Handout 0: Syntax

1. Historical background
The study of programming languages borrows ideas from the study of natural languages, called linguistics, as well as the study of logic in mathematics. The first concept we borrow is the division of the study into two disciplines: syntax, which is the study of structure of statements, and semantics, which is the study of meaning of statements. The study of syntax goes back a very long time. Panini in 400 BC gave formal rules clarifying the grammar of Sanskrit. Dionysius Thrax studied the grammar of Greek in 150 BC. In modern times, Emil Post in 19th century devised string manipulation rules by which one define languages in a generative way. (These are called “Post systems” and are equivalent in computational power to Turing machines.) The seminal work in defining formal grammars was done by Noam Chomsky, who developed what is known as the Chomsky hierarchy, a classification of systems of rules with increasing power. Right-linear grammars are equivalent in power to finite state automata. These are used to define the lexical structure of the symbols used in programming languages. Context-free grammars are equivalent in power to push-down automata. They are widely used for describing the syntax of programming languages. Context-sensitive grammars and Unrestricted grammars are higher in the higher hierarchy, with the highest class being equivalent in power to Turing machines.

2. The idea of syntax
A language is a collection of strings, each made up of certain symbols. The first task of syntax is to specify which strings belong to the language. However, languages are not arbitrary collections of strings. There is structure within each string. This structure allows us to specify the syntax of the language in a systematic way, using a collection of rules. The structure of a string can be explicated by drawing a tree, called its syntax tree, showing how the different parts of the string are grouped together. In a sense, the syntax tree is the real element of the language and the string is merely a convenient way to write it down. We should be able to quickly go back and forth between the “string view” and the “tree view” of any given element of the language as if they were the same thing.

3. Inductive definitions
The systems of rules that we use to define languages go by the name of inductive definitions in mathematics. Inductive definitions are systems of rules written in clear English in order to specify what elements belong to the collections of interest to us. Only those elements belong to the collection which can be generated by a finite application of the rules.

4. Example
To introduce inductive definitions, we look at an example language. Consider the language of arithmetic expressions in a programming language, made up of variables, constants, and unary/binary operators. The binary operators we will represent include +, −, *, and /. We will treat the unary negation operator, which we write as ⊖ in order to avoid confusion with the binary − operator. Other than the operators, we have three kinds of entities that we need to deal with. We use a standard mathematical symbol to stand for each kind of an entity, so that we can understand the notations easily.

- Variables, represented by the symbol $x$. (Examples include sum, delta.)
- Constants, represented by the symbol $n$. (Examples include 23 and 365, in fact all integers. We will ignore other forms of numbers.)
- Expressions, represented by the symbol $E$.

Variables and constants are primitive, from our point of view. We will not deal with their structure in any detail. Our interest is in defining the structure of expressions. Here is a collection of rules, which constitute an inductive definition of the class of expressions:

a. Every variable $x$ is an expression.

b. Every constant $n$ is an expression.

c. If $E$ is an expression, then $\ominus E$ is an expression.

d. If $E_1$ and $E_2$ are expressions, then $E_1 + E_2$, $E_1 - E_2$, $E_1 \ast E_2$ and $E_1/E_2$ are expressions.
Consider the example \( \text{prod} + 2 \ast \text{delta} \). It is an expression. Why? \text{prod} is a variable. By virtue of rule (a), it is an expression. 2 is a constant. By virtue of rule (b), it is an expression as well. So, \( \text{prod} + 2 \) is an expression by rule (d). Since \text{delta} is a variable, and hence an expression, we have that \( \text{prod} + 2 \ast \text{delta} \) is an expression by rule (d) again. We can draw this grouping as a tree, which is called the syntax tree of the string. Figuring out how the rules are applied in order to generate the given string is called parsing.

Wait a minute! This seems wrong. When we look at \( \text{prod} + 2 \ast \text{delta} \), we don’t think of \( \text{prod} + 2 \) as being a subexpression. But the above parse claims that it is. Does this mean that our rules are misleading in explaining the structure of expressions? Yes and no.

There is another way to parse the expression \( \text{prod} + 2 \ast \text{delta} \). 2 \( \ast \) \text{delta} is an expression by rule (d). Hence the expression can be obtained by applying rule (d) to \( \text{prod} \) and \( 2 \ast \text{delta} \). This gives another syntax tree for the same string.

5. Ambiguity

If there are multiple ways to parse a given string, we say that the string is ambiguous. Ambiguity is really a concept of semantics, i.e., meanings. But usually meaning is understood by looking at how the parts are grouped together. So, whenever a string has multiple parses, it is very likely that each parse can be understood to have a different meaning.

There are several techniques used to disambiguate a language so that its strings have unique syntax trees:

a. Use brackets to group substrings that are intended to be grouped together. For example, \((\text{prod} + 2) \ast \text{delta}\) and \(\text{prod} + (2 \ast \text{delta})\) are two possible bracketings.

b. Use a left-to-right order for grouping subparts. This means that the string is scanned left to right and, as soon as a group of symbols is noticed that form a subgroup, it is recognized as a subgroup. For example, \(\text{prod} + 2 \ast \text{delta}\) may be parsed left-to-right by noticing that \(\text{prod} + 2\) is an expression which is immediately recognized as such. The remaining symbols \(\ast \text{delta}\) are used later for the remaining parse.

Similarly, a right-to-left order can also be used in some contexts. Such rules for parsing are called associativity rules. We use the terminology that the operators “associate to the left” or “associate to the right” as the case may be. The language definition must state which rules are applicable in which context.

c. Use precedence rules rules that say that some constructs in the language should be given precedence (or “priority”) over other constructs for the purpose of grouping subparts. For example, a widely used convention in the syntax of expressions is that \(\oplus\) has precedence over \(\ast\) and \(/\), which in turn have precedence over \(+\) and \(-\). Using this convention the string \(\text{prod} + 2 \ast \text{delta}\) is always parsed as \(\text{prod} + (2 \ast \text{delta})\). \(\ast\) has precedence over \(+\).

d. Use other rules that might specify the scope of the constructs. For example, consider mathematical expressions that have iterated summation and product operations \(\sum\) and \(\prod\). An example expression would be \(\sum_i f(x_i) + \sum_j g(y_i/x_j)\). A possible convention in defining such a language is to say that the scope of each \(\sum\) or \(\prod\) operator extends as far to the right as possible. So, \(\sum_i f(x_i) + \sum_j g(y_i/x_j)\) is parsed as \(\sum_i(f(x_i) + (\sum_j g(y_i/x_j)))\).

The purpose of all such rules is essentially to reduce the need for the use of brackets, which would otherwise add clutter to the statements in the language.

6. Production rules

Inductive definitions of syntax are often expressed symbolically using the notation of production rules. Collections of such rules are referred to as “grammars”. The production rule version for our example inductive definition for expressions is as follows:

\[
\begin{align*}
E & ::= x \\
E & ::= n \\
E & ::= \oplus E' \\
E & ::= E_1 + E_2 \\
E & ::= E_1 - E_2 \\
E & ::= E_1 \ast E_2 \\
E & ::= E_1/E_2
\end{align*}
\]
Note that we use the symbols $E$, $x$ and $n$ previously agreed to stand for expressions, variables and constants respectively. Each rule can be read in English as stating that the left hand side entity can be of the form of the right hand side. For example, the first rule says “an expression can be a variable $x$”. The last rule says “an expression can be of the form $E_1/E_2$ where $E_1$ and $E_2$ are expressions”. It is also common to group all the production rules for a particular kind of entity into a single rule, using “|” to separate the multiple choices.

$$E ::= x \mid n \mid \ominus E' \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \ast E_2 \mid E_1/E_2$$

7. Inference rules

Another formal notation used for expressing inductive definitions is that of inference rules, as used in the symbolic logic. In this notation, an English statement of the form “if $A_1$ and $A_2$ and \ldots $A_n$ then $B$” is written graphically as:

$$A_1 \quad A_2 \quad \ldots \quad A_n \quad B$$

The statements appearing in the rules are called “judgments”. Those appearing above the horizontal line are called the “antecedents” of the rule and the statement below the line is called the “consequent” of the rule. It is important to note that there can be multiple statements above the line, including the possibility of zero statements. However, there is always a single statement below the horizontal line.

We can write inference rules for our example language of expressions using the judgement forms “$x$ Variable”, “$n$ Constant” and “$E$ Expression”. Here is the gallery of the rules:

- $x$ Variable
- $x$ Expression
- $n$ Constant
- $n$ Expression
- $\ominus E'$ Expression
- $E_1 + E_2$ Expression
- $E_1 - E_2$ Expression
- $E_1 \ast E_2$ Expression
- $E_1/E_2$ Expression

This notation is evidently more verbose than the production rule notation. However, it has the advantage of being more general form of a notation for inductive definitions. So, it is extensible. In particular, it can be used for expressing the context-sensitive aspects of syntax, such as type-checking rules.

8. Inductive proof

In basic mathematics, one has the notion of “mathematical induction,” which is used to prove properties of natural numbers (integers $0, 1, 2 \ldots$). To prove a property $P(n)$ for all natural numbers $n$, one has to prove:

- $P(0)$, and
- $P(k) \Rightarrow P(k + 1)$ for arbitrary natural numbers $k$.

Using this method, we can reduce the problem of proving an infinite number of statements $P(0)$, $P(1)$, $P(2)$, \ldots to just proving two statements! In proving the property for $k + 1$ we can assume that it holds for the smaller number $k$. Such an assumption is often referred to as the “inductive hypothesis.”

This method works because natural numbers have an inductive definition:

- 0 is a natural number, and
- if $k$ is a natural number then $k + 1$ is a natural number.

The moral is that inductive definitions give rise to inductive proof principles.

Every grammar for a language is similarly an inductive definition. So, it can be used as inductive proof principle to prove properties of the strings in the language. This form of induction is called structural induction because one often says “by induction on the structure of \ldots”. Here is the inductive proof schema for our example language of expressions. To prove a property $P(E)$ for all expressions $E$, we have to prove:

- $P(x)$ for all variables $x$,
- $P(n)$ for all constants $n$,
• \( P(E') \Rightarrow P(\oplus E') \),

• \( P(E_1) \land P(E_2) \Rightarrow P(E_1 + E_2), P(E_1) \land P(E_2) \Rightarrow P(E_1 - E_2), P(E_1) \land P(E_2) \Rightarrow P(E_1 \times E_2) \) and \( P(E_1) \land P(E_2) \Rightarrow P(E_1/E_2) \).

9. Example inductive proof

We use structural induction to prove a simple property of the expressions in our example language: The number of operands (variables and constants) in an expression is always 1 more than the number of binary operators in the expression. Or, symbolically:

\[ \# \text{ operands in } E = 1 + \# \text{ binary operators in } E \]

We use the induction schema mentioned above.

- Case \( E = x \): \( \# \) operands in \( x = 1 \) and \( \# \) binary operators in \( x = 0 \). So, the property holds.

- Case \( E = n \): Similar to the previous case.

- Case \( E = \oplus E' \): \( \# \) operands in \( \oplus E' = \# \) operands in \( E' \) and \( \# \) binary operators in \( \oplus E' = \# \) binary operators in \( E' \). So, by inductive hypothesis, which says \( \# \) operands in \( E' = 1 + \# \) binary operators in \( E' \), the conclusion follows: \( \# \) operands in \( \oplus E' = \# \) binary operators in \( \oplus E' \).

- Case \( E = E_1 + E_2 \): \( \# \) operands in \( (E_1 + E_2) = \# \) operands in \( E_1 + \# \) operands in \( E_2 \) and \( \# \) binary operators in \( (E_1 + E_2) = 1 + \# \) binary operators in \( E_1 + \# \) binary operators in \( E_2 \) (because + itself is a binary operator). By inductive hypothesis, \( \# \) operands in \( E_1 = 1 + \# \) binary operators in \( E_1 \) and \( \# \) operands in \( E_2 = 1 + \# \) binary operators in \( E_2 \). Hence, \( \# \) operands in \( (E_1 + E_2) = 2 + \# \) binary operators in \( E_1 \) + \# binary operators in \( E_2 \). The conclusion follows: \( \# \) operands in \( (E_1 + E_2) = 1 + \# \) binary operators in \( (E_1 + E_2) \).

10. Language processors defined by induction

The structure of the strings in a language can also be used to define language processors (i.e., programs that process languages). We define such processors as functions of the strings of the language.

Here is an example. We define a function \( T(E) \) that translates an expression \( E \) into the postfix notation. (Recall that the postfix notation notation for expressions requires the operators to be written after their operands. This is in contrast to the normal infix notation, where the binary operators are written in between their operands.)

\[
\begin{align*}
T(x) &= x \\
T(n) &= n \\
T(\oplus E') &= T(E') \oplus \\
T(E_1 + E_2) &= T(E_1) \oplus T(E_2) + \\
T(E_1 - E_2) &= T(E_1) \oplus T(E_2) - \\
T(E_1 \times E_2) &= T(E_1) \oplus T(E_2) \times \\
T(E_1/E_2) &= T(E_1) \oplus T(E_2) / \\
\end{align*}
\]

We say that this form of a definition is given by induction on the structure of the argument \( E \). You would normally call it a recursive function, which it is. But it is a special case of recursion, well-behaved in certain ways. For example, this form of recursion always terminates.