8. Induction, Grammars, and Parsing

8.1 Introduction

This chapter presents the notion of grammar and related concepts, including how to use grammars to represent languages and patterns, after first discussing further the general idea of inductive definitions.

The general mathematical principle of inductive definition will be presented first. We then focus on the use of this principle embodied within the notion of *grammar*, an important concept for defining programming languages, data definition languages, and other sets of sequences. We devote some time to showing the correspondence between grammars and programs for parsing statements within a language.

8.2 Using Rules to Define Sets

In *Low-Level Functional Programming*, we used rules to define partial functions. A partial function can be considered to be just a *set* of source-target pairs, so in a way, we have a preview of the topic of this section, using rules to define sets. There are many reasons why this topic is of interest. In computation, the data elements of interest are generally members of some set, quite often an infinite one. While we do not usually *construct* this entire set explicitly in computation, it is important that we have a way of *defining* it. Without this, we would have no way to argue that a program intended to operate on members of such a set does so correctly. The general technique for defining such sets is embodied in the following principle.

Principle of Inductive Definition

A set S may be defined inductively by presenting

(i) A **basis**, *i.e.* a set of items asserted to be in S.

(ii) A set of **induction rules**, each of which produces, from a set of items known to be in S, one or more items asserted to be in S.

(iii) The **extremal clause**, which articulates that the only members of set S are those obtainable from the basis and applications of the induction rules.

For brevity, we usually avoid stating the extremal clause explicitly. However, it is always assumed to be operative.

The Set ω of Natural Numbers

Here is a very basic inductive definition, of the set of natural numbers, which will be designated ω :

> Basis: 0 is in ω .
>
> Induction rule: If *n* is in ω, then so is the number *n*+1.

Unfortunately, this example can be considered lacking, since we haven't given a precise definition of what n+1 means. For example, the definition would be satisfied in the case that n+1 happens to be the same as n. In that case, the set being defined inductively ends up being just {0}, not what we think of as the natural numbers.

We could give more substance to the preceding definition by defining +1 in the induction rule in a more elementary way. Following Halmos, for example, we could define the natural numbers purely in terms of sets. Intuitively, we think of a number n as being represented by a certain set with n items. The problem then is how to construct such a set.

There is only one set with 0 items, the empty set {}, so we will take that to represent the number 0. To build a set of one item, we could take that one item to be the empty set, so the first two numbers are:

0	is equated to	{}	
1	is equated to	{0}	*i.e.*, to {{}}

How do we get a set of 2 items? We can't just take two copies of 1, since according to the notion of a set, repetitions don't really count: {{}, {}} would be the same as {{}}, and our 2 would equal 1, not what we want. So instead, take the two elements to be 0 and 1, which we know to be distinct, since 0 is an empty set whereas 1 is a non-empty set:

> 2 is equated to {0, 1} *i.e.*, to {{}, {{}}}

Continuing in this way,

> 3 is equated to {0, 1, 2} *i.e.*, to {{}, {{}}, {{}, {{}}}}

How do we form *n*+1 given *n* in general? Since *n* is a set of "n" items, we can get a set of "n+1" items by forming a new set with the elements of *n* and adding in *n* as a new element. But the way to add in a single element is just to form the *union* of the original set with the set of that one element, i.e.

$$n + 1 \text{ is equated to } n \cup \{n\}$$

Since {n} is distinct from every element of n, this new set has one more element than n has.

The representation above is not the only way to construct the natural numbers. Another way, which gives more succinct representations, but one lacking the virtue that n is represented by a set of elements, is:

0	is equated to	{ }
n + 1	is equated to	{*n*}

For the same reason as before, 0 is distinct from 1. Similarly, n+1 is distinct from n in general, because at no stage of the construction is {n} ever a member of n. In this definition, numbers are characterized by the number of times an element can be extracted iteratively before arriving at the empty set, e.g.

{ }	0 times
{ { } }	1 time
{ { { } } }	2 times

and so on.

Finite, Infinite, and Countable Sets

Using the first definition of the natural numbers, we are able to give a more accurate definition of what is meant by "finite" and "infinite", terms that get used throughout the book:

> A set is **finite** if there is a one-to-one correspondence between it and one of the sets in ω using the first definition above (i.e. one of the sets of *n* elements for some *n*).
>
> A set that is not finite is called **infinite**.

The set of all natural numbers ω, is the simplest example of an infinite set. Here's how we know that it is infinite: For any element *n* of ω, there is no one-to-one correspondence between *n* and a subset of *n*. (This can be proved by induction). However, for itself, there are many such one-to-one correspondences. The following shows one such correspondence:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots \\ 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & \end{pmatrix}$$

the general rule being that *n* corresponds to 2(*n* + 1). Thus ω and the individual members of ω have substantially different properties.

A set such that there is a one-to-one correspondence between it and ω is called **countably-infinite**. A set is called **countable** if it is either finite or countably infinite. A set can be shown to be countable if there is a method for enumerating its elements. For example, the display above indicates that the even numbers are countable.

Example: The set of *all* subsets of ω is not countable.

For a justification of this assertion, please see the chapter *Limitations of Computing*.

The Set of All Finite Subsets of ω

Let fs(ω) stand for the set of all finite subsets of ω. Obviously fs(ω) is infinite. We can see this because there is a subset of it in one-to-one correspondence with ω, namely the set {{0}, {1}, {2}, {3}, ...}. To show fs(ω) is countable, we can give a method for enumerating its members. Here is one possible method: Let fs(n) represent the subsets of the set {0, 1, 2, ..., n-1}. An initial attempt at enumeration consists of a concatenation:

fs(0), fs(1), fs(2), fs(3),....

i.e.

fs(0) = {},
fs(1) = {}, {0},
fs(2) = {}, {0}, {1}, {0, 1},
fs(3) = {}, {0}, {1}, {2}, {0, 1}, {0, 2}, {1, 2}, {0, 1, 2},
...

There are some repetitions in this list, but when we drop the repeated elements, we will have the enumeration we seek:

{}, {0}, {1}, {0, 1}, {2}, {0, 2}, {1, 2}, {0, 1, 2}, ...

The Set Σ^* of Strings over a Given Finite Alphabet Σ

This is another example of a countabley-infinite set defined by induction. It is used very frequently in later sections.

Basis: The empty string λ is in Σ^*.

Induction rule: If x is in Σ^* and σ is in Σ, then the string σx is in Σ^*.

(where σx means symbol σ *followed by* symbols in x.)

As a special case, suppose that Σ = {0, 1}. Then elements of Σ^* could be introduced in the following order:

λ, by the basis
0, as $\lambda 0$, since $0 \in \Sigma$ and λ is in Σ^*
1, as $\lambda 1$, since $1 \in \Sigma$ and λ is in Σ^*
00, since $0 \in \Sigma$ and 0 is in Σ^*
10, since $1 \in \Sigma$ and 0 is in Σ^*
01, since $0 \in \Sigma$ and 1 is in Σ^*
11, since $1 \in \Sigma$ and 1 is in Σ^*
000, since $0 \in \Sigma$ and 00 is in Σ^*
100, since $1 \in \Sigma$ and 00 is in Σ^*
...

For reemphasis, note that the length of every string in Σ^* is finite, while Σ^* is a set with an infinite number of elements.

The Sets L_n of Strings of Length n over a Given Alphabet

Often we will want to define a set inductively using an indexed series of previously-defined sets. For example, L_0, L_1, L_2, ... could be a family of sets indexed by the natural numbers and the set we are defining inductively is their union.

Basis: L_0 is $\{\lambda\}$, the set consisting of just the empty string.

Induction rule: L_{n+1} is the set $\{\sigma x \mid \sigma \text{ is in } \Sigma \text{ and } x \text{ is in } L_n\}$

Another way to define Σ^* is then $\Sigma^* = L_0 \cup L_1 \cup L_2 \cup \ldots$.

The set of Hypercubes

A hypercube is a particular type of undirected graph structure that has recurrent uses in computer science. The hypercube of dimension n is designated as H_n

Basis: H_0 consists of just one point.

Induction rule: H_{n+1} consists of two copies of H_n, with lines connecting the corresponding points in each copy.

The figure below shows the how the first four hypercubes emerge from this definition.

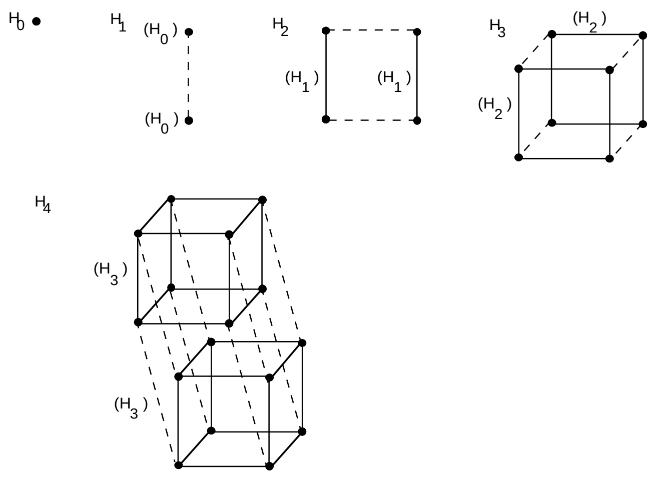

Figure 102: Hypercubes H_0 through H_4.

The Partial Recursive Functions (Advanced)

The partial recursive functions (PRF's) are those functions on the natural numbers that are defined inductively by the following definition. The importance of this set of functions is that it is hypothesized to be exactly those functions that are computable. This was discussed in *States and Transitions* as the Church/Turing Hypothesis.

Basis:

1. Every *constant* function (function having a fixed result value) is a PRF.

2. Every *projection,* a function having the form $\pi_i(x_1, ..., x_n) = x_i$ for fixed i and n, is a PRF.)

3. The *successor* function σ, defined by $\sigma(x) = x + 1$, is a PRF.

Induction rules:

3. Any *composition* of PRF's is a PRF, i.e. if *f* is an n-ary PRF, and $g_1, ..., g_n$ are n m-ary PRF's, then the m-ary function *h* defined by

$$h(x_1, ..., x_m) = f(g_1(x_1, ..., x_m), g_2(x_1, ..., x_m), ..., g_n(x_1, ..., x_m))$$

is also a PRF.

4. If f is a n-ary PRF and g is an (n+2)-ary PRF, then the (n+1)-ary function *h* defined by *primitive recursion,* i.e. one defined by following the pattern:

$h(0, x_1, ..., x_m) = f(x_1, ..., x_m)$

$h(y+1, x_1, ..., x_m) = g(y, h(y, x_1, ..., x_m), x_1, ..., x_m)$

is a PRF. Note that primitive recursion is just a way to provide definite iteration, as in the case of for-loops.

6. If f is a (n+1)-ary PRF, then the n-ary function g defined by the *μ operator*

$$g(x_1, ..., x_m) = \mu y[f(y, x_1, ..., x_m) = 0]$$

is a PRF. The right-hand side above is read "the least y such that $f(y, x_1, ..., x_m) = 0$". The meaning is as follows: $f(0, x_1, ..., x_m)$ is computed. If the result is 0, then the value of $\mu y[f(y, x_1, ..., x_m) = 0]$ is 0. Otherwise, $f(1, x_1, ..., x_m)$ is computed. If the result is 0, then the value of $\mu y[f(y, x_1, ..., x_m) = 0]$ is 1. If not, then $f(2, x_1, ..., x_m)$ is computed, etc. The value of $\mu y[f(y, x_1, ..., x_m) = 0]$ diverges if there is no value of y with the indicated property, or if any of the computations of $f(y, x_1, ..., x_m)$ diverge.

Notice that primitive recursion corresponds to *definite iteration* in programming languages, such as in a *for loop*, whereas the *μ* operator corresponds to *indefinite iteration* (as in a *while loop*).

Exercises

1 •• Using the first definition of natural numbers based on sets, give the set equivalents of numbers 4, 5, and 6.

2 •• Give a rex program for a function *nset* that displays the set equivalent of its natural number argument, using lists for sets. For example,

```
nset(0) ==> [ ]

nset(1) ==> [ [ ] ]

nset(2) ==> [ [ [ ] ], [ ] ]

nset(3) ==> [ [ [ [ ] ], [ ] ], [ [ ] ], [ ]  ]
```

etc.

3 ••• Let *fswor*(n) (*finite sets without repetition*) refer to the nth item in the list of all finite subsets of natural numbers presented above. Give a rex program that computes *fswor*, assuming that the sets are to be represented as lists.

4 ••• Construct rex rules that will generate the infinite list of finite sets of natural numbers without duplicates. The list might appear as

[[], [0], [1], [0, 1], [2], [0, 2], [1, 2], [0, 1, 2], ...]

5 •• Let $\omega - n$ mean the natural numbers beginning with n. Show that there is a one-to-one correspondence between ω and $\omega - n$ for each n.

6 ••• Show that no natural number (as a set) has a one-to-one correspondence with a subset of itself. (Hint: Use induction.)

7 ••• Construct a rex program that, with a finite set Σ as an argument, will generate the infinite list of all strings in Σ^***.**

8 •• Draw the hypercube H_5.

9 •• Prove by induction that the hypercube H_n has 2^n points.

10 ••• How many edges are in a hypercube H_n? Prove your answer by induction.

11 ••• A *sub-cube* of a hypercube is a hypercube of smaller or equal dimension that corresponds to selecting a set of points of the hypercube and the lines connecting them. For example, in the hypercube H_3, there are 6 sub-cubes of dimension 2, which correspond to the 6 faces as we usually view a 3-dimensional cube. However, in H_4, there are 18 sub-cubes of dimension 3, 12 that are easy to see and another 6 that are a little more subtle. Devise recursive rules for computing the number of sub-cubes of given dimension of a hypercube of given dimension.

12 ••• Refer to item 5 in the definition of partial recursive functions. Assuming that *f* and *g* are available as callable functions, develop both a flowchart and rex code for computing *h*.

13 ••• Refer to item 6 in the definition of partial recursive functions. Assuming that *f* is available as callable functions, develop both a flowchart and rex code for computing *h*.

14 ••• Show that each of the general recursive functions on natural numbers defined in *Low-Level Functional Programming* is also a partial recursive function.

8.3 Languages

An important use of inductive definition is to provide a definition of a **formal language**. Programming languages are examples of formal languages, but we shall see other uses as well. Let us consider a set of symbols Σ. We have already defined Σ* to be the set of all strings of symbols in Σ.

By a **language** over Σ, we just mean a subset of Σ*, in other words, any set of finite sequences with elements in Σ.

Here are a few examples of simple languages over various alphabets:

- For Σ = {1}, {1}* is the set of all strings of 1's. We have already encountered this set as one way to represent the set of natural numbers.
- For Σ = {1}, {λ, 11, 1111, 111111, ... } is the language of all even-length strings of 1's.
- For Σ = { '(', ')'}, {λ, (), ()(), (()), ()(()), (())(), (())(()), (()()), ((())), ...} is the language of all well-balanced parentheses strings.
- For any Σ, Σ* itself is a language.
- All finite sets of strings over a given set of symbols (which includes the empty set) are all languages.

When languages are reasonably complex, we have to resort to inductive definitions to define them explicitly.

The language of all well-balanced parenthesis strings

This language L is defined inductively as follows:

Basis: λ (the empty string) is in L.

Induction rules:

a. If *x* is in L, then so is the string (*x*).

b. If *x* and *y* are in L, then so is the string *xy*.

For example, we derive the fact that string (()(())) is in L:

1. λ is in L from the basis

2. () is in L, from 1 and rule a.

3. (()) is in L, from 2 and rule a.

4. (()(())) is in L, from 1, 2, and rule b.

Although we have presented languages as *sets* of strings, there is another way to use language-related concepts: to talk about *patterns.* Think of a set (of strings) as corresponding to a (possibly abstract) "pattern" that matches all strings in the set (and only those). For example, the pattern could be "every 0 in the string is immediately followed by two 1s". This corresponds to the language of all strings in which every 0 is immediately followed by two 1s. (Later on, we shall see how to use *regular expressions* as a means for concisely expressing such patterns.) In summary, there is a one-to-one correspondence between patterns and languages. The following section on grammars gives us another, more formal, tool for expressing languages, and therefore patterns.

8.4 Structure of Grammars

The form of inductive definition of languages used above occurs so often that special notation and nomenclature have been developed to present them. This leads to the concept of a *grammar*.

A **grammar** is a shorthand way of presenting an inductive definition for a language.

A grammar consists of the following four parts:

- The **terminal alphabet**, over which the language being defined is a language.
- The **auxiliary** or **non-terminal alphabet**, which has no symbols in common with the terminal alphabet. The symbols in the auxiliary alphabet provide a kind of "scaffolding" for construction of strings in the language of interest.
- The **start symbol**, which is always a member of the auxiliary alphabet.
- A finite set of **productions**. Each production is a rule that determines how one string of symbols can be rewritten as another.

It is common, but not universal, to write the productions using an arrow, so that production $x \rightarrow y$ means that *x can be rewritten as y*. Thus we have rewriting similar to rex rules, with the following exceptions:

With grammars, rewriting is purely string-based; there is no immediate identification of variables, no arithmetic, no lists, etc. Thus grammars are a lot more primitive than rex rules in this regard.

With grammars, the choice of rule is non-deterministic; we are not required to apply the *first* rule that matches; instead we can apply *any* rule that matches.

In order to define the language defined by a grammar, we begin with the strings generated by a grammar. If G denotes our grammar, then S(G) will denote this set of strings defined inductively:

Basis: The start symbol is in S(G).

Induction step: If u is in S(G) and u can be written as the concatenation vxw and there is a rule $x \rightarrow y$, then vyw is also in S(G). Whenever this relation holds, between strings vxw and vyw, we can write $vxw \Rightarrow vyw$.

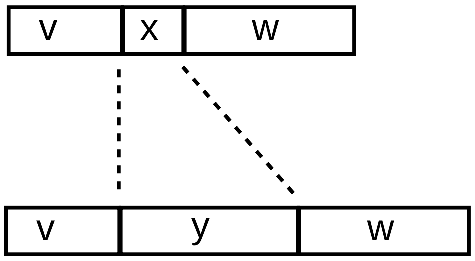

Figure 103: Application of a production x → y, in a string vxw to get string vyw

For example, if A → 0B10 is a rule, then the following are both true: 1AA ⇒ 10B10A, 1AA ⇒ 1A0B10. Furthermore, 10B10A ⇒ 0B100B10 and also 1A0B10 ⇒ 10B100B10.

The language L(G) defined by G is just that subset of S(G) the strings of that consist of only terminal symbols (*i.e.* no auxiliary symbols in the string).

The definition of → and ⇒ applies to grammars in which the left-hand side of → is an arbitrary string. However, the applications we will consider will mostly involve a left-hand side that is one-character long. Grammars with this property are called *context-free grammars*. When it is necessary to contrast with the fully general definition of grammar, the term *phrase-structure grammar* is sometimes used for the general case.

Grammar for Well-Balanced Parenthesis Strings

This grammar will exactly mimic the inductive definition for the parenthesis string language given earlier. The terminal alphabet has two symbols { '(', ')' }. The auxiliary alphabet has one symbol S. The start symbol is S. The productions are:

$S \to \lambda$	corresponding to the basis of the original inductive definition.
$S \to (S)$	corresponding to rule a. of the original inductive definition.
$S \to S\ S$	corresponding to rule b. of the original inductive definition.

Let us construct the derivation of (()(())) using the grammar notation. The top line shows the strings generated, while the bottom line shows the production used at each stepand the arrow points to the symbol being replaced.

```
S ⇒ ( S ) ⇒ ( S S ) ⇒ ( ( S ) S ) ⇒ ( () S ) ⇒ ( () ( S ) ) ⇒ ( () ( ( S ) ) ) ⇒ ( () ( ( ) ) )
↑     ↑       ↑           ↑              ↑            ↑                ↑
|     |       S → ( S )   S → λ          S → ( S )    S → ( S )        S → λ
|     S → S S
S → ( S )
```

Grammar for One-Operator Arithmetic Expressions

This grammar will generate arithmetic expressions of the form a, a+b, a+b+c, etc. The terminal alphabet is { a, b, c, + }. The auxiliary alphabet is {E, V}. The start symbol is E. The productions are:

```
E → V          // Every variable is an expression.
E → E + V      // Every string of the form E + V, where E is an expression
               //        and V is a variable, is also an expression.
V → a
V → b
V → c
```

Abbreviation for Alternatives: The | Symbol

The typical programming language definition includes hundreds of productions. To manage the complexity of such a definition, various abbreviation devices are often used. It is common to group productions by common left-hand sides and list the right-hand sides. One *meta-syntactic* device (i.e. device for dealing with the syntax of grammars, rather than the languages they describe) for doing this is to use the vertical bar | to represent alternatives. The bar binds less tightly than does juxtaposition of grammar symbols. Using this notation, the preceding grammar could be rewritten more compactly as:

E → V | E + V
V → a | b | c

Here the | symbol, like →, is part of the meta-syntax, not a symbol in the language.

We will adopt the practice of *quoting* symbols that are in the object language. With this convention, the grammar above becomes:

E → V | E '+' V

V → 'a' | 'b' | 'c'

Notice that, by repeated substitution for E, the productions for E effectively say the following:

E ⇒ V
E ⇒ E '+' V ⇒ V '+' V
E ⇒ E '+' V '+' V ⇒ V '+' V '+' V
E ⇒ E '+' V '+' V '+' V ⇒ V '+' V '+' V '+' V
...

In other words, from E we can derive V followed by any number of the combination '+' V. We introduce a special notation that replaces the two productions for E:

E → V { '+' V }

read "E produces V followed by *any number of* '+' V". Here the braces {....} represent the *any number of* operator. Later on, in conjunction with regular expressions, we will use (....)* instead of {....} and will call * the *star operator*.

By replacing two productions E → V | E '+' V with a single rule E → V { '+' V } we have contributed to an understanding of how the combination of two productions is actually used.

Other Syntactic Conventions in the Literature

Another related notation convention is to list the left-hand side followed by a colon as the head of a paragraph and the right-hand sides as a series of lines, one right-hand side per line. The preceding grammar in this notation would be:

E:
    V
    E '+' V

V: *one of*

'a' 'b' 'c'

A portion of a grammar for the Java language, using this notation, is as follows:

```
Assignment:
        LeftHandSide AssignmentOperator AssignmentExpression
LeftHandSide:
        Name
        FieldAccess
        ArrayAccess
AssignmentOperator: one of
        = *= /= %= += -= <<= >>= >>>= &= ^= |=
AssignmentExpression:
        ConditionalExpression
        Assignment
ConditionalExpression:
        ConditionalOrExpression
        ConditionalOrExpression ? Expression : ConditionalExpression
```

Grammar for S Expressions

S expressions (S stands for "symbolic") provide one way to represent arbitrarily-nested lists and are an important class of expressions used for data and language representation. S expressions are defined relative to a set of "atoms" A, which will not be further defined in the grammar itself. The start symbol is E. The productions are

$S \rightarrow A$	// Every atom by itself is an S expression.
$S \rightarrow (L)$	// Every parenthesized list is an S expression.
$L \rightarrow \lambda$	// The empty list is a list.
$L \rightarrow S\ L$	// An S expression followed by a list is a list.

An example of an S expression relative to common words as atoms is:

(This is (an S expression))

Later in this chapter we will see some of the uses of S expressions.

Grammar for Regular Expressions (Advanced)

Regular expressions are used to represent certain kinds of patterns in strings, or equivalently, to represent sets of strings. These are used to specify searches in text editors and similar software tools. Briefly, regular expressions are expressions formed using letters from a given alphabet of letters and the meta-syntactic operators of juxtaposition and | as already introduced. However, instead of using {...} to indicate iteration, the expression that would have been inside the braces is given a superscript asterisk and there is no more general form of recursion. The symbols 'λ' for the empty string and '∅' for the empty *set* are also allowable regular expressions. Each regular expression defines a language. For example,

(0 1)* | (1 0)*

defines a language consisting of the even-length strings with 0's and 1's strictly alternating. In the notation of this chapter, we could have represented the set as being generated by the grammar (with start symbol E)

$E \rightarrow \{ 0\ 1 \} \mid \{ 1\ 0 \}$

In the following grammar for regular expressions themselves, the start symbol is R. The productions are:

$R \rightarrow S \{ \text{'|'}\ S \}$

$R \rightarrow T \{ T \}$

$T \rightarrow U^{\text{'*'}}$

$U \rightarrow \text{'}\lambda\text{'}$

$U \rightarrow \text{'}\varnothing\text{'}$

$U \rightarrow \sigma$ (for each symbol σ in A)

$U \rightarrow \text{'('}\ R\ \text{')'}$

Above the symbol λ is quoted to indicate that we mean that λ is a *symbol* used in regular expressions, to distinguish it from the empty string. We will elaborate further on the use of regular expressions in the chapter *Finite-State Machines*. Meanwhile, here are a few more examples of regular expressions, and the reader may profit from the exercise of verifying that they are generated by the grammar:

(0*1 | 1*0)*

((000)*1)*

A Grammar for Grammar Rules (Advanced)

This example shows that grammars can be self-applied, that the given structure of a grammar rule is itself definable by a grammar. We take the set of auxiliary and terminal symbols to be given (not further defined by this grammar). The start symbol is *rule*. The rules are:

rule → lhs '→' rhs

lhs → auxiliary_symbol

rhs → symbol

rhs → rhs symbol

symbol → 'λ'
symbol → auxiliary_symbol
symbol → terminal_symbol

Note that in this grammar we have not included the meta-syntactic | and { } operators. Doing so is left to the reader. It is a simple matter of combining this and the previous example.

The grammars described above all had the property that the left-hand side of a rule consists of a single auxiliary symbol. As already mentioned, such a grammar is known as a **context-free grammar**. There are languages that are definable by rules with *strings* of grammars symbols on the left-hand side that are not definable by context-free grammars. The multiple-symbol capability for a general grammar allows arbitrary Turing machines to be simulated by grammar rules. This most general type of grammar is known as a **phrase-structure grammar**.

8.5 The Relationship of Grammars to Meaning

So far, our discussion of grammars has focused on the "syntactic" aspects, or "syntax", of a language, i.e. determining the member strings of the language. A second role of grammars bears a relation to the "semantics" of the language, i.e. determining meanings for strings. In particular, the *way* in which a string is derived using grammar rules is used by the compiler of a programming language to determine a meaning of strings, i.e. programs. The grammar itself does not provide that meaning, but the structure of derivations in the grammar allows a meaning to be assigned.

In the arithmetic expression example, we can think of a meaning being associated with each use of an auxiliary symbol:

The meaning of an E symbol is the value of an *expression*.

The meaning of a V symbol is the value of the corresponding *variable* (a, b, or c).

The production E → V suggests that the *value* of a single-variable expression is just the *value* of the variable.

The production E → E '+' V suggests that the *value* of an expression of the form E + V is derived from the sum of the *values* of the constituent E and V.

Derivation Trees

The easiest way to relate grammars to meanings is through the concept of **derivation tree**. A derivation tree is a tree having symbols of the grammar as labels on its nodes, such that:

If a node with parent labeled P has children labeled C_1, C_2, ..., C_n, then there is a production in the grammar P → C_1 C_2 ....C_n.

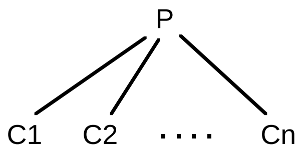

Figure 104: Depicting a production in a derivation tree

A **complete derivation tree** is one such that the root is labeled with the start symbol of the grammar and the leaves are labeled with terminal symbols.

Example

Consider again the grammar for additive arithmetic expressions:

E → V
E → E '+' V
V → a | b | c

Here *E* is the start symbol, and *a*, *b*, and *c* are terminals.

Below we show two derivation trees for this grammar, and a third tree that is not.

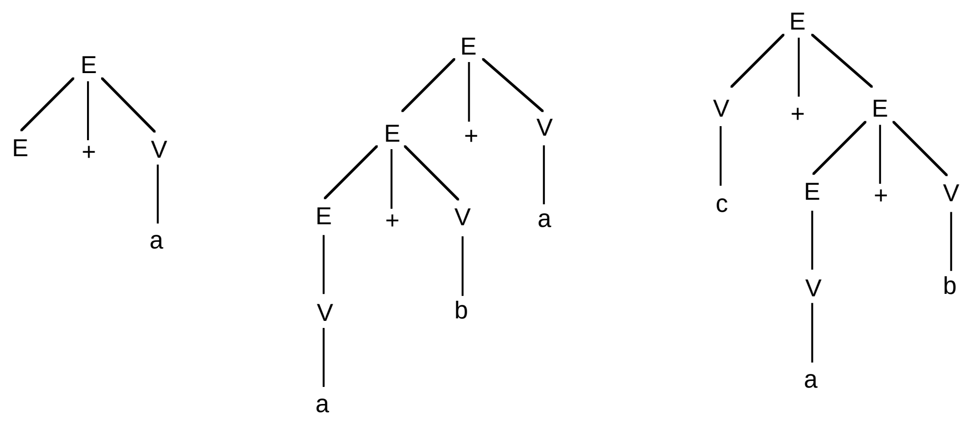

Figure 105: On the left is a derivation tree for the grammar above. In the middle is a complete derivation tree. Despite its similarity, the tree on the right is not a derivation tree for this grammar, since there is no production E → V '+' E.

Notice that the leaves of a complete derivation tree, when read left to right, give a string that is in the language generated by the grammar, for example a+b+a is the leaf sequence for the middle diagram. In this case, we say the derivation tree *derives* the string.

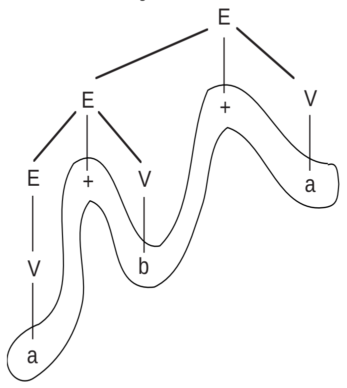

Figure 106: Deriving the string a + b + a in a complete derivation tree

A meaning to the strings in a grammar can be explained by associating a meaning to each of the nodes in the tree. The leaf nodes are given a meaning of some primitive value. For example, *a*, *b*, and *c* might mean numbers associated with those symbols as variables. The meaning of + might be the usual add operator. Referring to the tree above, the meaning of the V nodes is just the meaning of the symbol nodes below them. The meaning of the E nodes is either the sum of the nodes below, if there are three nodes, or the meaning of the single node below. So if *a*, *b*, and *c* had meanings 3, 5, and 17, we would show the meanings of all nodes as follows:

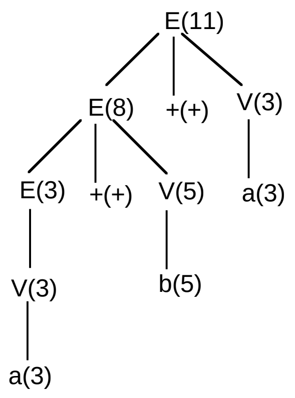

Figure 107: Complete derivation tree annotated with meanings (in parentheses) for each node

Processing a string in a language to determine its meaning, known as **parsing**, is essentially a process of constructing a derivation tree that derives the string. From that tree, a meaning can be determined. In some sense, parsing is like working backward against an inductive definition, to determine whether or not the proposed end result can be derived. Parsing is one of the tasks that programming language compilers have to perform.

Ambiguity

In order to convert a string into a derivation tree and then to a meaning, it is helpful for each string in the language to have a unique derivation tree. Otherwise there might be ambiguity in the meaning.

A grammar is called **ambiguous** if there is at least one string in the language the grammar generates that is derived by more than one tree.

Example – An ambiguous grammar

The grammar below will generate arithmetic expressions of the form a, a+b, a+b+c, a*b, a+b*c, etc. The terminal alphabet is { a, b, c, + }. The auxiliary alphabet is {E, V}. The start symbol is E. The productions are:

$E \rightarrow V$
$E \rightarrow E$ '+' E
$E \rightarrow E$ '*' E
$V \rightarrow a \mid b \mid c$

To see that the grammar is ambiguous, we present two different trees for the string a+b*c.

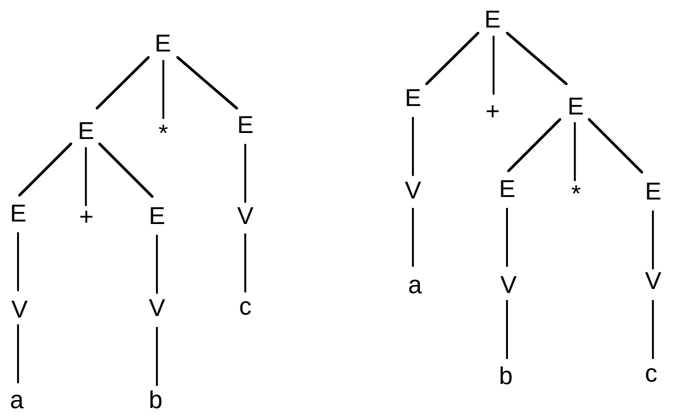

Figure 108: Two distinct derivation trees for a string in an ambiguous grammar

For example, if *a*, *b*, and *c* had meanings 3, 5, and 17, the tree on the left would give a meaning of 136 for the arithmetic expression, whereas the tree on the right would give a meaning of 88. Which meaning is correct? With an ambiguous grammar, we can't really say. But we can say that in common usage, a+b*c corresponds to a+(b*c) (i.e. * *takes precedence* over +), and not to (a+b)*c. The expression usually regarded as correct corresponds to the right-hand derivation tree.

Although the problem of ambiguity is sometimes resolved by understandings about which rule should be given priority in a derivation when there is a choice, it is perhaps most cleanly resolved by finding another grammar that generates the same language, but one that is not ambiguous. This is usually done by finding a different set of productions that does not cause ambiguity. For the present language, a set of productions that will do is:

An unambiguous grammar for simple arithmetic expressions

$$
\begin{aligned}
&E \rightarrow T\\
&E \rightarrow E\ \text{'+'}\ T\\
&T \rightarrow V\\
&T \rightarrow T\ \text{'*'}\ V\\
&V \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'}
\end{aligned}
$$

This implements * taking precedence over +.

Here we have added a new auxiliary T (for "term"). The idea is that value of an expression (an E) is either that of a single term or that of an expression plus a term. On the other hand, the value of a term is either that of a variable or a term times a variable. There is now only one leftmost derivation of each string. For example, the derivation of a+b*c is

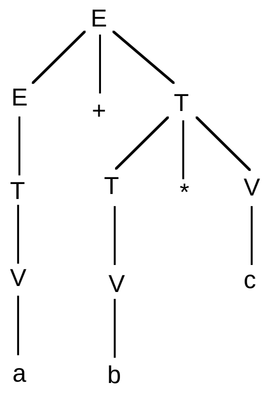

Figure 109: Derivation of a+b*c in an unambiguous grammar

In essence, the new grammar works by enforcing precedence: It prohibits us from expanding an expression (i.e. an E-derived string) using the * operator. We can only expand a term (i.e. a T-derived string) using the * operator. We can expand expressions using the + operator. But once we have applied the production, we can only expand the term, i.e. only use the * operator, not the + operator. Thus the new grammar has a *stratifying* effect on the operators.

The new grammar coincidentally enforces a left-to-right grouping of sub-expressions. That is, a + b + c is effectively grouped as if (a + b) + c, rather than a + (b + c). To see this, note that the derivation tree for this expression is:

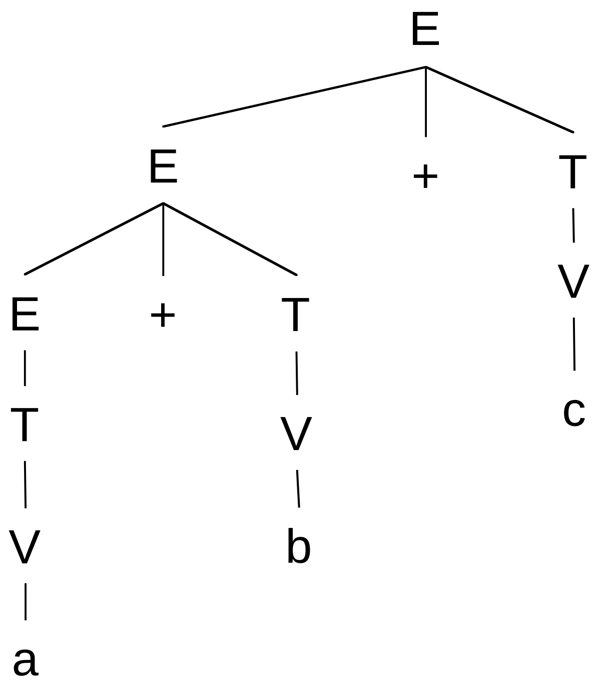

Figure 110: Derivation tree showing grouping

Exercises

1 •• Try to find another derivation tree for a+b*c in the unambiguous grammar. Convince yourself it is not possible.

2 •• Show that the grammar for regular expressions given earlier is ambiguous.

3 •• Give an unambiguous grammar for regular expressions, assuming that * takes precedence over juxtaposition and juxtaposition takes precedence over |.

4 ••• Determine, enough to convince yourself, whether the given grammar for S expressions is ambiguous.

5 •• The unambiguous grammar above is limited in that all the expressions derived within it are *two-level* + and *. That is, we can only get expressions of the form:

V*V*...V + V*V*...V + ... + V*V*...V

consisting of an overall sum of terms, each consisting of a product of variables. If we want more complex structures, we need to introduce parentheses:

(a + b)*c + d

for example. Add the parenthesis symbols to the unambiguous grammar and add a new production for including parentheses that represents this nuance and leaves the grammar unambiguous.

Abstract Syntax Trees

A tree related to the derivation tree is called the **abstract-syntax tree**. This tree is an abstract representation of the *meaning* of the derived expression. In the abstract-syntax tree, the non-leaf nodes of the tree are labeled with *constructors* rather than with auxiliary symbols from the grammar. These constructors give an indication of the meaning of the string derived.

In the particular language being discussed, it is understood that an expression is a summation-like idea, and the items being summed can be products, so an abstract syntax tree would be as follows:

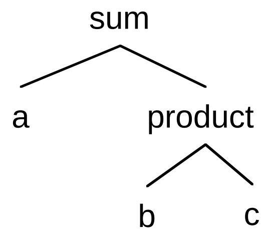

Figure 111: Abstract syntax tree for a+b*c

Note that abstract syntax does not show the specific operators, such as + and *; it only shows the type of entity and the constituent parts of each such entity. Using abstract syntax, we can appreciate the *structure* of a language apart from its "concrete" representation as a set of strings.

An abstract syntax tree could be represented as the result of applying *constructors*, each of which makes a tree out of component parts that are subtrees. For the language discussed in preceding examples, the constructors could be as follows (where *mk* is an abbreviation for *make*):

mk_sum
mk_product
mk_variable

Equivalent to the abstract syntax tree is an expression involving the constructors. For the tree above, this expression would be:

```
mk_sum(mk_variable('a'),
       mk_product(mk_variable('b'),
                  mk_variable('c')))
```

The final meaning of the overall expression can now be derived by simply evaluating the constructor functions appropriately redefined. For example, if we wanted to compute an arithmetic value, we would use the following definitions for the constructors:

```
mk_sum(X, Y) => X + Y;
mk_product(X, Y) => X * Y;
mk_variable(V) => .... code to look up V's value in a symbol table....
```

On the other hand, if our objective were to generate machine language code, we would use functions that produce a code list. If we are just interested in verifying that our parser works, we could use functions that return a description of the abstract syntax of the expression, such as:

```
mk_sum(X, Y) => [“sum”, X, Y];
mk_product(X, Y) => [“product”, X, Y];
mk_variable(V) => [“fetch”, V];
```

One of the nice things about the abstract syntax idea is that we can leave these functions unspecified until we decide on the type of output desired.

Abstract Grammars (Advanced)

Corresponding to abstract syntax, we could invent the notion of an "**abstract grammar**". Again, the purpose of this would be to emphasize structure over particular character-based representations. For the arithmetic expression grammar, the abstract grammar would be:

E → V
E → E + E
E → E * E

Abstract grammars are frequently left ambiguous, as the above grammar is, with an understanding that the ambiguity can be removed with appropriate manipulation to restore precedence. The purpose of such abstract grammars is to convey the general structure of a programming language as succinctly as possible and issues such as precedence just get in the way.

Example

Assume that mailing lists are represented the following way: There is a list of all mailing lists. Each list begins with the name of the list, followed by the people in the list. (Assume that lists can't appear on lists.) So, for example, if the lists were "students", "faculty", and "staff", the overall list of lists might be:

```
[["students", "Joe", "Heather", "Phil"],
 ["faculty", "Patti", "Sam", "John", "Cathy"],
 ["staff", "Phil", "Cathy", "Sheldon"]]
```

Suppose we want to specify a grammar for the correct construction of such an overall list of lists (with an arbitrary number of lists and list members). There are two types of answer: An ordinary grammar would require that one specify each symbol, including brackets, commas, and quote marks. An *abstract grammar* (corresponding to *abstract syntax*) gives just the list *structure* without the specific punctuation marks.

An abstract grammar for this problem, with S as the start symbol, is:

```
S → { M }            // directory of zero or more mailing lists

M → N { N }          // list name followed by people names

N → A { A }          // a list or person name

A → 'a' | 'b' | .... | 'Z'
```

An ordinary grammar, with S as the start symbol, is:

```
S → '[' L ']'

L →> λ | M { ',' M }   // directory of zero or more mailing lists

M → '[' N { ',' N } ']'   // list name followed by people names

N → '"' A { A } '"'     // one or more letters in quotes symbols
```

A → 'a' | 'b' | .... | 'Z'

Additional Meta-Syntactic Forms

We mentioned in the previous section how the symbols | and * can be used to fold several rules together into one. There are several different symbols that are commonly used for informal and formal communication about patterns.

The Brackets-as-Optional Convention:

In many descriptions of patterns, brackets around text means that the text is *optional*. If we include this convention in grammar rules, for example, then

N → ['+'] D

reads: "An N produces an optional + followed by a D." This can be taken as an abbreviation for two separate productions, one that includes the + and one that does not.

N → '+' D
N → D

For example, if we are describing numerals with an optional + or - sign, we could use brackets in combination with |:

Numeral → ['+' | '-'] Digits

This convention is used to describe optional command-line arguments for UNIX™ programs. For example, typing 'man awk' produces for the synopsis something like:

`awk [ -f` *program-file*`] [`*program*`] [`*parameters*`] [`*filename*`]`

meaning that there are four optional command-line arguments: a program-file (preceded by `-f`), a literal program, some parameters, and a filename. (Presumably there won't be both a program-file and a program, but the specification does not preclude it.)

The Braces-as-Alternatives Convention:

In other descriptions of patterns, people use braces to represent a number of different alternatives rather than *any number of* as we have been doing. For example, if Bob, Josh, and Mike all have email addresses at cs.hmc.edu, then one might give their addresses collectively as

```
{bob, josh, mike}@cs.hmc.edu
```

rather than spelling them out:

```
bob@cs.hmc.edu, josh@cs.hmc.edu, mike@cs.hmc.edu.
```

Obviously this convention is redundant in grammars if both | and [...] conventions are operative. One also sometimes sees the same effect using braces or brackets, with the alternatives stacked vertically:

$$\left\{\begin{array}{l}\texttt{bob}\\\texttt{josh}\\\texttt{mike}\end{array}\right\}\ \texttt{@cs.hmc.edu}$$

The Ellipsis Convention:

This is another way to represent iteration:

(D ...)

means 0 or more D's.

8.6 Syntax Diagrams

The graphical notation of "syntax diagrams" is sometimes used in place of grammars to give a better global understanding of the strings a grammar defines. In a syntax diagram, each left-hand side auxiliary symbol has a graph associated with it representing a combination of productions. The graph consists of arrows leading from an incoming side to an outgoing side. In between, the arrows direct through terminal and auxiliary symbols. The idea is that all strings generated by a particular symbol in the grammar can be obtained by following the arrows from the incoming side to the outgoing side. In turn, the auxiliary symbols that are traversed in this process are replaced by traversals of the graphs with those symbols as left-hand sides. A recursive traversal of the graph corresponding to the start symbol corresponds to a string in the language.

A syntax diagram equivalent to the previous grammars is shown below. Syntax diagrams are sometimes preferred because they are intuitively more readable than the corresponding grammar, and because they enable one to get a better overview of the connectivity of concepts in the grammar. Part of the reason for this is that recursive rules in the grammar can often be replaced with cyclic structures in the diagram.

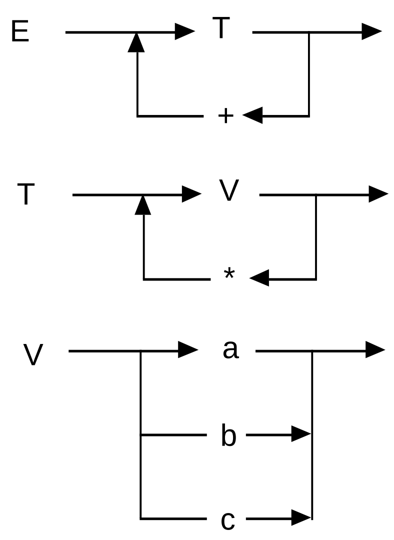

Figure 112: Syntax diagram for a simple language

Derivation by syntax diagram is similar to derivation in a grammar. Beginning with the graph for the start symbol, we trace out a path from ingoing to outgoing. When there is a "fork" in the diagram, we have a *choice* of going either direction. For example, in the diagram above, E is the start symbol. We can derive from E the auxiliary T by taking the upper choice, or derive T + ... by taking the lower choice. Thus we can derive any number of T's, separated by +'s. Similarly, any T derives any number of V's separated by *'s. Finally a V can derive only one of a, b, or c. So to generate a+b*c, we would have the following steps:

E	
T+T	looping back once through the E graph
V+T	going straight through the T graph
V+V*V	looping back once through the T graph
a+V*V	
a+b*V using the V graph three times, each with a different choice	
a+b*c	

A syntax diagram for S expressions over an unspecified set of atoms would be:

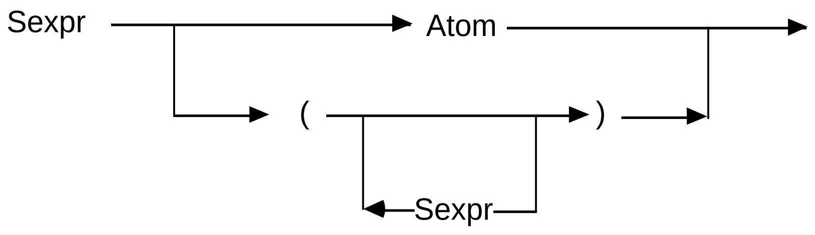

Figure 113: S expression syntax diagram

Actually, there is another more general notion of S expressions, one that uses a period symbol '.' in a manner analogous to the vertical bar in rex list expressions. For example, (A B (C D) . E) corresponds to [A, B, [C, D] | E]. This diagram for this generalized form is:

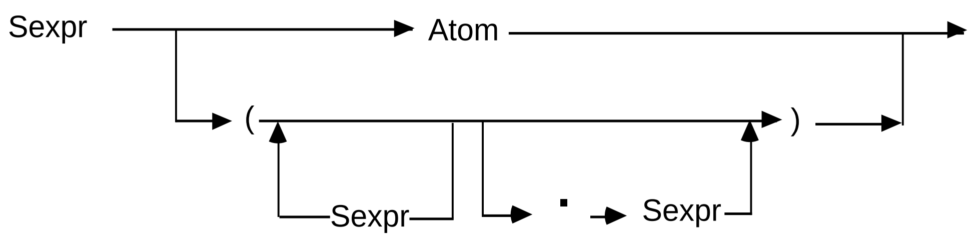

Figure 114: Generalized S expression syntax diagram

Exercises

1 •• Give a syntax diagram for the set of well-balanced parenthesis strings.

2 •• Give a syntax diagram, then a grammar, for the set of signed integers, i.e. the set of strings consisting of the any number of the digits {0, 1, ..., 9} preceded by an optional + or -.

3 •• Give a grammar corresponding to the generalized S expression syntax given in the diagram above.

4 ••• Give a syntax diagram, then a grammar, for the set of floating-point numerals, i.e. the set of strings consisting of an optional + or -, followed by an optional whole number part, followed by an optional decimal point, an optional fraction, and optionally the symbol 'e' followed by an optional signed exponent. The additional constraint is that, with all the options, we should not end up with numerals having *neither* a whole number part nor a fraction.

5 ••• Give a syntax diagram for terms representing lists as they appear in rex. Include brackets, commas, and the vertical bar as terminal symbols, but use A to represent atomic list elements.

6 ••• Locate a complete copy of the Java grammar and construct syntax diagrams for it.

8.7 Grouping and Precedence

We showed earlier how the structure of productions has an effect on the precedence determination of operators. To recapitulate, we can insure that an operator * has higher precedence than + by structuring the productions so that the + is "closer" to the start symbol than *. In our running example, using the * operator, E is the start symbol:

$$
\begin{aligned}
&E \rightarrow T \ \{ \ '+' \ T \ \} \\
&T \rightarrow V \ \{ \ '*' \ V\} \\
&V \rightarrow 'a' \mid 'b' \mid 'c'
\end{aligned}
$$

Generally, the lower precedence operators will be "closer" to the start symbol.

For example, if we wanted to add another operator, say ^ meaning "raise to a power", and wanted this symbol to have higher precedence than *, we would add ^ "farther away" from the start symbol than '*' is, introducing a new auxiliary symbol P that replaces V in the existing grammar, and adding productions for P:

$$
\begin{aligned}
&E \rightarrow T \ \{ \ '+' \ T \ \} \\
&T \rightarrow P \ \{ \ '*' \ P \ \} \\
&P \rightarrow V \ \{ \ '\wedge' \ V \ \} \\
&V \rightarrow 'a' \mid 'b' \mid 'c'
\end{aligned}
$$

Another issue that needs to be addressed is that of **grouping**: how multiple operators at the same level of precedence are grouped. This is often called "associativity", but this term is slightly misleading, because the discussion applies to operators that are not necessarily associative. For example, an expression

a + b + c

could be interpreted with left grouping:

(a + b) + c

or with right grouping:

a + (b + c)

You might think this doesn't matter, as everyone "knows" + is associative. However, it does matter for the following sorts of reasons:

- In floating point arithmetic, + and * are not associative.

- It is common to group related operators such as + and - together as having the same precedence. Then in a mixed statement such as

  a - b + c

  grouping matters very strongly: (a - b) + c is not the same as a - (b + c). When these operators are arithmetic, left-grouping is the norm.

Let's see how grouping can be brought out in the structure of productions. For an expression that is essentially a sequence of terms separated by '+' signs, there are two possible production sets:

	E → T *vs.*	E → T
	E → E '+' T	E → T '+' E
or		or
	E → T { '+' T }	E → { T '+' } T
	left grouping	*right grouping*

To see that the production set on the left gives left-grouping, while the one on the right gives right-grouping, compare the forms of the respective derivation trees:

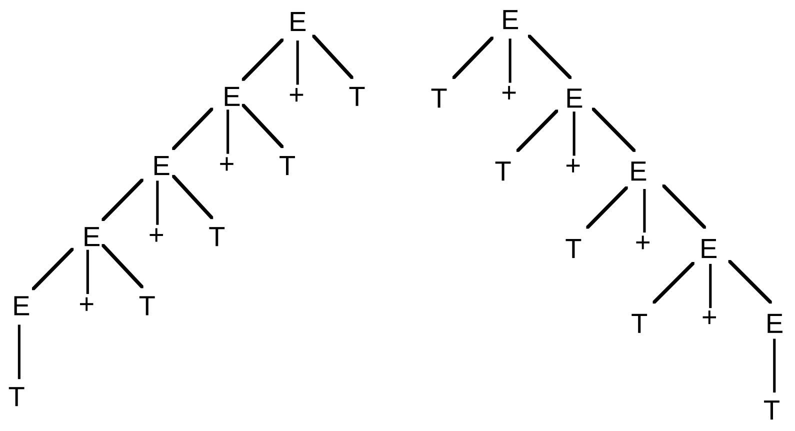

Figure 115: Left and right grouping shown by derivation trees

The *interpretations* of these two trees are:

((((T + T) + T) + T) + T)   vs.   (T + (T + (T + (T + T))))

respectively. [Do not confuse the parentheses in the above groupings with the meta-syntactic parentheses in the grammar. They are, in some sense, opposite!]

8.8 Programs for Parsing

One function that computer programs have to perform is the interpretation of expressions in a language. One of the first aspects of such interpretation is parsing the expression, to determine a meaning. As suggested earlier, parsing effectively constructs the derivation tree for the expression being parsed. But parsing also involves the rejection of ill-formed input, that is, a string that is not in the language defined by the grammar. In many ways, this error detection, and the attendant recovery (rather than just aborting when the first error is detected in the input) is the more complex part of the problem.

We will describe one way to structure parsing programs using the grammar as a guideline. This principle is known as "**recursive-descent**" because it begins by trying to match the start symbol, and in turn calls on functions to match other symbols recursively. Ideally, the matching is done by scanning the input string left-to-right without backing up. We will give examples, first using rex rules that correspond closely to the grammar rules, then using Java methods. Before we can do this, however, we must address a remaining issue with the structure of our productions. The production

$$E \rightarrow E \text{ '+' } T$$

has an undesirable property with respect to left-to-right scanning. This property is known as "**left recursion**". In the recursive descent method, if we set out to parse an E, we will be immediately called upon to parse an E before any symbols in the input have been matched. This is undesirable, as it amounts to an infinite loop. The production

$$E \rightarrow T \text{ '+' } E$$

does not share this problem. With it, we will not try for another E until we have scanned a T and a '+'. However, as we observed, this production changes the meaning to right-grouping, something we don't want to do.

To solve this problem, we already observed that the net effect of the two left-grouping productions can be cast as

$$E \rightarrow T \ \{ \text{ '+' } T \ \}$$

To represent this effect in a recursive language such as rex, *without* left recursion, we change the productions so that the first T is produced first, then as many of the combination ('+' T) are produced as are needed. We add a new symbol C that stands for "continuation". The new productions are:

$$\begin{array}{l} E \rightarrow T\ C \\ C \rightarrow \text{ '+' } T\ C \\ C \rightarrow \lambda \end{array}$$

We can see that this is correct, since the last two productions are clearly equivalent to

$$C \rightarrow \ \{ \text{ '+' } T \ \}$$

and when we substitute for C in $E \rightarrow T\ C$ we get

$$E \rightarrow T \ \{ \text{ '+' } T \ \}$$

as desired. Informally, under recursive descent, these productions say: "To parse an E, first parse a T followed by a C. To parse a C, if the next symbol is '+', parse another T, then another C. Otherwise return." Evidently, these productions do not have the left-recursion problem. The rules only "recurse" as long as there is another '+' to parse.

An interesting observation in connection with the above rules is that they correspond naturally to a certain while loop:

```
to parse E:
              parse T;
              while( next char is '+' )
                     parse T;
```

We will see this connection in action as we look at a parser written in Java.

A Language Parser in Java

We are going to give a Java parser for an arithmetic expression grammar with two operators, '+' and '*', with '*' taking precedence over '+'. This parser simply checks the input for being in the language. Then we will give one that also computes the corresponding arithmetic value.

```
A -> M { '+' M }            sum

M -> V { '*' V }            product

V -> a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
```

To each auxiliary symbol there corresponds a Java method responsible for determining whether the remainder of the input can be generated by that symbol.

```
class addMult extends LineBuffer                // addMult parser class
  {
  // Parse method for A -> M { '+' M }

  Object A()
    {
    Object result;
    Object M1 = M();                            // get the first addend
    if( isFailure(M1) ) return failure;

    result = M1;

    while( peek() == '+' )                      // while more '+'
      {
      nextChar();                               // absorb the '+'
      Object M2 = M();                          // get the next addend
      if( isFailure(M2) ) return failure;
      result = mkTree("+", result, M2);         // create tree
      }
    return result;
    }
  // Parse method for M -> V { '*' V }

  Object M()
    {
    Object result;
    Object V1 = V();                            // get the first variable
    if( isFailure(V1) ) return failure;

    result = V1;

    while( peek() == '*' )                      // while more '*'
      {
      nextChar();                               // absorb the '*'
      Object V2 = V();                          // get the next variable
      if( isFailure(V2) ) return failure;
      result = mkTree("*", result, V2);         // create tree
      }
    return result;
    }


// Parse method for V -> a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

  Object V()
    {
    skipWhitespace();
    if( isVar(peek()) )                         // if there is a variable
      {
      return (new StringBuffer(1).append(nextChar())).toString();
      }
    return failure;
    }

  Object parse()        // TOP LEVEL: parse with check for residual input
    {
    Object result = A();                        // parse an A
    skipWhitespace();                           // ignore trailing whitespace
    if( position < lastPosition )               // see if any residual junk
      {
      System.out.print("*** Residual characters after input: ");
      while( !eof )                             // print residual characters
        {
        char c = nextChar();
        System.out.print(c);
        }
      System.out.println();
      }
    return result;                              // return result of parse
    }


  // isVar indicates whether its argument is a variable

  boolean isVar(char c)
    {
    switch( c )
      {
      case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':
      case 'h': case 'i': case 'j': case 'k': case 'l': case 'm': case 'n':
      case 'o': case 'p': case 'q': case 'r': case 's': case 't': case 'u':
      case 'v': case 'w': case 'x': case 'y': case 'z':
        return true;
      default:
        return false;
      }
    }

  // SUPPORT CODE

  static String promptString = "input: ";              // prompt string

  static ParseFailure failure = new ParseFailure();     // failure

  addMult(String input)                                 // parser constructor
    {
    super(input);                                       // construct LineBuffer
    }

  boolean isFailure(Object ob)                          // test for failure
    {
    return ob instanceof ParseFailure;

  static boolean prompt()                               // prompter
    {
    System.out.print(promptString);
    System.out.flush();
    return true;
    }
  }  // class addMult


// ParseFailure object is used to indicate a parse failure.

class ParseFailure
  {
  }


// LineBuffer provides a way of getting characters from a string
// Note that eof is a variable indicating end-of-line. It should
// not be confused with the eof method of LineBufferInputStream.

class LineBuffer
  {
  String input;                 // string being parsed
  boolean eof;                  // end-of-file condition exists

  int position;                 // position of current character

  int lastPosition;             // last position in input string

  LineBuffer(String input)
    {
    this.input = input;         // initialize variables dealing with string
    position = -1;              // initialize "pointer"
    lastPosition = input.length()-1;
    eof = lastPosition == -1;
    }

  char nextChar()                     // get next character in input
    {
    if( position >= lastPosition )    // if no more characters
      {
      eof = true;                     // set eof
      return ' ';                     // and return a space
      }
    return input.charAt(++position);
    }

  void skipWhitespace()               // skip whitespace
```

```
    {
    while( !eof && peek() == ' ' )
      {
      nextChar();
      }
    }
  }  // LineBuffer


// LineBufferInputStream is an input stream capable of reading one line
// at a time and returning the line as a string, by calling method getLine().
// It also provides the method eof() for testing for end-of-file.
// It extends PushbackInputStream from package java.io

class LineBufferInputStream extends PushbackInputStream
  {
  /**
    * LineBufferInputStream constructs from an InputStream
    */

  LineBufferInputStream(InputStream in)
    {
    super(in);               // call constructor of PushbackInputStream
    }

  /**
    *  getLine() gets the next line of input
    */

  String getLine()
    {
    StringBuffer b = new StringBuffer();         // buffer for line
    try
      {
      int c;
      while( !eof() && ((c = read()) != '\n') )  // read to end-of-line
        {
        b.append((char)c);                        // input.charAt(++position);
        }
      return b.toString();                       // get string from buffer
      }
    catch( java.io.IOException e )
      {
      handleException("getLine", e);
      return "";
      }
    }

  /**
    *  eof() tells whether end-of-file has been reached.
    */

  public boolean eof()
    {
    try
      {
      int c = read();
      if( c == -1 )
        {
        return true;
        }
      else
        {
        unread(c);
        return false;
        }
      }
    catch( IOException e )
      {
      handleException("eof", e);
      }
    return false;
    }

  /**
    *  handleException is called when there is an IOException in any method.
    */

  public void handleException(String method, IOException e)
    {
    System.out.println("IOException in LineBufferInputStream: " + e +
                       " calling method " + method);
    }
  }
```

Extending the Parser to a Calculator

The following program extends the parse to calculate the value of an input arithmetic expression. This is accomplished by converting each numeral to a number and having the parse methods return a number value, rather than simply an indication of success or failure. We also try to make the example a little more interesting by adding the nuance of parentheses for grouping. This results in mutually-recursive productions, which translate into mutually-recursive parse methods. The grammar used here is:

```
A -> M { '+' M }             sum

M -> U { '*' U }             product

U -> '(' A ')' | N           parenthesized expression or numeral

N -> D {D}                   numeral

D -> 0|1|2|3|4|5|6|7|8|9     digit
```

We do not repeat the definition of class `LineBuffer`, which was given in the previous example.

```
class SimpleCalc extends LineBuffer              // SimpleCalc parser class
  {
  static public void main(String arg[])          // USER INTERFACE
    {
    LineBufferInputStream in = new LineBufferInputStream(System.in);

    while( prompt() && !in.eof() )               // while more input
      {
      String input = in.getLine();               // get line of input
      SimpleCalc parser = new SimpleCalc(input); // create parser
      Object result = parser.parse();            // use parser
      if( result instanceof ParseFailure )       // show result
        System.out.println("*** syntax error ***");
      else
        System.out.println("result: " + result);
      System.out.println();
      }
    System.out.println();
    }


  Object A()                            // PARSE FUNCTION for A -> M { '+' M }
    {
    Object result = M();                // get first addend
    if( isFailure(result) ) return failure;

    while( peek() == '+' )                      // while more addends
      {
      nextChar();
      Object M2 = M();                          // get next addend
      if( isFailure(M2) ) return failure;
      try
        {
        result = Poly.Arith.add(result, M2);    // accumulate result
        }
      catch( argumentTypeException e )
        {
        System.err.println("internal error: argumentTypeException caught");
        }
      }
    return result;
    }


  Object M()                            // PARSE FUNCTION for M -> U { '*' U }
    {
    Object result = U();                // get first factor
    if( isFailure(result) ) return failure;

    while( peek() == '*' )              // while more factors
      {
      nextChar();
      Object U2 = U();                  // get next factor
      if( isFailure(U2) ) return failure;
      try
        {
        result = Poly.Arith.multiply(result, U2);       // accumulate result
        }
      catch( argumentTypeException e )
        {
        System.err.println("internal error: argumentTypeException caught");
        }
      }
    return result;
    }
  Object U()                    // PARSE FUNCTION for U -> '(' A ')' | N
    {
    if( peek() == '(' )         // Do we have a parenthesized expression?
      {
      nextChar();
      Object A1 = A();          // Get what's inside parens
      if( peek() == ')' )
        {
        nextChar();
        return A1;
        }
      return failure;
      }

    return N();                 // Try for numeral
    }


  Object parse()        // TOP LEVEL: parse with check for residual input
    {
    Object result = A();
    if( position < lastPosition )
      {
      return failure;
      }
    return result;
    }


  static String promptString = "input: ";               // prompt string

  static ParseFailure failure = new ParseFailure();     // failure

  SimpleCalc(String input)               // constructor for parser
    {
    super(input);                        // construct LineBuffer
    }


  boolean isFailure(Object ob)          // test for failure
    {
    return ob instanceof ParseFailure;
    }


  static boolean prompt()
    {
    System.out.print(promptString);
    System.out.flush();
    return true;
    }
  }  // class SimpleCalc
```

8.9 More on Expression Syntax vs. Semantics

The syntax of expressions in programming languages varies widely from language to language. Less variable is the "semantics" of expressions, that is, the meaning of expressions. In this note, we discuss some of the different syntaxes both the programmer and the user might encounter. It is important for the student to get used to thinking in terms of semantics apart from specific syntax. In other words, we should try to "see through" the expressions to the conceptual meaning. Acquiring this knack will make the programming world less of a blur.

For our initial comparisons, we shall confine ourselves to arithmetic expressions. We will later discuss more general forms. In arithmetic cases, the **meaning** of an expression is defined as follows:

> Given any assignment of values to each of the variables in an expression, a value is determined by "evaluating" the expression.

In the chapter on Information Structures, we described the idea of bindings. Here we are going to become more precise and make additional use of this idea.

Definition: A **binding** (also called **assignment**) for an expression is a function from each of its variable symbols to a value. [Please do not confuse assignment here with the idea of an *assignment statement*.]

Example Consider the expression A + B. There are two variables, A and B. A binding is any function that gives a value for each of these. For example,

$\{A \to 3, B \to 7\}$ is a binding giving a value of 3 to A and 7 to B.
$\{A \to 9, B \to 13\}$ is another binding,

and so on.

Definition: The **meaning** of an expression is a function from bindings to values.

Example Considering again the expression A+B, the meaning of A+B is the function that maps

$\{A \to 3, B \to 7\} \to 10$
$\{A \to 9, B \to 13\} \to 22$
etc.

The exact meaning of a given expression is defined recursively, in terms of meta rules. To avoid plunging too deeply into the theory at this time, we will rely on the intuitive meanings of such expressions.

In the chapter on Predicate Logic we will have occasion to extend this idea to other kinds of expressions.

Exercises

1 •• Extend the expression evaluator coded in Java to include the subtraction and division operators, - and /. Assume that these are on the same precedence level as + and *, respectively, and are also left-grouping.

2 •• Give a grammar for floating-point numerals.

3 •• Extend the expression evaluator coded in C++ to include floating-point numerals as operands.

8.10 Syntax-Directed Compilation (Advanced)

A common use of grammars is to control the parsing of a language within a compiler. Software tools that input grammars directly for this purpose are called *syntax-directed compilers* or *compiler generators*. We briefly illustrate a common compiler generator, yacc (*Yet Another Compiler-Compiler*). In yacc, the grammar rules are accompanied by program fragments that indicate what to do when a rule is successfully applied.

A second way to handle the issue of ambiguity is to leave the grammar in ambiguous form, but apply additional control over the applicability of productions. One form of control is to specify a precedence among operators; for example, if there is a choice between using the + operator vs. the * operator, always use the * first. In yacc, such precedence can be directed to the compiler, as can the grouping of operators, i.e. should a + b + c be resolved as (a + b) + c (left grouping) or a + (b + c) (right grouping).

Yacc Grammar for Simple Arithmetic Expressions

Below is the core of a yacc specification of a compiler for the simple arithmetic language. The output of yacc is a C program that reads arithmetic expressions containing numerals, +, and * and computes the result. The yacc specification consists of several productions, and each production is accompanied by a code section that states the action to be performed. These actions involve the symbols $$, $1, $2, etc. $$ stands for the *value* of the expression on the left-hand side of the production, while $1, $2, etc. stand for the value of the symbols in the first, second, etc. positions on the right-hand side. For example, in the production

```
expr:
        expr '+' expr
            { $$ = $1 + $3; }
```

the code fragment in braces says that the value of the expression will be the sum of the two expressions on the right-hand side. ($2 refers to the symbol '+' in this case.) Although the value in this particular example is a numeric one, in general it need not be. For example, an abstract syntax tree constructed using constructors could be used as a value. In turn, that abstract syntax tree could be used to produce a value or to generate code that does.

```
%left '+'             /* tokens for operators, with grouping          */
%left '*'             /* listed lowest precedence first               */

%start seq            /* start symbol for the grammar                 */

%%                    /* first %% demarks beginning of grammar rules  */

                      /* seq is LHS of first grammar rule.            */

seq :                                /* The "null" event"             */
      expr eol                       /* Parse expression.             */
       { printf("%d\n", $1);}        /* Print value of expr           */
    ;

expr   :
         expr '+' expr
           { $$ = $1 + $3; }         /* forward the sum               */
       | expr '*' expr
           { $$ = $1 * $3; }         /*   etc.                        */
       | number
           { $$ = $1; }
     ;

number :                             /* accumulate numeric value      */
         digit
           { $$ = $1; }
       | number digit
           { $$ = 10 * $1 + $2; }
       ;

digit  :                             /* get digit value               */
          '0' { $$ = 0; }  | '5' { $$ = 5; }
        | '1' { $$ = 1; }  | '6' { $$ = 6; }
        | '2' { $$ = 2; }  | '7' { $$ = 7; }
        | '3' { $$ = 3; }  | '8' { $$ = 8; }
        | '4' { $$ = 4; }  | '9' { $$ = 9; }
         ;

eol     : '\n';                      /* end of line                   */
%%
```

yacc source for a simple arithmetic expression calculator

8.11 Varieties of Syntax

So far, we have mostly considered expressions in "infix form", i.e. ones that place binary (i.e. two-argument) operators between expressions for the operands. As we have discussed, parentheses, implied precedence, and grouping are used to resolve ambiguity. Examples are

A + B
A + B * C
(A + B) * C
A - B - C

There are several other types of syntax that occur either in isolation or in combination with the types we have studied already. These are mentioned briefly in the following sections.

Prefix syntax

Prefix syntax puts the operator before the arguments. It is used in many languages for user-defined functions, e.g.

```
f(A, B, C).
```

The equivalent of the preceding list of expressions in prefix syntax would be:

```
+(A, B)
+(A, *(B, C))
*(+(A, B), C)
-(-(A, B), C)
```

Parenthesis-free Prefix Syntax

This is like prefix syntax, except that no parentheses or commas are used. In order for this form of syntax to work without ambiguity, the **arity** (number of arguments) of each operator must be fixed. For example, we could not use - as both unary minus and binary minus. The running set of expressions would appear as

```
+ A B
+ A * B C
* + A B C
- - A B C
```

Parenthesis-free prefix syntax is used in the Logo language for user-defined procedures.

Expressions in the Prolog language using arithmetic operators can be written in either infix or postfix.

Postfix and Parenthesis-free Postfix

This is like prefix, except the operator comes after the arguments. It is most usually seen in the parenthesis-free form, where it is also called **RPN (reverse Polish notation)**.

```
(A, B)+            A B +
(A, (B, C)*)+      A B C * +
((A, B)+, C)*      A B  + C *
```

((A, B)-, C)-   A B - C -

The parenthesis-free version of this syntax also requires the property that each operator must be of fixed arity.

S Expression Syntax

A grammar for S expressions was presented earlier. The notion of S expressions provide a syntax that, despite its initially foreign appearance, has several advantages:

> The arity of operators need not be fixed. For example, we could use a + operator that has any number of arguments, understanding the meaning to be compute the sum of these arguments. [For that matter, prefix and postfix **with parentheses required** also have this property.]
>
> The number of symbols is minimal and a parsing program is therefore relatively easy to write.

In S-expression syntax, expressions are of the form

> (*operator argument*-1 *argument*-2 .... *argument*-N)

where the parentheses are required. S-expressions are therefore a variant on prefix syntax. Sometimes S-expressions are called "Cambridge Polish" in honor of Cambridge, Mass., the site of MIT, the place where the language Lisp was invented. Lisp uses S-expression syntax exclusively.

> **In S expressions, additional parentheses cannot be added without changing the meaning of the expression.** Additional parentheses are never necessary to remove ambiguity because there is never any ambiguity: every operator has its own pair of parentheses.

Example: Arithmetic Expressions in S Expression Syntax

```
( + A B)
( + A ( * B C ) )
( * (+ A B) C)
( - ( - A B ) C )
```

S expressions are not only for arithmetic expressions; they can be used for representing many forms of structured data, such as heterogeneous lists. For example, the heterogeneous list used in an earlier rex example

```
[ my_dir,
  mail,
  [ archive, old_mail, older_mail],
  [ projects,
    [ sorting, quick_sort, heap_sort],
    [ searching, depth_first, breadth_first]
  ],
  [games, chess, checkers, tic-tac-toe]
]
```

could be represented as the following S expression:

```
( my_dir
  mail
  ( archive old_mail older_mail)
  ( projects
    ( sorting quick_sort heap_sort)
    ( searching depth_first breadth_first)
  )
  (games chess checkers tic-tac-toe)
)
```

S expressions are, in many ways, ideal for representing languages directly in their abstract syntax. The syntax is extremely uniform. Moreover, given that we have a reader for S expressions available, we don't need any other parser. We can analyze a program by taking the S expression apart by recursive functions. This is the approach taken in the language Lisp. Every construct in the language can play the role of an abstract syntax constructor. This constructor is identified as the first argument in a list. The other arguments of the constructor are the subsequent items in the list.

We saw above how arithmetic expressions are represented with S expressions. Assignment in Lisp is represented with the *setq* constructor:

`(setq` *Var* *Expression* `)`

sets the value of variable *Var* to that of expression *Expression*. A two-armed conditional is represented by a list of four elements:

`(if` *Condition* *T-branch* *F-branch*`)`

and function definitions are represented by the constructor *defun*. For example, the following is a factorial program in Lisp:

```
(defun factorial (N)
 (if ( < N 1 )
   1
   (* N (factorial (- N 1) ) ) ) )
```

In the next major section, we give examples of extendible interpreters that exploit the S expression syntax.

Expression-Tree Syntax

Expression trees are trees in which the leaves are labeled with variables or constants and the interior nodes are labeled with operators. The sub-trees of an operator node represent the argument expressions. The advantage of trees is that it allows us to visualize the flow of values that would correspond to the evaluation of an expression. These correspond closely to the abstract syntax trees discussed earlier.

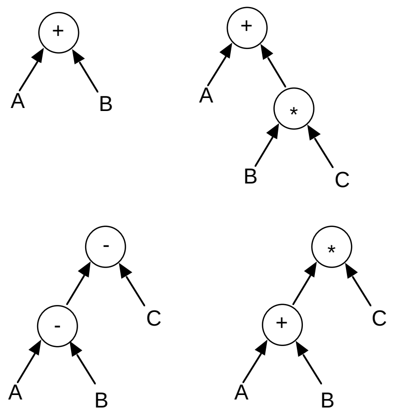

Figure 116: Expression-tree syntax for the equivalents of infix A+B, A+(B*C), (A+B)*C, and (A-B)-C (reading clockwise from upper-left)

DAG (Directed Acyclic Graph) Syntax

DAG syntax is an extension of expression-tree syntax in which the more general concept of a DAG (Directed, Acyclic, Graph) is used. The added benefit of DAGs is that sharing of common sub-expressions can be shown. For example, in

$$(A + B) * C + D / (A + B)$$

we might want to indicate that the sub-expression A + B is one and the same. This would be shown by having two arcs leaving the node corresponding to A + B. Note that is related to the issue of structure sharing which has appeared several times before.

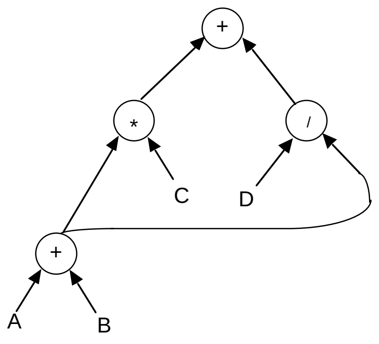

Figure 117: DAG for an arithmetic expression

DAG syntax is a subset of what is used in dataflow languages. Typically the latter allow loops as well, assuming that an appropriate semantics can be given to a loop, and are therefore not confined to DAGs (they can be cyclic). A typical example of languages that make use of loops are those for representing signal-flow graphs, as occur in digital signal processing.

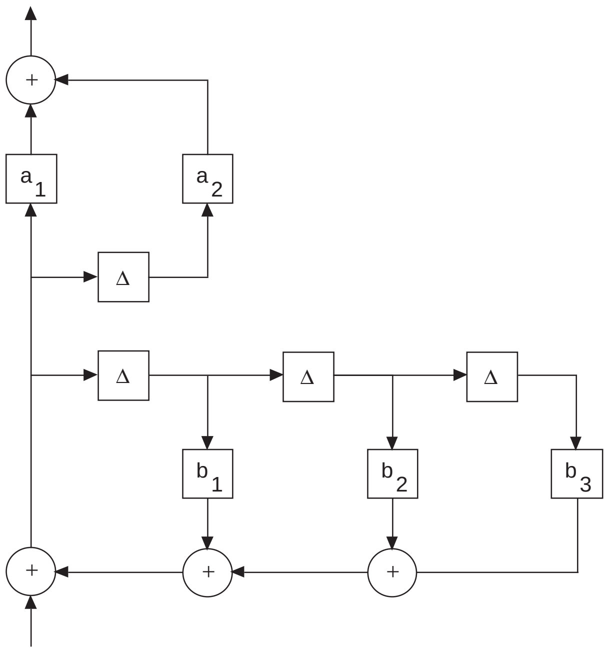

Figure 118: Graphical syntax for a digital filter

In the jargon of digital filters, a filter with loops is called "recursive".

The Spreadsheet Model

The spreadsheet model is a relatively user-friendly computing model that implicitly employs DAGs to represent computations. In a spreadsheet, the computational framework is represented in a two-dimensional array of cells. Each cell contains either a primitive data value, such as an integer or string, or an expression. An expression is typically an arithmetic or functional expression entailing operators and the names of other cells as variables. The spreadsheet system computes the values in cells containing expressions, and displays the value in the cell itself. Whenever the user changes one of the primitive data values in its cell, the relevant cells containing expressions are changed by the system to reflect the new value. The underlying mechanism is that of a DAG, where some nodes are identified with cells.

For simplicity, consider a 2 x 3 spreadsheet, as shown below. The cells in the spreadsheet are designated A1, A2, A3, B1, B2, B3.

	1	2	3
A	B1 * B2 + B3 / B1	value of A	value of B
B	A2 + A3	value of C	value of D

Figure 119: A simple spreadsheet representable as a DAG

Cell A1 of this spreadsheet represents the value of the previous DAG expression, if we associate the values A, B, C, and D with A2, A3, B2, and B3 respectively. The expression in cell B1 represents the common sub-expression shown in the DAG.

Typical states of the spreadsheet, with values entered by the user for A, B,C, and D, would show as follows:

	1	2	3
A	32.75	3	5
B	8	4	6

	1	2	3
A	50.7	4	6
B	10	5	7

Figure 120: Two different states of the simple spreadsheet

Exercises

1 ••• Give unambiguous grammars for representing arithmetic expressions with operators + and *:

(a) In postfix form

(b) In prefix form

(c) In S-expression form

2 •• Build an evaluator for expressions in postfix form. Although it could be done with recursion, such an evaluator is typically described using a "stack" data structure that retains operand values. Values are removed in the order of their recency of insertion:

> When an operand is scanned, it is pushed on the stack.
>
> When an operator is scanned, the appropriate number of arguments is removed from the stack, the operator is applied, and the result pushed on the stack.

Such is the principle on which RPN calculators work. It is also the basis of the FORTH language, and the PostScript™ language that is built on FORTH syntax.

3 ••• Build an evaluator for expressions in postfix form that uses no explicit stack. (An implicit stack is used in the recursive calls in this evaluator). [Hint: This can be done by defining recursive functions having arguments corresponding to the top so many values on the stack, e.g. there would be three functions, assuming at most 2-ary operators). When such a function returns, it corresponds to having used those two arguments.]

4 •• Having done the preceding two problems, discuss tradeoffs of the two versions with respect to program clarity, tail recursion, space efficiency, etc.

5 ••• Build an evaluator for expressions in prefix form.

6 •••• Devise an expression-based language for specifying graphics. Develop a translator that produces PostScript™. Display the results on a laser printer or workstation screen.

7 •••• Develop a spreadsheet calculator, including a means of parsing the expressions contained within formula cells.

8 •••• Explore the ergonomics of copying formulas in cells, as are done in commercial spreadsheet implementations. After devising a good explanation (or a better scheme), implement your ideas.

9 ••• Implement a system simplifying arithmetic expressions involving the operators +, -, *, /, and *exp* (exponent) with their usual meanings. For convenience, use S expression input syntax, wherein each expression is either:

A numeric constant

An atom, representing a variable

A list of the form (Op E1 E2), where E1 and E2 are themselves expressions and Op is one of the four operators.

The kinds of operations involved in simplification would include:

removing superfluous constants such as 1's and 0's:

(+ E1 0) → E1

(* E1 1) → E1

carrying out special cases of the operators symbolically:

(- E1 E1) → 0

(*exp* E 0) → 1

eliminating / in favor of *:

(/ (/ A B) C) → (/ A (* B C))

etc.

For example, if your system is given the input expression

```
(/ (/ A (* B (- C C)) D)
```

it would produce

```
(/ A (* B D))
```

These kinds of simplifications are included automatically in computer algebra systems such as Mathematica™ and Maple, which have a programming language component. However, the user is usually unable to customize, in an easy way, the transformations to suit special purposes.

10 •••• Implement a system in the spirit of the previous exercise, except include trigonometric identities.

11 •••• Implement a system in the spirit of the previous exercise, except include integrals and derivatives.

8.12 Chapter Review

Define the following terms:

abstract syntax tree
ambiguity
assignment
auxiliary symbol
basis
binding
countable
countably-infinite
DAG syntax
derivation tree
expression tree
finite
grammar
hypercube
induction rule
inductive definition
infinite
language
left recursion
natural numbers
parsing
precedence
prefix vs. postfix syntax
production
recursive descent
semantics
S expression
spreadsheet syntax
start symbol
syntax
syntax diagram
terminal alphabet

8.13. Further Reading

•• Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1985. [Introduces the PostScript™ language, which uses RPN syntax.]

••• A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1986. [Demonstrates how grammars are used in the construction of compilers for programming languages.]

• Alan L. Davis and Robert M. Keller, *Data Flow Program Graphs*, Computer, February, 1982, pages 26-41. [Describes techniques and interpretations of data flow programs.]

• Bob Frankston and Dan Bricklin, *VisiCalc*, VisiCalc Corporation, 1979. [The first spreadsheet.]

•• Paul R. Halmos, *Naive Set Theory*, Van Nostrand, Princeton, New Jersey, 1960. [Develops set theory from the ground up.]

• Brian Harvey, *Computer science Logo style*, MIT Press, 1985. [One of several possible references on the Logo language.]

•• S.C. Johnson, *Yacc – Yet Another Compiler-Compiler*, Computer Science Tech. Rept. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975. [Original reference on yacc.]

•• John R. Levine, Tony Mason, and Doug Brown, *lex & yacc*, O'Reilly & Associates, Inc., Sebastpol, CA, 1990.

••• J. McCarthy, *et al.*, *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass., 1965. [The original reference on Lisp list processing. Moderate.]

••• J. McCarthy and J.A. Painter, *Correctness of a compiler for arithmetic expressions*, in J.T. Schwartz (ed.), Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science, pp. 33-41, American Mathematical Society, New York, 1967. [The first mention of "abstract syntax".]

•• C.H. Moore, F*ORTH: A New Way to Program Minicomputers*, Astronomy and Astrophysics, Suppl. No. 15, 1974. [Introduces the FORTH language.]