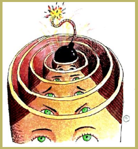
Lecture 7: Formal Languages

Aims:

- To define strings, alphabets and languages; and
- To look at syntax diagrams (being one way to define the syntax of programming languages).



7.1. Formal Languages

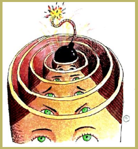
- We have been using the DECAFF language to describe algorithms to one another. It isn't a real programming language. In DECAFF, we are writing *pseudocode*. If we want to instruct a computer, we have to use something much more precise. Languages with precise syntax and semantics are called *formal languages*.
- *Programming languages* are examples of *formal languages*.
- *Formal languages* are defined by two sets of rules:
  - *Syntax:* precise rules that tell you the symbols you are allowed to use and how to put them together into legal expressions.
  - *Semantics:* precise rules that tell you the meanings of the symbols and legal expressions.
- Consider learning a new language (even a language such as French, say). How are you taught what words there are and how to put these together to make grammatical sentences (the syntax)? How are you taught what the words and phrases mean (the semantics)?

  Answer: Someone tells you these rules in a language whose rules you already know.

  Object language: the language under discussion e.g. the one you are being taught;

  Metalanguage: the language in which the object language is discussed.

  For most of us, the usual metalanguage is English. The object language might be French, German, Japanese or the Java programming language.



7.2. Strings

- We start with the notion of a **symbol**. In principle, any object can be a symbol. But we will only use characters: letters ($a, b, c, \ldots$), digits ($0, 1, 2, \ldots$) and maybe a few punctuation marks ($\#, \$, \ldots$). One symbol we might want to use is a space (or blank). For explicitness, we will show space as ␣.

- A **string** is a finite sequence of symbols. E.g.

  001 abba cat␣dog

  (In Theoretical Computer Science, *sentence* and *word* are used interchangeably with *string*. I'll avoid this because students find it confusing: they get confused because using *sentence* and *word* in this way does not tally with their everyday meanings.)

  We can give a string a name; common names are u, v and w. To avoid confusion, the symbols we use to name strings should be different from the symbols that can appear in the strings themselves.

- There is a string that contains no symbols at all. It is called the **empty string** and it is written ϵ.

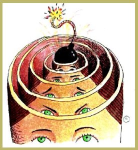
  Compare these two strings:

  $$\begin{aligned} u &=_{\text{def}} \text{␣} \\ v &=_{\text{def}} \epsilon \end{aligned}$$

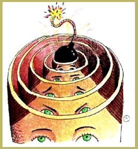
  They are different! u contains one symbol; v contains no symbols at all.

- The **length** of a string w, written $|w|$ is the total number of symbols in the string. E.g.

  $$|001| = 3 \quad |cat\text{␣}dog| = 7 \quad |\epsilon| = 0$$



- If u and v are strings, then the **concatenation** of u and v, written uv, is the string formed by the symbols of u followed by the symbols of v. E.g. if $u =_{\text{def}} abc$ and $v =_{\text{def}} de$, then
$$uv = abcde$$
- If w is a string, then we will write w^i to mean w repeated i times. We take w^0 to be ϵ for any string w. E.g. suppose $w =_{\text{def}} ba$, then
$$\begin{array}{lllllll} w^0 & = & \epsilon & w^2 & = & baba \\ w^1 & = & ba & w^3 & = & bababa \end{array}$$
and so on.
- If w is a string, then we will write w^R to mean the **reversal** of string w. E.g. suppose $w =_{\text{def}} ba$, then
$$w^R = ab$$



7.3. Alphabets

- An **alphabet** is a finite set of symbols. A common name for an alphabet is Σ. E.g.

$$\begin{array}{lll} \Sigma_1 & =_{\text{def}} & \{0,1\} \\ \Sigma_2 & =_{\text{def}} & \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\} \end{array}$$

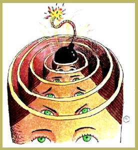
- Strictly, when we define strings as we did earlier, we ought to do so with respect to a particular alphabet. E.g.

$$001 \text{ is a string over the alphabet } \{0,1\}$$

$$abba \text{ is a string over the alphabet } \{a,b\}$$

- If Σ is an alphabet, then Σ^* is the set of all strings over alphabet Σ. It will include the empty string, and it will be an infinite set.

  E.g. if $\Sigma = \{0,1\}$, then Σ^* is all *bit strings*, including the empty string.



7.4. Languages

- A **language** is a set of strings formed from some alphabet. E.g.

$$\begin{array}{lll} L_1 & =_{\text{def}} & \emptyset \\ L_2 & =_{\text{def}} & \{\epsilon\} \\ L_3 & =_{\text{def}} & \{000, 001, 010, 011, 100, 101, 110, 111\} \end{array}$$

  Of course, this definition from Theoretical Computer Science only partly ties in with our understanding of the everyday meaning of the word *language*, which we might define in terms of communication. That, remember, is why we sometimes use the phrase *formal language* instead.

- Recall that Σ^* is the set of all strings that can be formed over alphabet Σ. Then, we can see that a language is a *subset of* Σ^*.

- Since languages are sets, we can define them extensionally (see the examples above) or intensionally. Many languages will be infinite sets and, for these, only intensional definitions can be used. They will tend to take the form

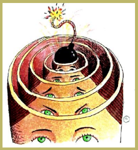
$$\{w \in \Sigma^* \mid P(w)\}$$

  i.e. the set of strings over alphabet Σ that satisfy property P. E.g.

$$\begin{array}{lll} L_1 & =_{\text{def}} & \{w \in \{0,1\}^* \mid w \text{ has an equal number of 0's and 1's}\} \\ L_2 & =_{\text{def}} & \{w \in \Sigma^* \mid w = w^R\} \end{array}$$

- Languages are sets, so we can apply all the usual set operations (e.g. union and intersection).

- If L_1 and L_2 are languages, then the **concatenation** of L_1 and L_2, written L_1L_2, is the language consisting of all strings uv which can be formed by selecting a string u



from L_1, a string v from L_2 and concatenating them in that order. More formally,

$$L_1L_2 =_{\text{def}} \{uv \mid u \in L_1 \text{ and } v \in L_2\}$$

E.g. if $L_1 =_{\text{def}} \{0, 01, 110\}$ and $L_2 =_{\text{def}} \{10, 110\}$, then

$$L_1L_2 = \{010, 0110, 01110, 11010, 110110\}$$

- If L is a language, then we will write L^i to mean $LL \ldots L$ (i times). We take L^0 to be $\{\epsilon\}$.

  E.g. suppose $L =_{\text{def}} \{a, ab\}$, then

$$\begin{array}{lcl} L^0 & = & \{\epsilon\} \\ L^1 & = & \{a, ab\} \\ L^2 & = & \{aa, aab, aba, abab\} \\ L^3 & = & \{aaa, aaba, abaa, ababa, aaab, aabab, abaab, ababab\} \end{array}$$

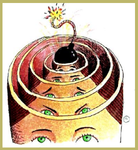
  and so on.

- If L is a language, then the **closure** of L, written L^*, is the concatenation of L with itself any number of times. Formally,

$$L^* =_{\text{def}} \bigcup_{i=0}^{\infty} L^i$$

  i.e. $L^0 \cup L^1 \cup L^2 \cup \ldots$.

  E.g. if $L = \{aa\}$, then L^* is all strings of an even number of a's, since $L^0 = \{\epsilon\}$, $L^1 = \{aa\}$, $L^2 = \{aaaa\}$, and so on.

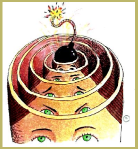


- If L is a language, then the **positive closure** of L, written L^+, is the concatenation of L with itself any number of times but at least once. Formally,

$$L^+ =_{\text{def}} \bigcup_{i=1}^{\infty} L^i$$

  i.e. $L^1 \cup L^2 \cup \ldots$.

  E.g. if $L = \{aa\}$, then L^+ is all non-empty strings of an even number of a's.



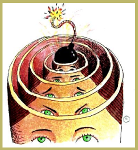
7.5. Recursive Definitions of Languages

- The previous sections showed us one way to define a language (a set of strings), namely by writing an extensional or intensional set definition, possibly making use of operators such as concatenation and closure.
- Another possibility is to give a recursive definition. By way of example, here's a recursive definition (which, in fact, you saw in a previous lecture) of arithmetic expressions. (The alphabet is $\{0, 1, 2, \ldots, 9, +, -, \times, \text{div}, \text{mod}\}$.)

  Base case: Integers
  $$0, 1, 2, 3, \ldots$$
  Recursive case: If E_1 and E_2 are arithmetic expressions then
  $$(E_1 \ \mathbf{op} \ E_2)$$
  is an arithmetic expression, where **op** is one of
  $$+ \qquad - \qquad \times \qquad \text{div} \qquad \text{mod}$$
  Closure: Nothing else.

  We have already noted that this definition is recursive. Recursion is a common feature of the definition of languages.

  The other thing to note is that it is quite cumbersome to define even these very simple expressions using English as our metalanguage. Already we are resorting to abbreviations, such as E_1, E_2 and **op**. A symbol such as **op** is not part of the object language; it's part of the metalanguage. We are using it to define the object language.



Class Exercise

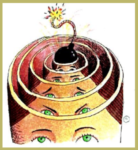
- It's important not to confuse metalanguage with object language. Hence, from the definition, which of these are legal arithmetic expressions?

$$(23 \ \mathbf{op} \ 78)$$

$$(23 \ \mathbf{op} \ (56 \ \mathbf{op} \ 102)$$

$$(E_1 + E_2)$$

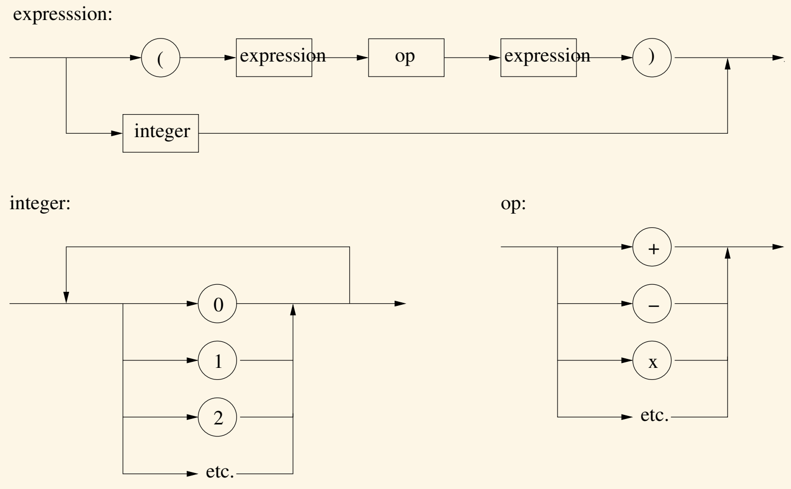
$$(E_1 \times (56 \,\mathrm{div}\, E_2))$$



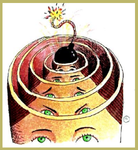
Module Home Page
Title Page
◀◀ ▶▶
◀ ▶
Page 11 of 13
Back
Full Screen
Close
Quit

7.6. Syntax Diagrams

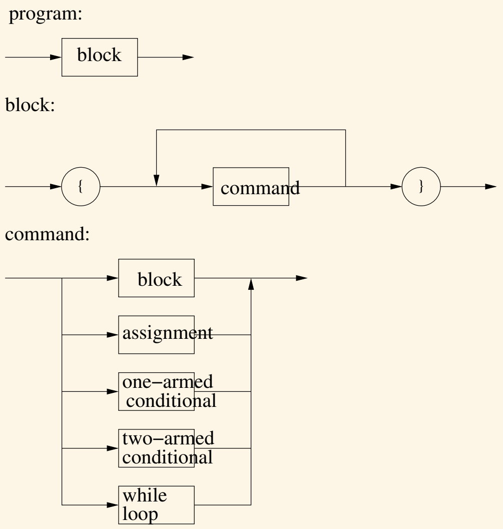
- Syntax diagrams are another way of defining the syntax of a formal language. They are readily-understood because (a) they are pictorial and (b) they keep a clear distinction between metalanguage and object language.
- Our first examples define arithmetic expressions:

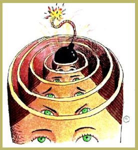


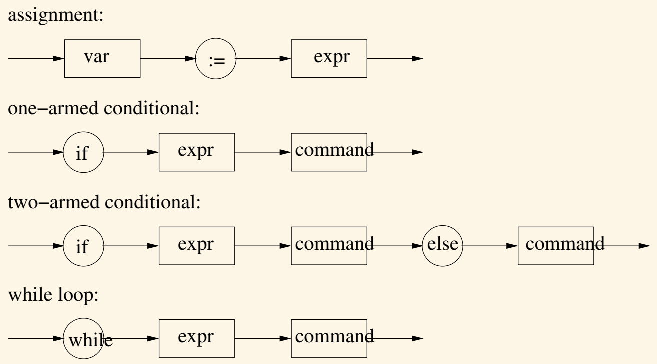
- Each syntax diagram has a name.
- A string of symbols is legal according to some syntax diagram if it can be generated by moving through the diagram from its entry point to its exit point.
- Symbols shown in circles must appear 'as is'. (They must be symbols from our alphabet.)



- A name inside a rectangle is the name of another diagram, and so this diagram must also be traversed.
- When a branch is encountered, multiple paths may need to be checked, but we require only that at least one of them can be successfully traversed.
- Usually, one of the diagrams is regarded as special in the sense that it defines the *start symbol* of the grammar. Here, it would be the diagram labelled ‘expression’.
- Let’s use syntax diagrams to define part of the syntax of a simple programming language that we will call MOCCA. The start symbol is ‘program’. The alphabet is $\{\{, \}, :=,$ **if** , **else** , **while** $\}$.







- That's enough! To finish this off properly, we should define 'var' and 'expr' (variables and expressions). But we won't bother. You get the idea.

Acknowledgements

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.