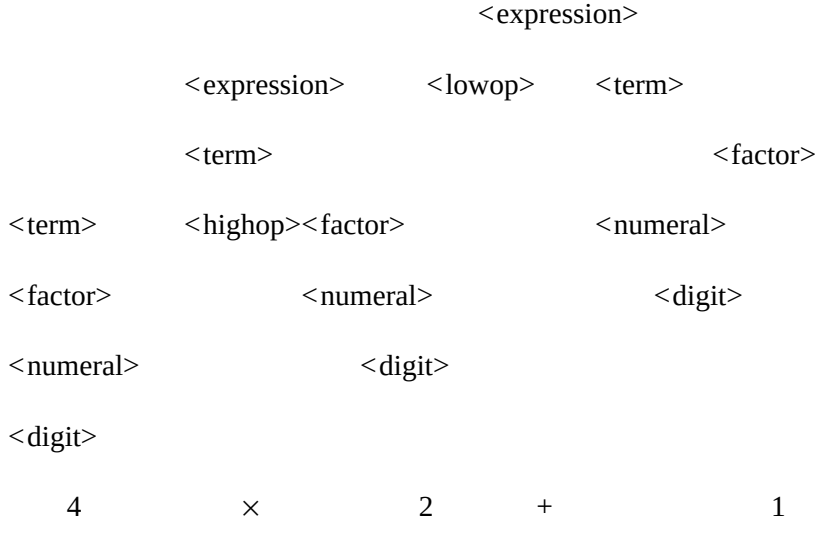
Ambiguous BNF definitions can often be rewritten into an unambiguous form, but the price paid is that the revised definitions contain extra, artificial levels of structure. An unambiguous definition of arithmetic reads:

<expression> ::= <expression> <lowop> <term> | <term>
<term> ::= <term> <highop> <factor> | <factor>
<factor> ::= <numeral> | (<expression>)
<lowop> ::= + | –
<highop> ::= × | /

(The rules for <numeral> and <digit> remain the same.) This definition solves the ambiguity problem, and now there is only one derivation tree for $4\times2+1$, given in Figure 1.4. The tree is more complex than the one in Figure 1.2 (or 1.3) and the intuitive structure of the expression is obscured. Compiler writers further extend BNF definitions so that fast parsers result. Must we use these modified, complex BNF definitions when we study semantics? The answer is no.

We claim that the derivation trees are the *real* sentences of a language, and strings of symbols are just abbreviations for the trees. Thus, the string $4\times2+1$ is an ambiguous abbreviation. The original BNF definition of arithmetic is adequate for specifying the structure of sentences (trees) of arithmetic, but it is not designed for assigning a unique derivation tree to a

Figure 1.4



string purporting to be a sentence. In real life, we use *two* BNF definitions: one to determine the derivation tree that a string abbreviates, and one to analyze the tree's structure and determine its semantics. Call these the *concrete* and *abstract syntax definitions*, respectively.

A formal relationship exists between an abstract syntax definition and its concrete counterpart. The tree generated for a string by the concrete definition identifies a derivation tree for the string in the abstract definition. For example, the concrete derivation tree for $4 \times 2 + 1$ in Figure 1.4 identifies the tree in Figure 1.2 because the branching structures of the trees match.

Concrete syntax definitions will no longer be used in this text. They handle parsing problems, which do not concern us. We will always work with derivation trees, not strings. Do remember that the concrete syntax definition is derived from the abstract one and that the abstract syntax definition is the true definition of language structure.

1.1 ABSTRACT SYNTAX DEFINITIONS

Abstract syntax definitions describe structure. Terminal symbols disappear entirely if we study abstract syntax at the word level. The building blocks of abstract syntax are words (also called *tokens*, as in compiling theory) rather than terminal symbols. This relates syntax to semantics more closely, for meanings are assigned to entire words, not to individual symbols.

Here is the abstract syntax definition of arithmetic once again, where the numerals, parentheses, and operators are treated as tokens:

<expression> ::= <numeral> | <expression> <operator> <expression>
| *left-paren* <expression> *right-paren*
<operator> ::= *plus* | *minus* | *mult* | *div*
<numeral> ::= *zero* | *one* | *two* | · · · | *ninety-nine* | *one-hundred* | · · ·

The structure of arithmetic remains, but all traces of text vanish. The derivation trees have the same structure as before, but the tree's leaves are tokens instead of symbols.

Set theory gives us an even more abstract view of abstract syntax. Say that each nonterminal in a BNF definition names the set of those phrases that have the structure specified by the nonterminal's BNF rule. But the rule can be discarded: we introduce syntax builder operations, one for each form on the right-hand side of the rule.

Figure 1.5 shows the set theoretic formulation of the syntax of arithmetic.

The language consists of three sets of values: expressions, arithmetic operators, and numerals. The members of the *Numeral* set are exactly those values built by the ''operations'' (in this case, they are really constants) *zero*, *one*, *two*, and so on. No other values are members of the *Numeral* set. Similarly, the *Operator* set contains just the four values denoted by the constants *plus*, *minus*, *mult*, and *div*. Members of the *Expression* set are built with the three operations *make-numeral-into-expression*, *make-compound-expression*, and *make-bracketed-expression*. Consider *make-numeral-into-expression*; it converts a value from the *Numeral* set into a value in the *Expression* set. The operation reflects the idea that any known numeral can be used as an expression. Similarly, *make-*

Figure 1.5

Sets:

Expression
Op
Numeral

Operations:

make-numeral-into-expression: *Numeral* → *Expression*
make-compound-expression: *Expression* × *Op* × *Expression* → *Expression*
make-bracketed-expression: *Expression* → *Expression*

plus: *Op*
minus: *Op*
mult: *Op*
div: *Op*

zero: *Numeral*
one: *Numeral*
two: *Numeral*
. . .
ninety-nine: *Numeral*
one-hundred: *Numeral*
. . .

compound-expression combines two known members of the *Expression* set with a member of the *Operation* set to build a member of the *Expression* set. Note that *make-bracketed-expression* does not need parenthesis tokens to complete its mapping; the parentheses were just ‘‘window dressing.’’ As an example, the expression 4 + 12 is represented by *make-compound-expression (make-numeral-into-expression(four), plus, make-numeral-into-expression(twelve))*.

When we work with the set theoretic formulation of abstract syntax, we forget about words and derivation trees and work in the world of sets and operations. The set theoretic approach reinforces our view that syntax is not tied to symbols; it is a matter of structure. We use the term *syntax domain* for a collection of values with common syntactic structure. Arithmetic has three syntax domains.

In this book, we use a more readable version of set-theoretic abstract syntax due to Strachey. We specify a language’s syntax by listing its syntax domains and its BNF rules. Figure 1.6 shows the syntax of a block-structured programming language in the new format.

As an example from Figure 1.6, the phrase B∈ Block indicates that Block is a syntax domain and that B is the nonterminal that represents an arbitrary member of the domain. The structure of blocks is given by the BNF rule B::= D;C which says that any block must consist of a declaration (represented by D) and a command (represented by C). The ; token isn’t really

Figure 1.6

Abstract syntax:

P∈ Program
B∈ Block
D∈ Declaration
C∈ Command
E∈ Expression
O∈ Operator
I∈ Identifier
N∈ Numeral

P ::= B.
B ::= D;C
D ::= **var** I | **procedure** I; C | D_1; D_2
C ::= I:=E | **if** E **then** C | **while** E **do** C | C_1;C_2 | **begin** B **end**
E ::= I | N | E_1 O E_2 | (E)
O ::= + | – | ∗ | **div**

necessary, but we keep it to make the rule readable.

The structures of programs, declarations, commands, expressions, and operators are similarly specified. (Note that the Expression syntax domain is the set of arithmetic expressions that we have been studying.) No BNF rules exist for Identifier or Numeral, because these are collections of tokens. Figures 1.7 and 1.8 give the syntax definitions for an interactive file editor and a list processing language. (The **cr** token in Figure 1.7 represents the carriage return symbol.)

A good syntax definition lends a lot of help toward understanding the semantics of a language. Your experience with block-structured languages helps you recognize some familiar constructs in Figure 1.6. Of course, no semantic questions are answered by the syntax definition alone. If you are familiar with ALGOL60 and LISP, you may have noted a number of constructs in Figures 1.6 and 1.8 that have a variety of possible semantics.

1.2 MATHEMATICAL AND STRUCTURAL INDUCTION

Often we must show that all the members of a syntax domain have some property in common. The proof technique used on syntax domains is called *structural induction.* Before studying the general principle of structural induction, we first consider a specific case of it in the guise

Figure 1.7

Abstract syntax:

P∈ Program-session
S∈ Command-sequence
C∈ Command
R∈ Record
I ∈ Identifier

P ::= S **cr**
S ::= C **cr** S | **quit**
C ::= **newfile** | **open** I | **moveup** | **moveback** |
insert R | **delete** | **close**

Figure 1.8

Abstract syntax:

P∈ Program
E∈ Expression
L∈ List
A∈ Atom

P ::= E,P | **end**
E ::= A | L | **head** E | **tail** E | **let** A = E_1 **in** E_2
L ::= (A L) | ()

of *mathematical induction.* Mathematical induction is a proof strategy for showing that all the members of $\mathbb{N}$, the natural numbers, possess a property *P*. The strategy goes:

1. Show that 0 has *P*, that is, show that *P*(0) holds.
2. Assuming that an arbitrary member $i \in \mathbb{N}$ has *P*, show that $i+1$ has it as well; that is, show that *P*(*i*) implies *P*($i+1$).

If steps 1 and 2 are proved for a property *P*, then it follows that the property holds for all the numbers. (Why? Any number $k \in \mathbb{N}$ is exactly $(\cdots((0+1)+1)+\cdots+1)$, the 1 added *k* times. You take it from there.) Here is an application of mathematical induction:

1.1 Proposition:

For any $n \in \mathbb{N}$, *there exist exactly* $n!$ *permutations of n objects.*

Proof: We use mathematical induction to do the proof.

Basis: for 0 objects, there exists the ''empty'' permutation; since 0! equals 1, this case holds.

Induction: for $n \in \mathbb{N}$ assume that there exist $n!$ permutations of n objects. Now add a new object j to the n objects. For each permutation $k_{i1}, k_{i2}, \cdots, k_{in}$ of the existing objects, $n+1$ permutations result: they are $j, k_{i1}, k_{i2}, \cdots, k_{in}$; $k_{i1}, j, k_{i2}, \cdots, k_{in}$; $k_{i1}, k_{i2}, j, \cdots, k_{in}$; $k_{i1}, k_{i2}, \cdots, j, k_{in}$; and $k_{i1}, k_{i2}, \cdots, k_{in}, j$. Since there are $n!$ permutations of n objects, a total of $(n+1)*n! = (n+1)!$ permutations exist for $n+1$ objects. This completes the proof. □

The mathematical induction principle is simple because the natural numbers have a simple structure: a number is either 0 or a number incremented by 1. This structure can be formalized as a BNF rule:

N ::= 0 | N+1

Any natural number is just a derivation tree. The mathematical induction principle is a proof strategy for showing that all the trees built by the rule for N possess a property *P*. Step 1 says to show that the tree of depth zero, the leaf 0, has *P*. Step 2 says to use the fact that a tree *t* has property *P* to prove that the tree *t*+1 has *P*.

The mathematical induction principle can be generalized to work upon any syntax domain defined by a BNF rule. The generalized proof strategy is structural induction. Treating the members of a syntax domain *D* as trees, we show that all trees in *D* have property *P* inductively:

1. Show that all trees of depth zero have *P*.
2. Assume that for an arbitrary depth $m \geq 0$ all trees of depth *m* or less have *P*, and show that a tree of depth $m+1$ must have *P* as well.

This strategy is easily adapted to operate directly upon the BNF rule that generates the trees.

1.2 Definition:

The structural induction principle: for the syntax domain D and its BNF rule:
$d ::= Option_1 \mid Option_2 \mid \cdots \mid Option_n$
all members of D have a property P if the following holds for each $Option_i$, *for* $1 \leq i \leq n$: *if every occurrence of d in* $Option_i$ *has P, then* $Option_i$ *has P.*

The assumption ''every occurrence of *d* in $Option_i$ has *P*'' is called the *inductive hypothesis.* The method appears circular because it is necessary to assume that trees in *D* have *P* to prove that the *D*-tree built using $Option_i$ has *P*, but the tree being built must have a depth greater than the subtrees used to build it, so steps 1 and 2 apply.

1.3 Theorem:

The structural induction principle is valid.

Proof: Given a proposition *P*, assume that the claim ''if every occurrence of *d* in $Option_i$ has *P*, then $Option_i$ has *P*'' has been proved for each of the options in the BNF rule for *D*. But say that some tree *t* in D doesn't have property *P*. Then a contradiction results: pick the *D*-typed subtree in *t* of the least depth that does not have *P*. (There must always be one; it can be *t* if necessary. If there are two or more subtrees that are ''smallest,'' choose any one of them.) Call the chosen subtree *u*. Subtree *u* must have been built using some $Option_k$, and all of its proper *D*-typed subtrees have *P*. But the claim ''if every occurrence of *d* in $Option_k$ has *P*, then $Option_k$ has *P*'' holds. Therefore, *u* must have property *P*— a contradiction. □

Here are two examples of proofs by structural induction.

1.4 Example:

For the domain E: Expression and its BNF rule:

E ::= **zero** | E_1∗E_2 | (E)

show that all members of Expression have the same number of left parentheses as the number of right parentheses.

Proof: Consider each of the three options in the rule:

1. **zero**: this is trivial, as there are zero occurrences of both left and right parentheses.
2. E_1∗E_2: by the inductive hypothesis, E_1 has, say, *m* left parentheses and *m* right parentheses, and similarly E_2 has *n* left parentheses and *n* right parentheses. Then E_1∗E_2 has $m+n$ left parentheses and $m+n$ right parentheses.
3. (E): by the inductive hypothesis, E has *m* left parentheses and *m* right parentheses. Clearly, (E) has $m+1$ left parentheses and $m+1$ right parentheses. □

The structural induction principle generalizes to operate over a number of domains simultaneously. We can prove properties of two or more domains that are defined in terms of one another.

1.5 Example:

For BNF rules:

S ::= ∗E∗

E ::= +S | ∗∗

show that all S-values have an even number of occurrences of the ∗ token.

Proof: This result must be proved by a simultaneous induction on the rules for S and E, since they are mutually recursively defined. We prove the stronger claim that ''all

members of S *and* E have an even number of occurrences of ∗.'' For rule S, consider its only option: by the inductive hypothesis, the E tree has an even number of ∗, say, *m* of them. Then the ∗E∗ tree has *m*+2 of them, which is an even value. For rule E, the first option builds a tree that has an even number of ∗, because by the inductive hypothesis, the S tree has an even number, and no new ones are added. The second option has exactly two occurrences, which is an even number. □

SUGGESTED READINGS

Backus-Naur form: Aho & Ullman 1977; Barrett & Couch 1979; Cleaveland & Uzgalis 1977; Hopcroft & Ullman 1979; Naur et al. 1963

Abstract syntax: Barrett & Couch 1979; Goguen, Thatcher, Wagner, & Wright 1977; Gordon 1979; Henderson 1980; McCarthy 1963; Strachey 1966, 1968, 1973

Mathematical and structural induction: Bauer & Wossner 1982; Burstall 1969; Manna 1974; Manna & Waldinger 1985; Wand 1980

EXERCISES

1. a. Convert the specification of Figure 1.6 into the classic BNF format shown at the beginning of the chapter. Omit the rules for <Identifier> and <Numeral>. If the grammar is ambiguous, point out which BNF rules cause the problem, construct derivation trees that demonstrate the ambiguity, and revise the BNF definition into a nonambiguous form that defines the same language as the original.
   b. Repeat part a for the definitions in Figures 1.7 and 1.8.

2. Describe an algorithm that takes an abstract syntax definition (like the one in Figure 1.6) as input and generates as output a stream containing the legal sentences in the language defined by the definition. Why isn't ambiguity a problem?

3. Using the definition in Figure 1.5, write the abstract syntax forms of these expressions:

   a. 12
   b. (4 + 14) ∗ 3
   c. ((7 / 0))

   Repeat a-c for the definition in Figure 1.6; that is, draw the derivation trees.

4. Convert the language definition in Figure 1.6 into a definition in the format of Figure 1.5. What advantages does each format have over the other? Which of the two would be easier for a computer to handle?

5. Alter the BNF rule for the Command domain in Figure 1.6 to read:

   C ::= S | S;C
   S ::= I:=E | **if** E **then** C | **while** E **do** C | **begin** B **end**

Draw derivation trees for the old and new definitions of Command. What advantages does one form have over the other?

6. Using Strachey-style abstract syntax (like that in Figures 1.6 through 1.8), define the abstract syntax of the input language to a program that maintains a data base for a grocery store's inventory. An input program consists of a series of commands, one per line; the commands should specify actions for:
   a. Accessing an item in the inventory (perhaps by catalog number) to obtain statistics such as quantity, wholesale and selling prices, and so on.
   b. Updating statistical information about an item in the inventory
   c. Creating a new item in the inventory;
   d. Removing an item from the inventory;
   e. Generating reports concerning the items on hand and their statistics.

7. a. Prove that any sentence defined by the BNF rule in Example 1.4 has more occurrences of **zero** than occurrences of $*$.
   b. Attempt to prove that any sentence defined by the BNF rule in Example 1.4 has more occurrences of **zero** than of (. Where does the proof break down? Give a counterexample.

8. Prove that any program in the language in Figure 1.6 has the same number of **begin** tokens as the number of **end** tokens.

9. Formalize and prove the validity of simultaneous structural induction.

10. The principle of *transfinite induction* on the natural numbers is defined as follows: for a property P on $\mathbb{N}$, if for arbitrary $n \geq 0$, ((for all $m < n$, $P(m)$ holds) implies $P(n)$ holds), then for all $n \geq 0$, $P(n)$ holds.
    a. Prove that the principle of transfinite induction is valid.
    b. Find a property that is provable by transfinite induction and not by mathematical induction.

11. Both mathematical and transfinite induction can be generalized. A relation $\prec \subseteq D \times D$ is a *well-founded ordering* iff there exist no infinitely descending sequences in D, that is, no sequences of the form $d_n \succ d_{n-1} \succ d_{n-2} \succ \cdots$, where $\succ = \prec^{-1}$.
    a. The general form of mathematical induction operates over a pair $(D, \prec)$, where all the members of D form one sequence $d_0 \prec d_1 \prec d_2 \prec \cdots \prec d_i \prec \cdots$. (Thus $\prec$ is a well-founded ordering.)
       i. State the principle of generalized mathematical induction and prove that the principle is sound.
       ii. What is $\prec$ for $D = \mathbb{N}$?
       iii. Give an example of another set with a well-founded ordering to which generalized mathematical induction can apply.
    b. The general form of transfinite induction operates over a pair $(D, \prec)$, where $\prec$ is a

well founded ordering.

i. State the principle of general transfinite induction and prove it valid.
ii. Show that there exists a well-founded ordering on the words in a dictionary and give an example of a proof using them and general transfinite induction.
iii. Show that the principle of structural induction is justified by the principle of general transfinite induction.