Formal languages

We can view the syntax of a programming language as a game with the goal of distinguishing correct from incorrect programs. Given a program and the syntax rules of the language, you play the game by answering only one question: is this a correctly formed program? If the program matches the patterns in the rules you say 'Yes"; if the program does match the rules you say "No."

C program	Is it valid?

`main () {`
`  int X;`
`  X = 1;`
`  }`
	Yes.

`main [] {`
`  integer X;`
`  X = 1`
`  }`
	No. The brackets should be parentheses and a semicolon is missing.

Our game starts with strings of ASCII characters. Some of these strings look like C programs, such as the examples above; other sequences look nothing like C such as **`"This is English"`** or **`"*^%#HDF3DFF#"`**. The goal of a syntax for C is to sort all the possible ASCII strings into valid and invalid C programs. From this point of view the language C is simply the set of strings that meet our rules.

In our game if we say "Yes," a string is valid C, then we are saying the string is in this set. If we say "No," then we are saying the string is not in this set. This is all we mean by the term formal language; a formal language is a set of strings that meet a well-defined set of patterns or forms.

A *formal grammar* is a systematic way to define a formal language so we can distinguish strings in the language from strings outside the language. We will do this with a subset of C, simple arithmetic expressions that involve addition, subtraction and single digit numbers. A formal grammar has four parts: an *alphabet* of characters in the language, a set of *syntax names* not in the language but used in its definition, a distinguished syntax name called the *start symbol*, and a set of *syntax rules*. Formal grammar like this one are often called BNF for Backus-Naur Form, a notation developed by Backus and Naur for Algol.

Formal Grammar: **Simple arithmetic expressions in C - version 1**

Alphabet: the digits and operations **`0 1 2 3 4 5 6 7 8 9 + *`**
Syntax names: **`<E>`**
Start symbol: **`<E>`**

```
Rule 1     <E> ->
               <E> + <E>

Rule 2     <E> ->
               <E> * <E>

Rule 3     <E> ->
               0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Syntax names are written inside angle brackets **`<>`**. For our simple example we have only one, **`<E>`**, where E is short for Expression. When we talk about the syntax of English we use syntax names like "noun" and "verb"; the difference is that "noun" is a English word but **`<E>`** is not part of our simple arithmetic language.

Syntax rules are read as follows: whenever you see the symbol on the left-hand side in a formula, you can replace it with the right-hand side (the arrow means "<E> *goes to* <E> + <E>"). Whenever you see a vertical bar | it means you can choose between alternatives for the replacement (so the last rule is read "<E> *goes to* 1 *or* 2 *or* 3 etc.") To use the grammar is to create a *derivation* that produces valid formulas in the grammar. Here is a derivation for a simple formula "**1+2**":

```
<E>   ->  <E> + <E>   by Rule 1
      ->  1 + <E>     by Rule 3
      ->  1 + 2       by Rule 3
```

We start with the formula consisting of only the Start Symbol and make substitutions according to the rules, one step at a time, until we eliminate all syntax names; the result is a formula valid under this grammar. The set of all strings generated by the grammar is said to be the language defined by the grammar. We can apply the rules in any order that works so this is also a derivation of "**1+2**"; here we replace the "**1**" with **<E>** before the "**2**".

```
<E>   ->  <E> + <E>   by Rule 1
      ->  <E> + 2     by Rule 3
      ->  1 + 2       by Rule 3
```

If a formula is longer and more complex our derivation will be longer but it will still consist of a set of simple applications of the syntax rules. Here is a derivation for the string "**1+2*3**":

```
<E>   ->  <E> + <E>       by Rule 1
      ->  1 + <E>         by Rule 3
      ->  1 + <E> * <E>   by Rule 2
      ->  1 + 2 * <E>     by Rule 3
      ->  1 + 2 * 3       by Rule 3
```

Backwards derivations

So far we have done *forward* derivations. Forward derivations start with the start symbol and generate a string in the language (Chomsky called them generative grammars). In computer translation we have the inverse problem; given a formula, determine if it is valid by producing a derivation. Given "**3 * 1**", is it valid? We solve this by using the rules backwards; if you can match a subexpression to the right-hand side of one rule, then replace the subexpression by the symbol on the left:

```
3 * 1   ->  <E> * 1       by Rule 3   With no +, Rule 1 does not apply and
                                      without <E> Rule 2 does not apply.
        ->  <E> * <E>     by Rule 3
        ->  <E>           by Rule 2   Now Rule 2 applies.
```

If we can derive the start symbol the formula is valid. Just as with forward derivations, longer formulas require longer derivations but each step remains simple. Here is a derivation for "**6 * 9 + 5**"

```
6 * 9 + 5  ->  <E> * 9 + 5      by Rule 3
           ->  <E> * <E> + 5    by Rule 3
           ->  <E> + 5          by Rule 2
           ->  <E> + <E>        by Rule 3
           ->  <E>              by Rule 1
```

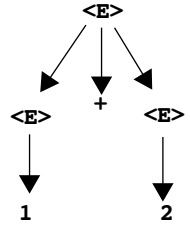
If we cannot derive the start symbol the formula is invalid. Consider "`1 # 1`"; no rule lets you replace a "`#`" so the derivation must stop at "`<E> # <E>`" Consider "1 + 1 * * 2"; no rule lets you replace a * *. Here is a failed derivation:

```
3 * 1 +    ->  <E> * 1 +        by Rule 3
           ->  <E> * <E> +      by Rule 3
           ->  <E> +            by Rule 2
           ->  ?                No rule matches - derivation fails.
                                We cannot reach a single <E>.
```

Formal grammars with forwards and backwards derivations give us a systematic way to determine if a string meets our syntax rules and belongs in the language. When computers compile your program they go through a backwards derivation.

Parse trees

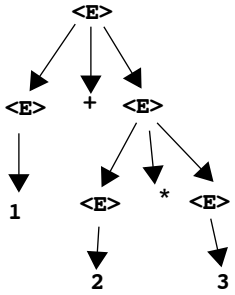
A *parse tree* is a graphical means of presenting a derivation. The example below shows a parse tree for our first forward derivation of the formula "`1 + 2`". At the top of the tree is the start symbol. Each time we apply a rule in our derivation, substituting for a syntax name, we show the substitution by drawing arrows from the syntax name downwards to the symbols that replaced it. In the diagram below our first rule, Rule 1, replaces **<E>** by **<E>**, **+** and **<E>** and yields the three arrows from **<E>** to its replacements.




```
<E>    ->  <E> + <E>     by Rule 1
       ->  1 + <E>       by Rule 3
       ->  1 + 2         by Rule 3
```

Figure 1.
Parse Tree for Example 1: 1 + 2

Here is a parse tree for the forwards derivation of "`1 + 2 * 3`".




```
<E>    ->  <E> + <E>         by Rule 1
       ->  1 + <E>           by Rule 3
       ->  1 + <E> * <E>     by Rule 2
       ->  1 + 2 * <E>       by Rule 3
       ->  1 + 2 * 3         by Rule 3
```

Figure 2.
Parse tree for "1+2*3"

In summary, formal grammars are intended to answer one question: does this string belong to the language defined by this grammar? Parse trees, forward derivation and backwards derivations are three techniques for applying a formal grammar to this question. Each technique has strengths, but all will come with the same result.

Exercises for 8.1.1

1. Give forward derivations and parse trees for the following expressions using the expression grammar given in this section.
   a. 6 * 7
   b. 3 * 6 * 8
   c. 1 + 2 + 3
   d. 7 + 3 * 1 + 2
2. Give backwards derivations for the following expressions with the same grammar.
   a. 2 + 5
   b. 7 * 5 + 4
3. How far could can you get before a forwards derivation for * 8 + 3 + 4 must fail ?
4. Extend the grammar to include the operations subtraction -, division / and exponentiation ^. Add any rules you need and use the extended grammar to give a forwards derivation of the expression 3^6 * 4 - 8/2 .

8.1.2 Formal rules for semantics

So far we have used our formal grammar to understand the syntax of expressions and classify strings as valid or invalid expressions. We now add semantic rules to our grammar to allow us to assign meanings to each expression.

Since we have simple arithmetic expressions with no variables the meaning of each expression is its numerical value. The value of "`7`" is 7; the value of "`5 + 4 + 5`" is 14. Notice that we distinguish between the symbol "`7`" and its numerical value 7. We will represent the value of an expression by the notation "`5 + 4`": 9 ; the expression will be followed by a colon and its value. This notation will apply to syntax names so if we use **<E>** to derive the expression "`5 + 4`" we can say **<E>**:9 . We will use parse trees to keep track of different instances of **<E>**.

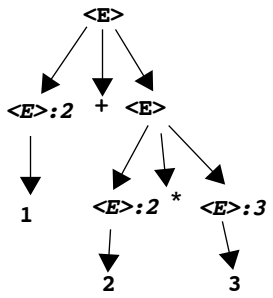
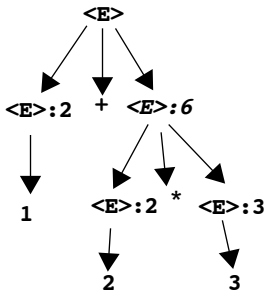
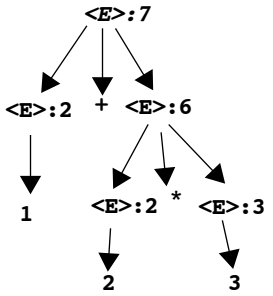
We will add a semantic rule for each syntax rule that tells us how to interpret an expression created by that syntax rule. If we create an expression by the Rule 3, **<E> -> 8**, then we say the meaning of **<E>** is **<E>**:8. In each of the semantics rules below we are talking about how to give a meaning to the syntax name **<E>** on the left-hand side of the rule.

TABLE 1. Semantic rules for simple arithmetic grammar.

Rule	Syntax Rule	Corresponding Semantic Rule
1	**<E> -> <E> + <E>**	**<E>** gets the sum of the two subexpressions.
2	**<E> -> <E> * <E>**	**<E>** gets the product of the two subexpressions.
3	**<E> -> 1 | 2 | 3 ... | 9**	**<E>** gets the value of the numerical digit.

Our semantic rules allow us to "bubble up" a meaning for a parse tree. At the bottom of every parse tree will be expressions that are single numerical digits; these we can immediately give values. Once the subexpressions for an expression have been assigned a value

we can compute, by addition or multiplication, a value for that expression and then bubble the value up. Here is an example for the expression "`1 + 2 * 3`".

Steps in interpretation	Parse tree with attached values
Use Semantic Rule 3 to give values to all expressions that are single digits.	
Now apply Semantic Rule 2 to give a value 6 to the multiplication subexpression.	
Finally apply Semantic Rule 1 to give a value to the entire expression by bubbling up the value 7 to the top.	

Ambiguity

Since our semantic rules are tied to our syntax rules the value of an expression becomes dependent on the sequence of rules used to derive it. If we reverse the order of the applica-

tion of rules 1 and 2 in the derivation of "`1 + 2 * 3`" we can get a second value for the expression as follows:

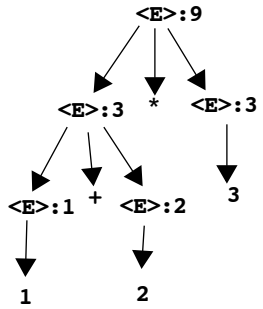


Figure 3.
Second parse tree for "1 + 2 * 3"

The meaning of this parse tree is 9, different from the value of 7 we gave to the earlier parse tree for "`1 + 2 * 3`". The two values result from either doing addition first, so we have (1 + 2) * 3, or multiplication first, so we have 1 + (2 * 3). This expression is said to be ambiguous because it has two parse trees.

A grammar that allows more than one parse tree and value for any expression is also said to be *ambiguous*. We do not want this for programming languages; we want single, clear meanings for programs. In standard C multiplication is done before addition; the value of this expression should be 7, not 9. To correct this we will develop a more complex grammar that treats multiplication differently from addition. We have also added subtraction and division for completeness.

Formal Grammar: **Simple arithmetic expressions in C - version 2**

Alphabet: digits, operations `0 1 2 3 4 5 6 7 8 9 + - * /`
Syntax names: `<E> <T> <N>` *This version adds T (term) and N (number)*
Start name: `<E>`

```
Rule 1     <E> ->
               <E> + <T> | <E> - <T>

Rule 2     <E> ->
               <T>

Rule 3     <T> ->
               <T> * <N> | <T> / <N>

Rule 4     <T> ->
               <N>

Rule 5     <N> ->
               0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

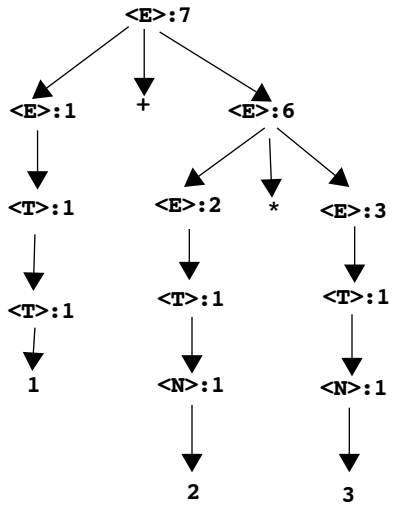
Our new rules and syntax names force the grammar to recognize multiplication (*) before addition (+), and to recognize them left to right, giving the traditional operator precedence

rules for C. The grammar does this by requiring the rules defines terms **<T>** and numbers **<N>** (rules 3, 4 and 5) to match the formula before you can use the rules defining **<E>**.

TABLE 2. Semantic rules for simple arithmetic grammar - version 2

Rule	Syntax Rule	Corresponding Semantic Rule
1	**<E> -> <E> + <T>**	**<E>** gets the value of the sum.
	| <E> - <T>	**<E>** gets the value of the difference.
2	**<E> -> <T>**	**<E>** gets the value of **<T>**
3	**<T> -> <T> * <N>**	**<T>** gets the value of product.
	| <T> / <N>	**<T>** gets the value of the division.
4	**<T> -> <N>**	**<T>** gets the value of **<N>**
5	**<N> -> 1 | 2 | 3 ... | 9**	**<N>** gets the value of the digit.

Here is a derivation and semantic interpretation using our second formal grammar.




```
<E>  ->  <E> + <T>          by Rule 1
     ->  <E> + <T> * <N>    by Rule 3
     ->  <E> + <T> * 3      by Rule 5
     ->  <E> + <N> * 3      by Rule 4
     ->  <E> + 2 * 3        by Rule 5
     ->  <T> + 2 * 3        by Rule 2
     ->  <N> + 2 * 3        by Rule 4
     ->  1 + 2 * 3          by Rule 5
```

Figure 4.
Parse Tree for Example 2

If we take our semantics from derivation 4, we get only one possible meaning: 7. The constraints in our second formal grammar allow us to construct only on possible tree. This grammar is unambiguous.

Natural language grammars and ambiguity

Consider the sentence

Bill saw Ann with the telescope.

This sentence has two reasonable meanings:

Bill looked through the telescope and saw Ann.

Bill saw Ann while she had the telescope.

If we construct a grammar for simple English sentences we will find we can construct multiple parse trees for this example. Here is a possible grammar:

Formal Grammar: **A very simple grammar for English sentences**

Alphabet:
Syntax names: **<S>** *sentence* **<NP>** *noun phrase* **<VP>** *verb phrase*
<PP> *prepositional phrase* **<N>** *noun* **<V>** *verb* **<D>** *determiner*
Start name: **<S>**

```
Rule 1     <S> ->
               <NP> <VP>

Rule 2     <NP> ->
                <N> | <D> <N> | <D> <N> <PP>

Rule 3     <PP> ->
               <P> <NP>

Rule 4     <VP> ->
               <V> <NP> | <V> <NP> <PP>

Rule 5     <N> ->
               noun: examples Bill, Ann, telescope

Rule 6     <V> ->
               verb: example saw

Rule 7     <D> ->
               determiner: example the

Rule 8     <P> ->
               preposition: example with
```

The syntax names in this grammar represent typical parts of English speech. Sentences <S> are composed of a noun phrase <NP> and a verb phrase <VP> indicating someone and what they did. Both noun phrases and verb phrases can be modified by prepositional phrases <PP> indicating exactly who or what was done.

The example sentence, Bill saw Ann with the telescope, is ambiguous because the prepositional phrase “with the telescope” can modify either the verb phrase or the noun phrase with Ann. If “with the telescope” modifies the verb phrase it indicates that Bill used the telescope so we have our first interpretation. If “with the telescope” modifies the noun phrase it indicates that Ann had the telescope so we have our second interpretation. Parse trees for the two interpretations are given in the diagram below.

We could change our grammar to force one or the other interpretation, but this would be an artificial view of English. English allows both interpretations; it is not the grammar that

is ambiguous but the language itself. We use information other than the sentence (Does Bill have a telescope? Does Ann?) to disambiguate English sentences.

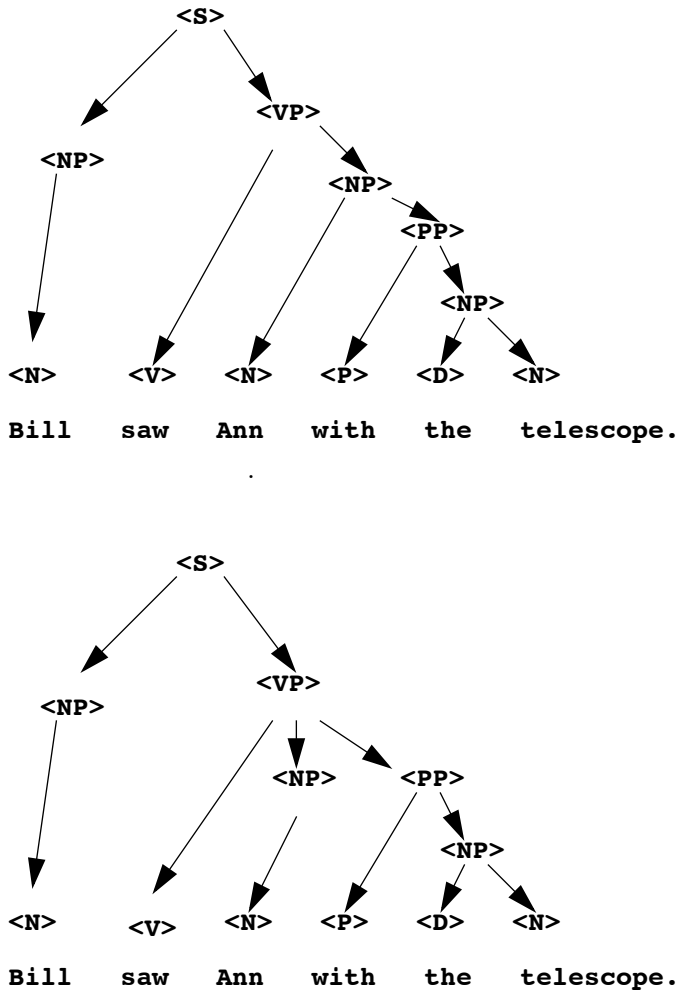


Figure 5.
Two parse trees for an English sentence

Computer languages have less forgiving rules of syntax than natural languages. You and I could figure out "Ball here now throw," and we could even guess that the algebraic formula "(12+2" should be 14 but a parentheses is missing. Computers are less flexible than us, hence the brittle syntax of their languages. This is a major source of frustration for beginning programmers. However, the brittle syntax is often a design choice. Computers can and frequently do correct simple errors like the last one by adding the parentheses, but the more choices you let the computer make in correcting your programs the more likely the computer may introduce an error.

Exercises for 8.1.2

1. Using the first version of the expression grammar, draw parse trees for the following expressions and label the trees with the values given by the first set of semantic rules.
   a. 3 + 5 * 3
   b. 7 * 2 + 6
   c. 8 * 9 + 3 * 4
2. Use expression (a) above to show that the first version of the expression grammar is

ambiguous.

3. Use the second version of the expression grammar to draw parse trees for expressions (a), (b) and (d) in exercise 1.
4. Label the parse trees you drew in exercise 3 with values given by the second set of semantic rules.
5. Using the grammar for English sentences, give a parse tree for the following sentences.
   a. Ann threw Bill.
   b. Ann threw Bill over the hedge.
   c. A hedgehog with attitude threw Ann over the hedge.

8.1.3 Using grammars for language translation

How would a compiler translate the C assignment statement

```
Y = 3 + 2 * X;
```

into machine language? Our examples in the previous section showed how to use semantic rules to compute a numerical value for an expression, but a compiler needs to do more to translate high-level language elements into machine instructions. Compilers use semantic rules that translate parts and pieces of high-level code into small sections of machine code, and then put those small sections together into a complete program.

In Section 7.2.2 we presented a translation of the statement above into the SIMPLE machine language, but we did not show how this translation might be algorithmically derived. In this section we will show how to use the formal grammar techniques from 8.1.1 to do the translation.

TABLE 3. SIMPLE machine language translation of Y = 3 + 2 * X;

Memory Address	Assembly Code or Data	Machine or Binary Code Translation	Interpretation of contents
0	2	0000 0010	Storage for constant 2
1	3	0000 0011	Storage for constant 3
2	5	0000 0101	Storage for variable X
3	0	0000 0000	Storage for variable Y
4	0	0000 0000	Storage for 2*X
5	0	0000 0000	Storage for 2*X + 3
6	Load 0,A	000 00000	Load constant 2 into A
7	Load 2,B	000 10010	Load value of X into B
8	Mult A,B,4	100 00100	Multiply 2 by X, store in 4
9	Load 4,A	000 00100	Load value of 2*X into A
10	Load 1,B	000 10001	Load constant 3 into B
11	Add A,B,5	010 00101	Add 2*X to 3, store in 5
12	Load 5,A	000 00101	Load value of 2*X+3 into A
13	Store A,3	001 00011	Store 2*X+3 into Y
14	0	0000 0000	
15	0	0000 0000	

To translate this formula we will need to add two syntax rules to the second version of the formal grammar for expression in C. The two rules will adopt the grammar to allow variables as part of a expression and will give form to the assignment statement.

Formal Grammar: **Simple arithmetic expressions in C - version 3: additions for variables and assignment statements**

New syntax name: **<assignment-statement>**

```
Rule 6     <variable> ->
               A | B | C | X | Y | Z

Rule 7     <N> -> <variable>

Rule 8     <assignment-statement> ->
               <variable> = <E> ;
```

Here is a backwards derivations of our assignment statement:

Y = 3 + 2 * X;	->	Y = 3 + <N> * X;	*by Rule 5*
	->	Y = 3 + <T> * X;	*by Rule 4*
	->	Y = 3 + <T> * <variable>;	*by Rule 6*
	->	Y = 3 + <T> * <N>;	*by Rule 7*
	->	Y = 3 + <T>;	*by Rule 3*
	->	Y = <N> + <T>;	*by Rule 5*
	->	Y = <T> + <T>;	*by Rule 4*
	->	Y = <E> + <T>;	*by Rule 2*
	->	Y = <E>;	*by Rule 1*
	->	<variable> = <E>;	*by Rule 6*
	->	<assignment-statement>	*by Rule 8*

Semantic records

We also need to modify the semantic rules. Instead of keeping only a single numeric value for every syntax name in our derivation, we will keep a *semantic record* that includes two parts. A semantic record of an data object like a variable, constant or expression will have an *address* where the value of that object is stored. A semantic record of a operation object, like an assignment statement or expression, will have machine *code* that gives show how to compute the object. Notice that expressions will have both; an expression gives a set of instructions that result in a data value. Variables and constants will only need the address, while many statements will only need the code. Semantic rules will direct how to create a semantic record for a given language element.

Here are semantic records for a single constant, **3**, and an addition expression, **3 + 4**. The constant is stored in a memory location (remember, SIMPLE uses addresses 0-15.) When we first translate **3**, we may not know where it will be stored so it gets a memory location represented by the variable M1. Similarly, in the expression **3 + 4** the constant **4** will get a variable memory location M2. To add two constants, SIMPLE first loads them into registers A and B, and then uses the Add instruction. This is the code part of the semantic

record of the expression **3 + 4**. Since the addition results in a value, this value must be given a temporary location M3.

Semantic record for the constant **3**

Address	Code
M1	*none needed*

Semantic record for the expression **3 + 4**

Address	Code
M3	Load M1,A Load M2,B Add A,B,M3

Figure 6.
Parse Tree for Example 2

Compilers use a *symbol table* to keep track of the memory locations for constants, variables and temporary locations. After the compiler has figured out how many locations it needs, it replaces variables like M1, M2 and M3 with real locations.

The figure below shows how the we might attach semantic records to a parse tree of the expression **<E> -> <E> + <E>**. The semantic records are labeled T1, T2 and T3 to distinguish them. The semantic record T3 for the entire expression glues together the semantic records for the two subexpressions. The code in T3 sequences the code for T1 and T2 before the code required to compute the addition operation.

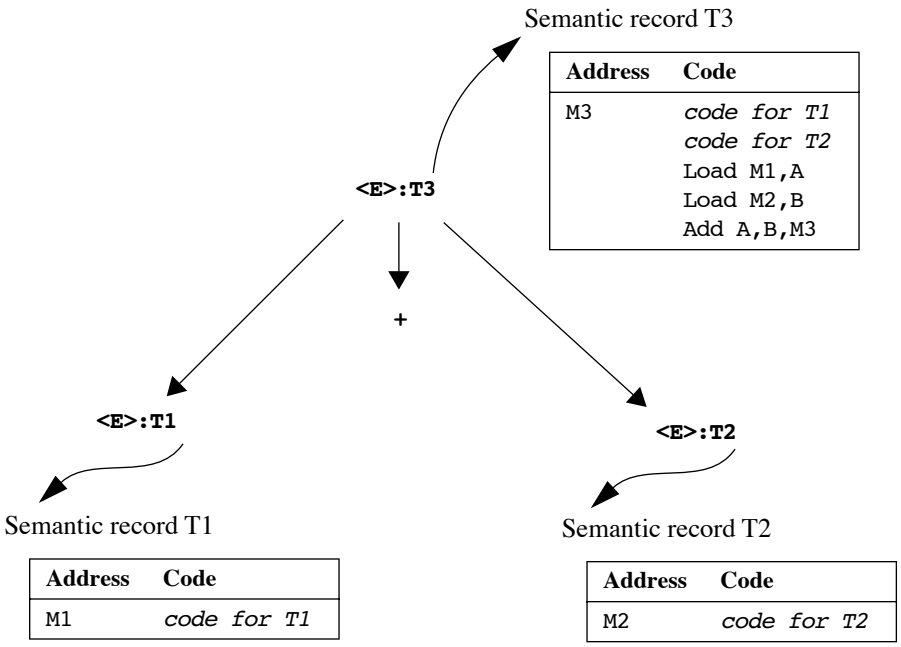


Figure 7.
Parse Tree for Example 2

On a small scale, this is how a compiler translates a program. The compiler starts by creating semantic records for data objects at the bottom of the parse tree and then passes these records up the parse tree, gluing them together at each syntax name.

Finishing our original goal: translating Y = 3 + 2 * X

Let us use the methods just explained to translate `Y = 3 + 2 * X`. From our backwards derivation we have the parse tree:

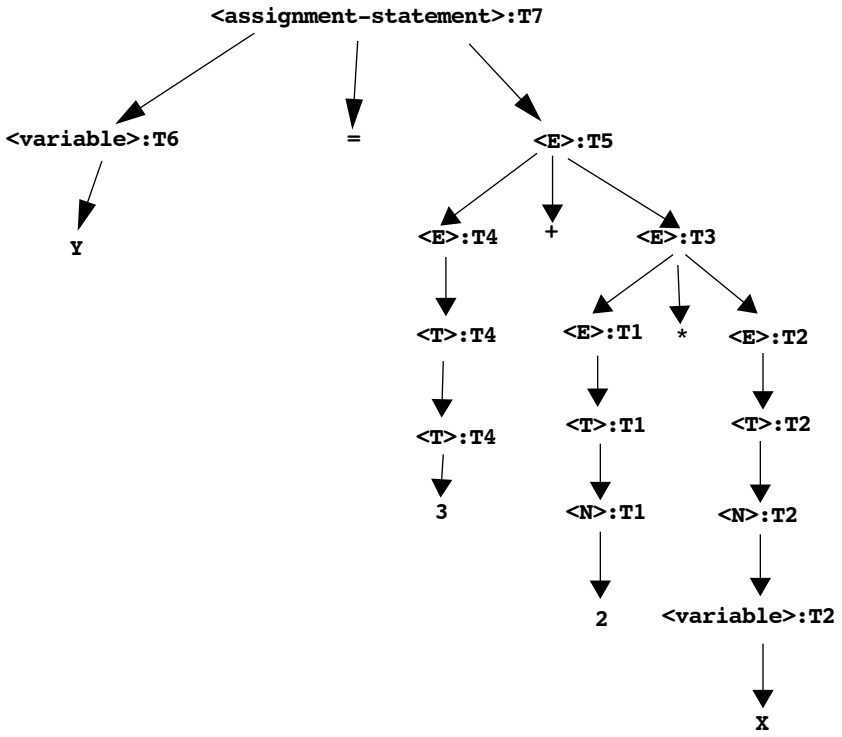


Figure 8.
Parse Tree for Example 2

In this parse tree each syntax name has an attached semantic record; the records are labeled T1 to T7. Many of the derivation steps do not change the semantic records, so they are passed up the tree unchanged; for example, on the right the record T2 is passed up four levels before it is combined into T3. The semantic records are given in the table below, indexed by their label.

TABLE 4.

Semantic record label	Address	Code	Expression translated
T1	0		The constant 2.
T2	2		The variable X.
T3	4	Load 0,A Load 2,B Mult A,B,4	The expression 2*X
T4	1		The constant 3.

T5	5	Load 0,A
		Load 2,B
		Mult A,B,4
		Load 4,A
		Load 1,B
		Add A,B,5
			The expression 3 + 2*X

TABLE 4.

Semantic record label	Address	Code	Expression translated
T6	3		The variable Y.

T7		Load 0,A
		Load 2,B
		Mult A,B,4
		Load 4,A
		Load 1,B
		Add A,B,5
		Load 5,A
		Store A,3
			The assignment statement Y = 3 + 2*X.

As you can imagine, for a large program the number and complexity of the semantic records may become immense. The compiler must have rules for all types of expressions, control structures like if and for, function calls and much more. But, compilation is not a magic process; each step can be written down and straightforwardly explained.

Exercises for 8.1.3

1. Give a parse tree with semantic records for the statement `Z = 2 + 2.` Show the assembly language translation that results.
2. Give a parse tree with semantic records for the statement `C = 9 * 4 * 5`. Show the assembly language translation that results.
3. Why is a symbol table needed? Why is it that a constant or variable cannot be given an address immediately, when its semantic record is created?
4. Give a set of semantic rules for generating semantic records. Your set should have seven rules to correspond with the seven syntax rules.

8.1.4 Grammars for Fractal trees

Formal grammars have not just been applied to natural and computer languages. They have a wide range of applications throughout the arts and sciences. A wonderful application that merges art and science is the generation of artificial plants. Botanist Aristid Lindermeyer started working on artificial plants in the 1960s and from his work came a specialized set of grammars called L-systems. L-systems are used to model the growth of plants. Algebraic equations can model the rate of growth of single cells, tissues and stems, but have a hard time with branching structures; as you can imagine from our parse trees, formal grammars can easily model the production of branches.

An L-system is very similar to a formal grammar, with one major difference; the interpretation of the resulting language. Instead of treating the language as mathematical formulas or English sentences, we will interpret each string as a set of instructions for a drawing a figure. We will use Turtle graphics as we did in Chapter 3 to draw fractal trees, but this time L-systems will substitute for recursion C calls.

Turtle graphics using command strings

To review, Turtle graphics uses an imaginary turtle that draws on in an (x,y) coordinate system according to a few short rules. The turtle has a state (x,y,A), where it is at position (x,y) and pointed a direction defined by the angle A.

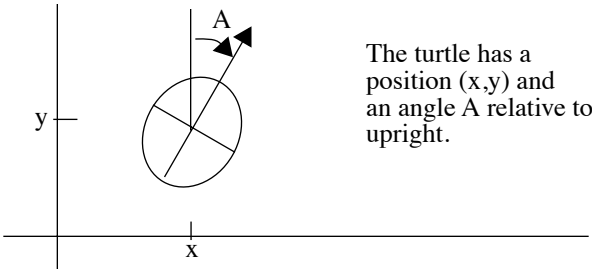


Figure 9.
Definition of the Graphics Turtle

For our L-system application we will reduce our turtle to three commands, each expressed as a single symbol: F, + and -. F means more ahead a distance d; + means rotate delta degrees left; and - means rotate delta degrees right. The values of d and delta are fixed in advanced of each figure; all the Fs during one session means the same distance d. After a command the turtle updates its state (x,y,A).

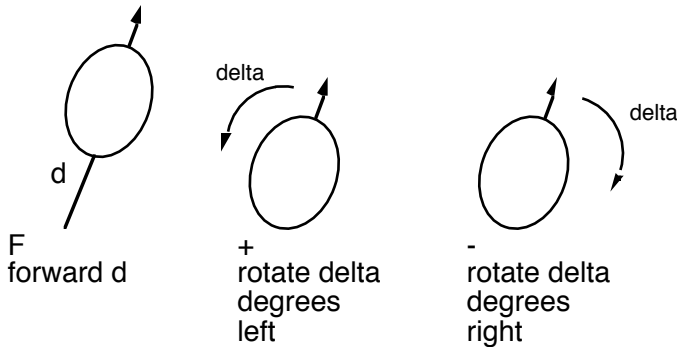


Figure 10.
Definition of Turtle Graphics Commands

We can now command our turtle by giving it a string composed of Fs, +s and -s. To draw a box, we set d = 1 inch and delta = 90 degrees. We start with the turtle facing up at the origin of our coordinate system, so the initial state is (0,0,90). The string for drawing a box

would be F+F+F+F, an F for each of the four sides. The figure below shows how the turtle draws F+F to start the box.

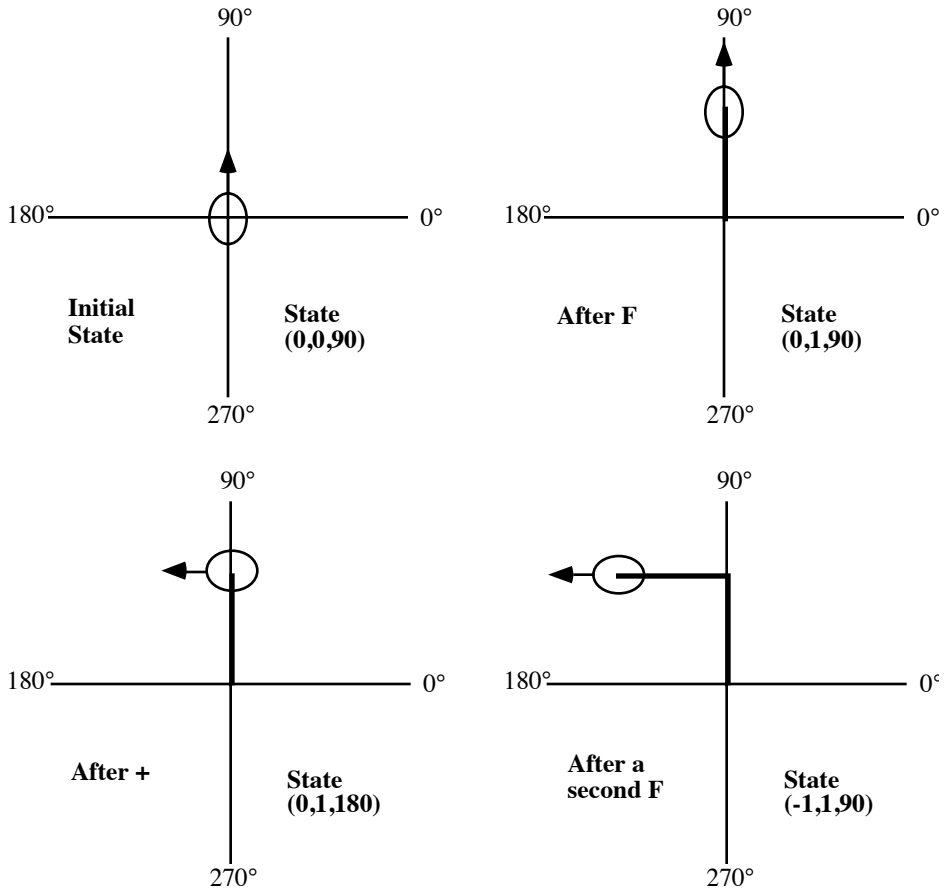


Figure 11.
Example of Turtle Graphic Commands

Drawing a branch

You can draw all types of figures with the simple commands F, + and -, but they do force the turtle to draw in one continuous line as if the turtle had a pen that could not leave the paper. One way to solve this is to give the turtle a little bit of memory, so the turtle can draw one part of a figure, remember where it is, go someplace else and draw another part of the figure, and then return to where it was. To go back to where it was earlier, the turtle must remember its previous state with location and direction. We used recursion in Chapter 3 to remember where the turtle was, but this time we will not.

This time we will give our turtle a *stack* for memory. A stack is a special kind of memory that can store as many previous states as we would like, but that only allows the turtle to retrieve the most recently stored state. Imagine that to remember a state, our little turtle writes down (x,y,A) on a sheet of paper and drops in the sheet on top of a pile. When the turtle wants to retrieve a location, it must pick up a sheet and read the state. But, a stack

restricts the turtle to the most recent sheet. The turtle cannot leaf through the pile looking for a particular place, it must take the one on top. When it reads a state, it throws it away.

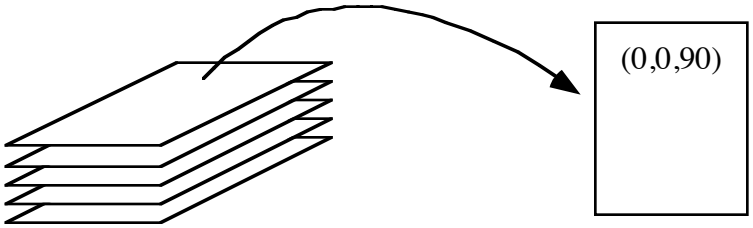


Figure 12.
Example of Turtle Graphic Commands

To command the turtle to remember where it is, we will use two new symbols. An opening bracket, `[`, tells the turtle to place its current state on the stack. A closing bracket, `]`, tells the turtle to take the current state off the top of the stack and move to that location. Here is a example of how to draw a branch with the commands `F[-F]F`; d is set to 1/2 inch and delta to 45°.

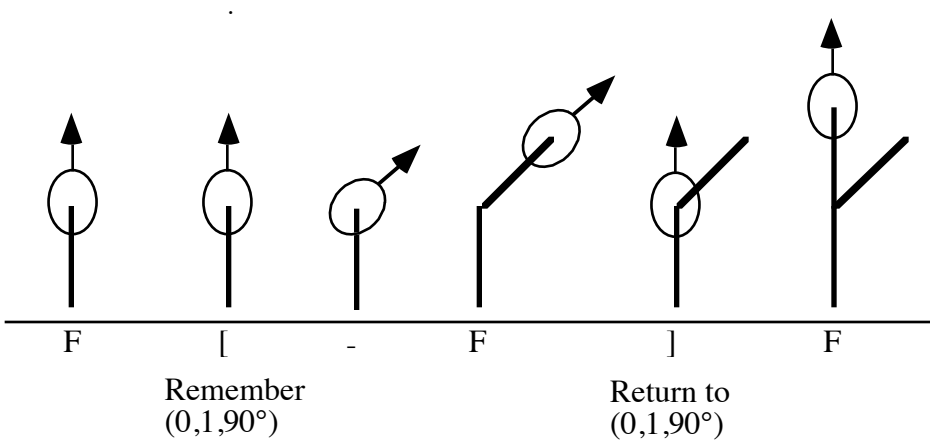


Figure 13.
Example of Turtle Graphic Commands

Here is a example of how to draw a branch with the commands **F[F[+F]F]-F** with the same values for d and delta. In this case the stack has to contain two states to handle two branches.

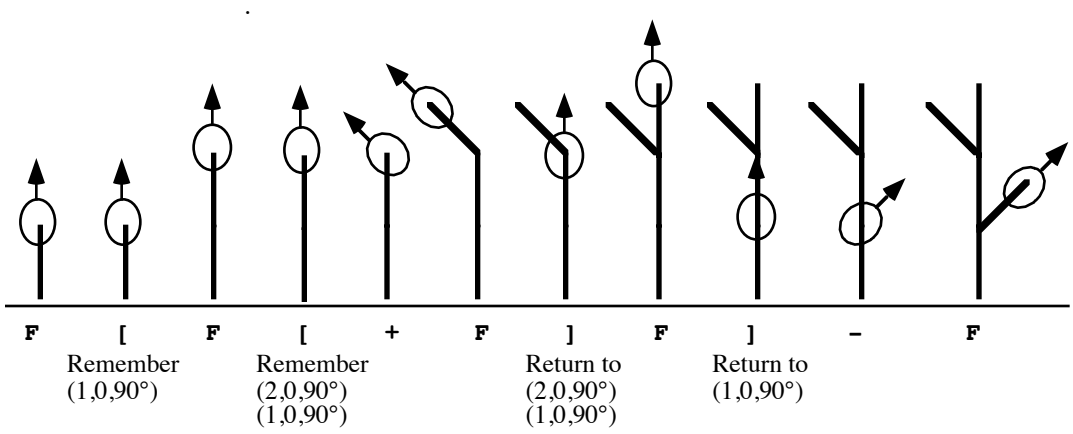


Figure 14.
Example of Turtle Graphic Commands

An L-System using Turtle graphics

Our L-systems will use this modified Turtle graphics to create strings of commands and then draw them. The L-systems use a BNF to generate a string, and then Turtle graphics to interpret the string; Turtle graphics are the semantics of the system. Here is an L-system for a symmetric bush. The system has only one syntax name and one rule.

Formal Grammar: **L-system for a symmetric bush**

Alphabet: **F + - [ ]**
Start character: **F**
Parameters: Distance = 1, Delta = 45°

```
Rule       F ->
                F[+F]F[-F]F
```

To use this grammar, you generate a derivation of n steps; at each step, you replace all possible Fs simultaneously. So if n=2, we'd have:

```
F    ->  F[+F]F[-F]F
     ->  F[+F]F[-F]F[+F[+F]F[-F]F]F[+F]F[-F]F[-F[+F]F[-F]F]F[+F]F[-F]F
```

As you can see, the strings get very long very soon, hence the need for a computer to handle the details. For n=3 the string has 301 characters. Once you have a string you send it to

a turtle for interpretation, and it draws the bush. Here are three generations of this L-system:

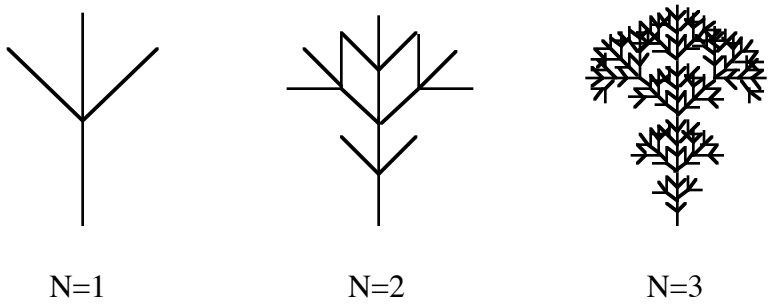


Figure 15.
Example of Turtle Graphic Commands

Other examples of L-Systems

The examples in the next figure appear in **The Algorithmic Beauty of Plants** by Przemyslaw Prusinkiewicz and Aristid Lindermayer. The upper two and lower left are plants, the lower right an abstract curve that could be a coastline. The next figure also comes from this book, but it uses a more complex form of Turtle graphics to draw a model of the bush

in a full three dimensions before the picture is created. Another example is included on color plate XX.

n = 5, delta = 25.7° Rule: F -> F[+F]F[-F]	n = 4, delta = 22.5° Rule:F-> FF- [-F+ F+F]+[+F-F-F]
n = 5, delta = 25.7° Rule: F -> F[+F]F[-F][F]	n = 4, delta = 90° Rule: F -> F-F+F-F-F

Figure 16.
Parse Tree for Example 2