CHAPTER 1

SYNTAX, VALUES AND TYPES

1.1 Syntax

The syntax of a programming language (PL) is concerned with the form of a program. It considers how expressions, commands, declarations and other constructs are arranged to form a well-formed program. When we learn a new PL, we start by learning its syntax. Note that the syntax must be specified; examples are not enough to show the generality of the PL's syntax. To see this, consider the following code:

```
if n > 0: write (n)
```

Although the example illustrates a way we can use an if-statement, it does not tell us all the possible ways we can use the if-statement. In particular,

- Within the `if` block, can we just have one command or multiple commands?
- Within the `write` function, can we just make use of a variable or use any sort of expression?

Formal specification of syntax

There are two ways to specify the syntax- informal and formal specification. An informal specification is expressed in a natural language (e.g. English), while a formal specification is expressed in a precise notation.

Formal specifications are more precise, usually more concise, while informal specifications tend to be ambiguous, inconsistent or incomplete. However, formal specifications are only accessible to those who are familiar with the notation, and is often difficult to use in practice.

We will illustrate the two types of specification with a while loop in a C-like language.

- Informal specification: A while-command consists of `while`, followed by an expression enclosed in parentheses, followed by a sequential command. A sequential command consists of a sequence of one or more commands, enclosed by `{` and `}`.
- Formal specification, using EBNF:

$$\begin{aligned} \textit{while-command} &= \text{`}\texttt{while}\text{'}\ (\ \textit{expression}\ )\ \textit{sequential-command} \\ \textit{sequential-command} &= \{\textit{command}^{+}\} \end{aligned}$$

There are 3 ways we can specify the syntax of a PL formally:

- Regular expression (RE) are good for specifying syntax of lexical elements of programs (e.g. identifiers, literals and comments).

- Backus Naur Form (BNF) are good for specifying syntax of larger and nested program constructs (e.g. expressions, commands and declarations).
- Extended Backus Naur Form (EBNF) combine RE and BNF, and are good in almost all the cases.

We will illustrate syntax using the PL Calc. It is a very simple calculator language, with:

- variables named `a`, `b`, ..., `z` (i.e. lowercase Roman characters).
- expressions consisting of variables numerals and arithmetic operators.
- assignment (`set`) and output (`put`) commands.

The following is a Calc program:

```
set x = 13
set y = x * (x + 1)
put x
put y / 2
```

Regular expressions

A RE is a type of pattern. That is, every RE matches a set of (possibly infinite) strings. We can use REs for a variety of applications:

- specifying a pattern of strings to be searched for in a text.
- specifying a pattern of filenames to be searched for in a file system.
- specifying the syntax of a PL's lexical elements.

We will look at some examples of REs:

- `M(r|rs|iss)` matches `Mr`, `Mrs` and `Miss`.
- `b(an)`*`a` matches `ba`, `bana`, `banana`, `bananana`, and so on.
- `(x|abc)`* matches `x`, `abc`, `xx`, `xabc`, `abcx`, and so on.

The following are the basic RE notations:

- a literal `xyz` matches the literal `xyz` itself.
- $\mathtt{RE}_1|\mathtt{RE}_2$ matches a string that matches either the regular expression $\mathtt{RE}_1$ or the regular expression $\mathtt{RE}_2$.
- $\mathtt{RE}_1$ $\mathtt{RE}_2$ matches a string that matches the regular expression $\mathtt{RE}_1$ concatenated with a string that matches the regular expression $\mathtt{RE}_2$.
- $\mathtt{RE}^*$ matches a string that is a concatenation of zero of more strings that match the regular expression `RE`.
- `(RE)` matches any string that matches the regular expression `RE` (and is used for grouping and precedence).

There are two further RE notations:

- $\texttt{RE}^?$ matches either the empty string or a string that matches the regular expression RE. Its basic RE notation is $\texttt{RE}|\epsilon$.
- $\texttt{RE}^+$ matches a string that is a concatenation of one or more strings that match the regular expression RE. Its basic RE notation is $\texttt{RE}\ \texttt{RE}^*$.

In terms of Calc, we can use a RE to represent the possible identifiers. It is

$$\mathsf{a}|\mathsf{b}|\ldots|\mathsf{z}.$$

We can also use REs for numeral, which consist of one or more decimal digits, e.g. 5, 12, 20. Its RE is

$$(\mathsf{0}|\mathsf{1}|\ldots|\mathsf{9})^+.$$

If we have a PL where an identifier consists of a sequence of one or more upper-case letters and digits, starting with a letter, then the syntax is given by the following RE

$$(\mathsf{a}|\mathsf{b}|\ldots|\mathsf{z})(\mathsf{a}|\mathsf{b}|\ldots|\mathsf{z}|\mathsf{0}|\mathsf{1}|\ldots|\mathsf{9})^*.$$

REs are used everywhere. For instance, the Unix shell scripting language uses an ad hoc pattern-matching notation in which:

- `[...]` matches any one of the enclosed characters;
- `?` matches any single character;
- `*` matches any string of 0 or more characters.

This is a restricted variant of the RE notation since it lacks choice $\texttt{RE}_1|\texttt{RE}_2$ and the star $\texttt{RE}^*$ in their general form. Some example commands are:

- `print bat.[chp]`, which prints files named either `bat.c`, `bat.h` or `bat.p`
- `print bat.?`, which prints all files whose names are `bat.`, followed by any single character
- `print *.c`, which prints all files whose names end with `.c`

Also, the Unix utility `egrep` uses the full pattern-matching notation, in which the following have their usual meanings: $\texttt{RE}_1|\texttt{RE}_2$, `RE*`, `RE+`, `RE?`. It also provides extensions such as:

- `[...]` which matches any of the enclosed characters and
- `.` which matches any single character.

Some example commands are given below:

- `egrep 'b[aei]t' file`, which finds all lines in `file` that contain `bat`, `bet` and `bit`.
- `egrep 'b.t' file`, which finds all lines in `file` that contain `b`, followed by any character, and then `t`.
- `egrep 'b(an)*a' file`, which finds all lines in `file` that contain `b`, followed by 0 or more occurrences of `an`, followed by `a`.

Moreover, some Java classes also use the full pattern-matching notation, with the same extension as `egrep`. An example is given below.

```java
String s;
if (s.matches("b.t")) {
    // code
}
if (s.matches("b[aei]]t")) {
    // code
}
if (s.matches("b(ab)*a")) {
    // code
}
```

BNF

Although RE have many applications, they are not powerful enough to express the syntax of nested/embedded phrases. For example,

- the nested expression `n * (n + 1)` and
- the nested loop

  ```java
  while (r > 0) {
      m = n;
      n = r;
  }
  ```

cannot be expressed using REs.

To specify the syntax of nested phrases such as expressions and commands, we need a context-free grammar. The grammar of a language is a set of rules specifying how the phrases of that language are formed. Each rule specifies how each phrase may be formed from symbols, such as words and punctuation, and simpler phrases.

We are familiar with grammar in English. For example, the following sentences in English are valid: `I smell a rat` and `the cat sees me`. We refer to the symbols `a`, `cat`, `I`, etc. by terminal symbols. We denote these words using grammatical words, such as *sentence*, *subject*, *object*, etc. These are non-terminal symbols.

To produce sentences in English, we have production rules, some of which are given below:

$$\begin{aligned}
\textit{sentence} &= \textit{subject}\ \ \textit{verb}\ \ \textit{object}\,\text{‘}\texttt{.}\text{’} \\
\textit{subject} &= \text{‘}\texttt{I}\text{’} \mid \text{‘}\texttt{a}\text{’}\ \textit{noun} \mid \text{‘}\texttt{the}\text{’}\ \textit{noun} \\
\textit{object} &= \text{‘}\texttt{me}\text{’} \mid \text{‘}\texttt{a}\text{’}\ \textit{noun} \mid \text{‘}\texttt{the}\text{’}\ \textit{noun} \\
\textit{noun} &= \text{‘}\texttt{cat}\text{’} \mid \text{‘}\texttt{mat}\text{’} \mid \text{‘}\texttt{rat}\text{’} \\
\textit{verb} &= \text{‘}\texttt{see}\text{’} \mid \text{‘}\texttt{sees}\text{’} \mid \text{‘}\texttt{smell}\text{’} \mid \text{‘}\texttt{smells}\text{’}
\end{aligned}$$

We use equality to show precisely what we mean, e.g. by a sentence. We use the pipe $\mid$ to denote different possible values, e.g. a noun can be `cat`, `mat` or `rat`. Using the production rules, we can describe how the sentence `I smell a rat` is structured, using a syntax tree.

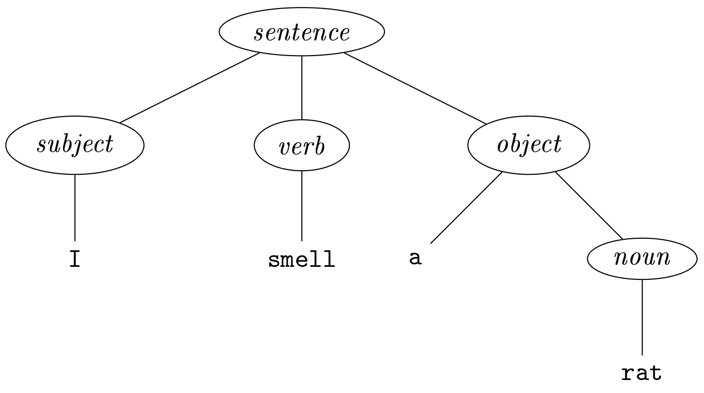

Figure 1.1: The syntax tree for `I smell a rat`

The non-terminal symbols are circled. Note that the terminal symbols are precisely composed of the leaves. Reading left to right allows us to reconstruct the sentence.

Formally, a (context-free) grammar consists of:

- a set of terminal symbols that occur in a sentence;
- a set of non-terminal symbols that define a phrase within a sentence;
- a sentence symbol, which is a non-terminal symbol that stands for a complete sentence; and
- a set of production rules that specify how phrases are composed from terminal symbols and sub-phrases.

Backus Naur Form (BNF) is a notation for expressing a grammar, which we saw above. So, a production rule in BNF is given by

$$N = \alpha,$$

where N is a non-terminal symbol and α is a sequence of terminal and non-terminal symbols. We say that N consists of α, meaning that a value of type N is an instance of the values in α presented in the given order. It is possible that a production rule has several alternatives, e.g.

$$N = \alpha \mid \beta \mid \gamma.$$

The pipe symbol $\mid$ gives the different possibilities for N.

Now, we will characterise the grammar of the PL Calc using BNF. The terminal symbols in Calc are: `put`, `set`, `=`, `+`, `-`, `*`, `(`, `)`, `\n`, `a`, `b`, ..., `z`, `0`, `1`, ..., `9`. The non-terminal symbols in Calc are: prog, com, exp, prim, num, id, where prog is the sentence symbol. The production rules for Calc are given below:

$$
\begin{aligned}
\mathit{prog} &= \mathit{eof} \\
&\mid \mathit{com}\ \mathit{prog} \\
\mathit{com} &= \text{‘}\texttt{put}\text{’}\ \mathit{expr}\ \mathit{eol} \\
&\mid \text{‘}\texttt{set}\text{’}\ \mathit{id}\ \text{‘}\texttt{=}\text{’}\ \mathit{expr}\ \mathit{eol} \\
\mathit{expr} &= \mathit{prim} \\
&\mid \mathit{expr}\ \text{‘}\texttt{+}\text{’}\ \mathit{prim} \\
&\mid \mathit{expr}\ \text{‘}\texttt{-}\text{’}\ \mathit{prim} \\
&\mid \mathit{expr}\ \text{‘}\texttt{*}\text{’}\ \mathit{prim} \\
\mathit{prim} &= \mathit{num} \\
&\mid \mathit{id} \\
&\mid \text{‘}\texttt{(}\text{’}\ \ \mathit{expr}\ \ \text{‘}\texttt{)}\text{’} \\
\mathit{num} &= \mathit{digit} \mid \mathit{num}\ \mathit{digit} \\
\mathit{id} &= \mathit{letter} \\
\mathit{letter} &= \text{‘}\texttt{a}\text{’} \mid \text{‘}\texttt{b}\text{’} \mid \ldots \mid \text{‘}\texttt{z}\text{’} \\
\mathit{digit} &= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{1}\text{’} \mid \ldots \mid \text{‘}\texttt{9}\text{’} \\
\mathit{eol} &= \text{‘}\texttt{\textbackslash n}\text{’}
\end{aligned}
$$

Figure 1.2: The BNF production rules for Calc.

A grammar defines how phrases may be formed from sub-phrases in the language. This is called phrase structure. Every phrase in the language has a syntax tree that explicitly represents in structure.

We will now look at the syntax trees of some code in Calc using the production rules above. The following is the syntax tree for the expression `n + 1`.

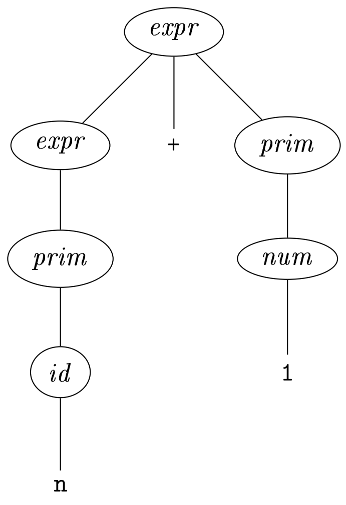

Figure 1.3: The syntax tree for the expression `n + 1`

The following is the syntax tree for the command `put n \n`.

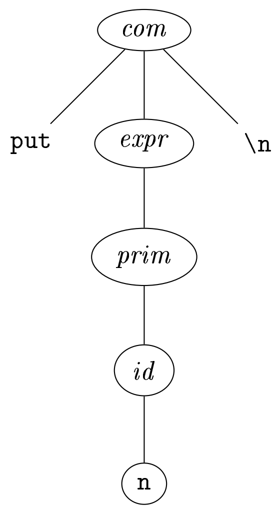

Figure 1.4: The syntax tree for the command `put n \n`

Formally, for a grammar G, the syntax tree of G is a tree where:

- every terminal node is labelled by a terminal symbol of G.
- every non-terminal node is labelled by a non-terminal symbol of G.
- a non-terminal node labelled N has children labelled X, Y, Z (from left to right) if and only if G has a product rule $N = XYZ$ or $N = \ldots|XYZ|\ldots$

Language

If N is a non-terminal symbol of G, a phrase of class N is a string of terminal symbols labelled the terminal symbols labelling the terminal nodes of a syntax whose root node is labelled N. We visit the terminal nodes from left to right. In Calc, some phrases are:

- `x * (22 - y)`, a phrase of class *expr*;
- `set n = 42 \n`, a phrase of class *com*;
- `set n = 42 \n put x * (22 - y) \n`, a phrase of class *prog*.

If S is the sentence symbol of G, a sentence of G is a phrase of class S, e.g. `set n = 42 \n put x * (22 - y) \n` is a sentence of Calc. Finally, the language generated by G is the set of all sentences of G. The language generated by G is typically infinite, even though G is finite.

Our definition of language is syntax-driven since it is defined as a set of sentences. We are also interested in the semantics (i.e. the meaning of each sentence). A grammar does more than generate a set of sentences; it also imposes a phrase structure on each sentence, which we can see from the sentence's syntax tree. Once we know a sentence's phrase structure, we can use it to give a meaning to that sentence.

For example, consider the grammar below:

$$\begin{aligned}
\textit{expr} &= \textit{prim} \\
&\mid \textit{expr}\ \text{‘}\texttt{+}\text{’}\ \textit{prim} \\
&\mid \textit{expr}\ \text{‘}\texttt{-}\text{’}\ \textit{prim} \\
&\mid \textit{expr}\ \text{‘}\texttt{*}\text{’}\ \textit{prim} \\
\textit{prim} &= \textit{num} \\
&\mid \textit{id} \\
&\mid \text{‘}\texttt{(}\text{’}\ \ \textit{expr}\ \ \text{‘}\texttt{)}\text{’} \\
\textit{num} &= \textit{digit} \mid \textit{num}\ \textit{digit} \\
\textit{id} &= \textit{letter} \\
\textit{letter} &= \text{‘}\texttt{a}\text{’} \mid \text{‘}\texttt{b}\text{’} \mid \ldots \mid \text{‘}\texttt{z}\text{’} \\
\textit{digit} &= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{1}\text{’} \mid \ldots \mid \text{‘}\texttt{9}\text{’}
\end{aligned}$$

In this grammar, the operators `+`, `-` and `*` have the same precendence, as we can see in the syntax tree below.

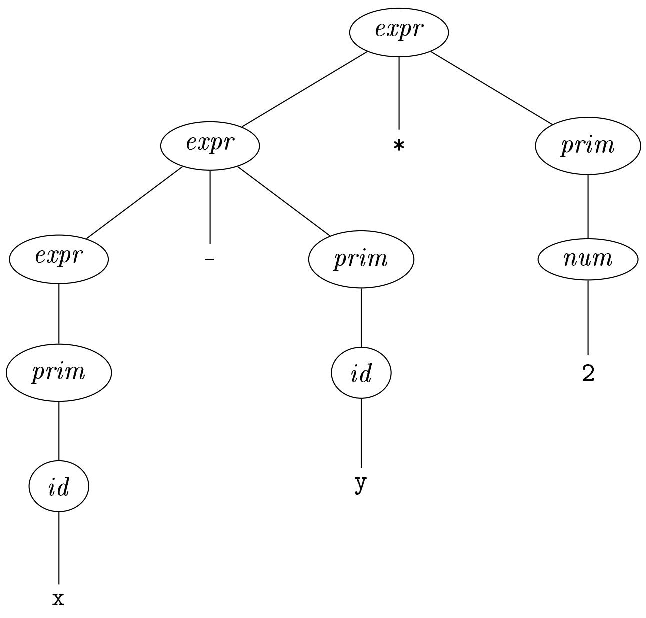

Figure 1.5: The syntax tree for `x - y * 2`

So, the result will be `(x - y) * 2`. This is because we have placed the 3 operations `-`, `*` and `+` at the same level, and the tree is created left to right (i.e. *expr* + *prim* not *prim* + *expr*). We can use brackets to control the evaluation, as we can see in the syntax tree below.

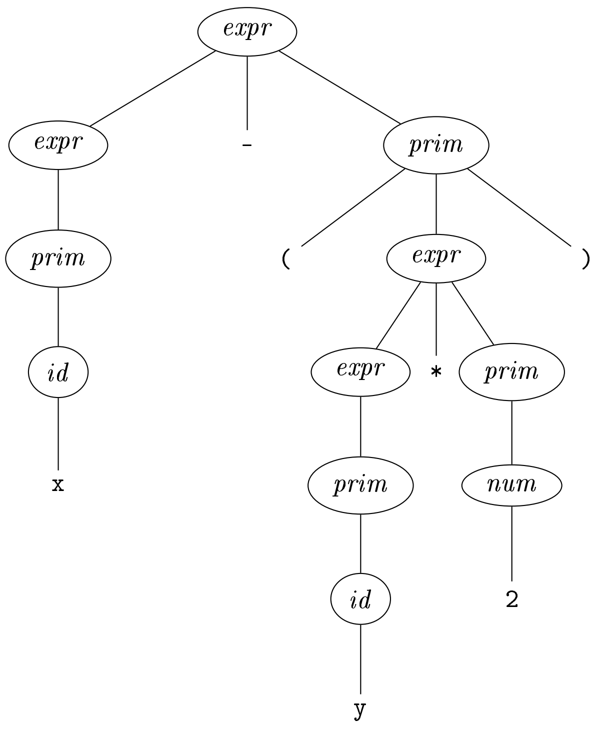

Figure 1.6: The syntax tree for `x - (y * 2)`

Now, consider the following grammar:

$$
\begin{aligned}
\mathit{expr} &= \mathit{term} \\
&\mid \mathit{expr}\ \text{‘}\texttt{+}\text{’}\ \mathit{term} \\
&\mid \mathit{expr}\ \text{‘}\texttt{-}\text{’}\ \mathit{term} \\
\mathit{term} &= \mathit{prim} \\
&\mid \mathit{term}\ \text{‘}\texttt{*}\text{’}\ \mathit{prim} \\
\mathit{prim} &= \mathit{num} \\
&\mid \mathit{id} \\
&\mid \text{‘}\texttt{(}\text{’}\ \ \mathit{expr}\ \ \text{‘}\texttt{)}\text{’} \\
\mathit{num} &= \mathit{digit} \mid \mathit{num}\ \mathit{digit} \\
\mathit{id} &= \mathit{letter} \\
\mathit{letter} &= \text{‘}\texttt{a}\text{’} \mid \text{‘}\texttt{b}\text{’} \mid \ldots \mid \text{‘}\texttt{z}\text{’} \\
\mathit{digit} &= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{1}\text{’} \mid \ldots \mid \text{‘}\texttt{9}\text{’}
\end{aligned}
$$

Within this grammar, we have a higher precedence for `*` than `+` and `-`. This can be seen in the syntax tree below.

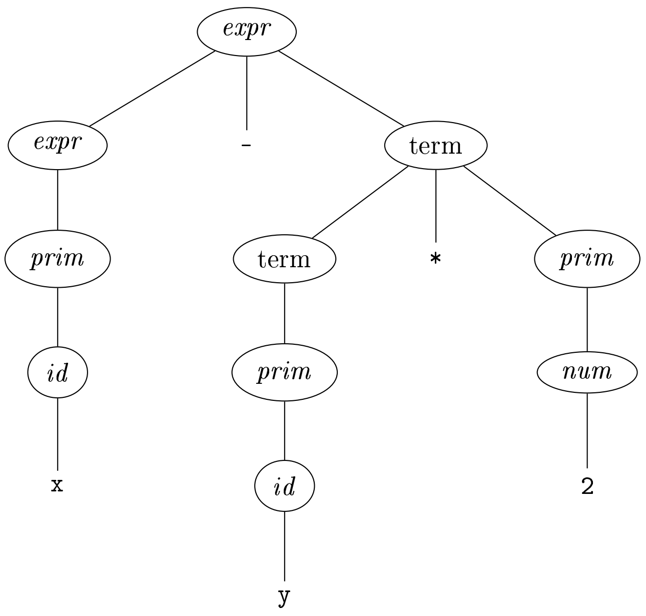

Figure 1.7: The syntax tree for `x - y * 2`

This follows the rules of arithmetic, and we see this grammar in most PLs.

Ambiguity

A phrase is ambiguous if it has more than one syntax tree. Moreover, we say that a grammar is ambiguous if any of its phrases is ambiguous. Natural languages can be ambiguous, but we do not allow the grammar of a PL to be ambiguous so that the meaning of a program is certain.

To illustrate an ambiguous grammar, consider the following grammar.

$$
\begin{aligned}
com = {} & \text{‘}\texttt{put}\text{’}\ \textit{expr} \\
\mid {} & \text{‘}\texttt{if}\text{’}\ \textit{expr}\ \text{‘}\texttt{then}\text{’}\ \textit{com} \\
\mid {} & \text{‘}\texttt{if}\text{’}\ \textit{expr}\ \text{‘}\texttt{then}\text{’}\ \textit{com}\ \text{‘}\texttt{else}\text{’}\ \textit{com} \\
\mid {} & \ldots
\end{aligned}
$$

This grammar is ambiguous since there are 2 productions for `if`. In fact, the phrase `if b then if c then put 1 else put 2` has two syntax trees, as shown below.

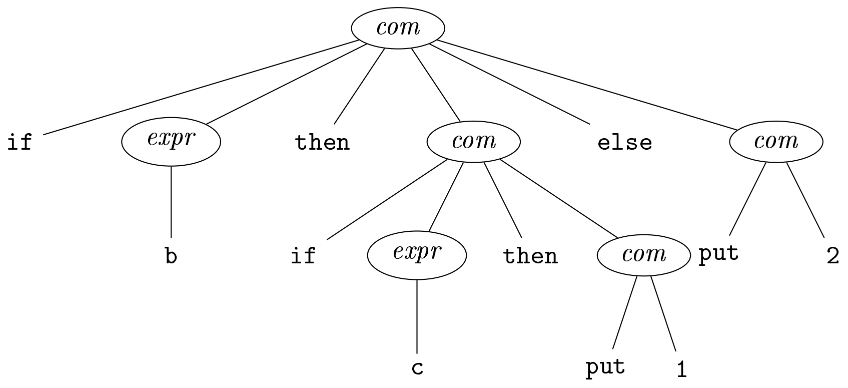

Figure 1.8: A syntax tree for `if b then if c then put 1 else put 2`

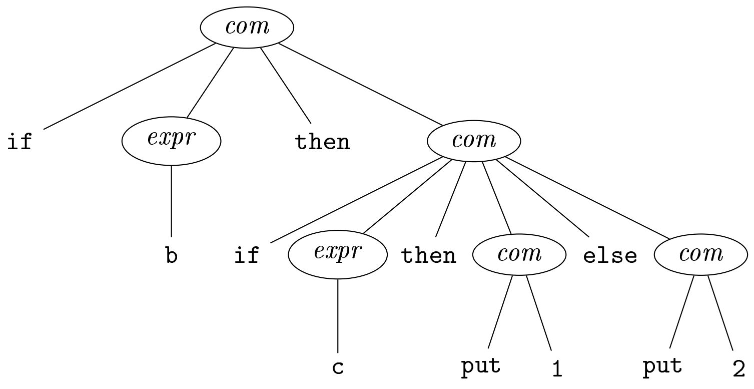

Figure 1.9: A syntax tree for `if b then if c then put 1 else put 2`

A PL should not be ambiguous.

EBNF

Extended Backus Naur Form (EBNF) is a combination of BNF and RE notation. An EBNF production rule has the form $N = \texttt{RE}$, where `RE` is a regular expression, expressed in terms of both terminal and non-terminal symbols. An example of such a rule is given below.

$$\textit{sequential-command} \;=\; \texttt{\{}\ \textit{command}^{+}\ \texttt{\}.}$$

EBNF is convenient for specifying all aspects of syntax.

The EBNF production rules for Calc is given below:

$$
\begin{aligned}
\textit{prog} &= \textit{com}^{*}\ \textit{eof} \\
\textit{com} &= \text{‘}\texttt{put}\text{’}\ \textit{expr}\ \textit{eol} \\
&\mid \text{‘}\texttt{set}\text{’}\ \textit{id}\ \text{‘}\texttt{=}\text{’}\ \textit{expr}\ \textit{eol} \\
\textit{expr} &= \textit{prim}\ (\text{‘}\texttt{+}\text{’}\ \textit{prim} \mid \text{‘}\texttt{-}\text{’}\ \textit{prim} \mid \text{‘}\texttt{*}\text{’}\ \textit{prim})^{*} \\
\textit{prim} &= \textit{num} \\
&\mid \textit{id} \\
&\mid \text{‘}\texttt{(}\text{’}\ \ \textit{expr}\ \ \text{‘}\texttt{)}\text{’} \\
\textit{letter} &= \text{‘}\texttt{a}\text{’} \mid \text{‘}\texttt{b}\text{’} \mid \ldots \mid \text{‘}\texttt{z}\text{’} \\
\textit{digit} &= \text{‘}\texttt{0}\text{’} \mid \text{‘}\texttt{1}\text{’} \mid \ldots \mid \text{‘}\texttt{9}\text{’} \\
\textit{eol} &= \text{‘}\texttt{\textbackslash n}\text{’}
\end{aligned}
$$

Figure 1.10: The EBNF production rules for Calc.

1.2 Values and Types

Values are grouped into types according to the operations that may be performed on them. Different PLs support a huge variety of types, such as integers, floats, booleans, characters, objects, etc. A type is a set of values equipped with operations that can be applied uniformly to all these values, e.g. the type `BOOL` has values $\{\texttt{true}, \texttt{false}\}$ with operations `not`, `and`, `or`. The cardinality of a type T is the number of values of type T, and is denoted by $\#T$, e.g. $\#\texttt{BOOL} = 2$.

A value v of type T is an element $v \in T$. For example, the value `false` is of type `BOOL`, and the value `13` is of type `INT`. An expression E is of type T if its (expected) evaluation results in a value v of type T. For example, if `n` is of type `INT`, then `n - 1` is an expression of type `INT`, and `n > 0` is an expression of type `BOOL`.

Primitive types

A primitive type is a type whose values are primitive, meaning that they cannot be decomposed into smaller values. Every PL has many built-in primitive types. The choice of primitive types is influenced by the PL's intended application area, e.g.

- Fortran (scientific computing) has floating point numbers,
- Cobol (data processing) has fixed-length strings,
- C (system programming) has bytes and pointers.

Some PLs allow programmers to define new primitive types.

Below are some common built-in primitive types:

$$\begin{aligned}
\texttt{VOID} &= \{\texttt{void}\} \\
\texttt{BOOL} &= \{\texttt{true}, \texttt{false}\} \\
\texttt{CHAR} &= \{\texttt{A}, \dots, \texttt{Z}, \dots\} \\
\texttt{INT} &= \{\texttt{-m}, \dots, \texttt{-1}, \texttt{0}, \texttt{1}, \dots, \texttt{m-1}\} \\
\texttt{FLOAT} &= \{\dots, \texttt{0.0}, \dots\}.
\end{aligned}$$

The types `CHAR`, `INT` and `FLOAT` are present in most PLs, but there are some differences in their implementation depending on the PL/implementation. Now, we consider the cardinalities of the sets:

- $\#\texttt{VOID} = 1$,
- $\#\texttt{BOOL} = 2$,
- $\#\texttt{CHAR} = 128, 256, 32768, \dots$ (depends on the language),
- $\#\texttt{INT} = 2m$.

The definition of primitive types depends on the PL, e.g. in C, booleans, characters and enums are just (small) numbers.

Composite types

A composite type is a type whose values are composite. This means that these values can be decomposed into simpler values. In a programming language, there are 4 fundamental ways of generating composite types from primitive/composite types:

- Cartesian products, which gives rise to tuples, structs and records;
- disjoint unions, which give algebraic types, variant records and objects;
- mappings, which give arrays and functions;
- recursive types, which give rise to lists and trees.

Cartesian product

In a Cartesian product, values of two (or more) types are grouped into pairs (or tuples). Formally, for types S and T, the product type $S \times T$ has values of the form (s, t), where s is a value of type S and t is a value of type t. In other words,

$$S \times T = \{(s, t) \mid s \in S, t \in T\}.$$

The cardinality of the Cartesian product type is given by $\#(S \times T) = \#S \times \#T$. We can generalise from pairs to tuples:

$$S_1 \times \cdots \times S_n = \{(s_1, \ldots, s_n) \mid s_1 \in S_1, \ldots, s_n \in S_n\}.$$

There are 2 basic operations we can perform on tuples:

- we can construct a tuple given its components;
- we can select a specific component from a tuple.

Pascal records, C structs and Haskell tuples can all be understood in terms of cartesian products. Python tuples are somewhat different from normal tuples. This is because they can be indexed like arrays. The following shows the tuple operations on C structs:

```c
enum Month {
    JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC};
struct Date {Month m; int d;}

struct Date date1 = {JAN, 1};
printf ("%d/%d", date1.d, date1.m+1);
```

As an enumerated type, `Month` has values 0 to 11. We construct a struct by `{JAN, 1}`, and select the components by `date1.d` and `date1.m`. The possible values of struct type are

$$\texttt{MONTH} \times \texttt{INT} = \{0, 1, \ldots, 11\} \times \{\ldots, -1, 0, 1, \ldots\}.$$

Disjoint union

In a disjoint union, a value is chosen from one of two (or more) types. So, the union type $S + T$ is the type of disjoint-union values. Each disjoint-union value consists of a variant (chosen from either type S or type T), along with a tag:

$$S + T = \{\texttt{left}\ \ s \mid x \in S\} \cup \{\texttt{right}\ \ t \mid t \in T\}.$$

The value `left` s consists of the tag `left` and a variant $s \in S$. Similarly, the value `right` t consists of the tag `right` and a variant $t \in T$. We can write `left` $S +$ `right` T to make the tags explicit. The cardinality of the disjoint union type $\#(S + T) = \#S + \#T$. We can generate to disjoint unions with multiple variants: $T_1 + T_2 + \cdots + T_n$.

There are 3 basic operations we can perform on disjoint-union values:

- construction of a disjoint-union value from its tag and variant;
- tag test, to inspect the tag of the disjoint-union value;
- projection, to recover a specific variant of a disjoint-union value (e.g. its T_1 variant)- trying to recover the wrong variant will fail.

Haskell algebraic types, Pascal/Ada variant records and Java objects can be understood in terms of disjoint unions. The following shows the disjoint-union operations on Java objects:

```java
class Point {
    Point () {
        // code
    }
    // code
}

class Circle extends Point {
    int r;
    Circle (int r) {
        this.r = r;
    }
    // code
}

class Box extends Point {
    int w;
    int h;
    Box (int w, int h) {
        this.w = w;
        this.h = h;
    }
}
```

So, the set of objects in Java is:

$$\texttt{OBJECT} = \cdots + \texttt{Point VOID} + \texttt{Circle INT} + \texttt{Box (INT} \times \texttt{INT)} + \ldots$$

The other objects are library classes and the other declared classes. We have objects: `Point VOID`, `Circle 5`, `Box (3, 8)`, etc. Each object's tag identifies its class.

The operations we can perform on these Java classes are given below:

```java
1 Circle c = new Circle(5);
2 Box b = new Box(3, 4);
3
4 Point p;
5 if (p instanceof Circle) {
6     int radius = ((Circle) p).r;
7     // code
8 }
```

We construct objects in lines 1 and 2. In line 5, we perform a tag test- `p instanceof Circle`. In line 5, we project a point into a circle- `(Circle) p`. The set of objects in a Java program is open-ended. Initially, the set contains objects of library classes (non-abstract). Subsequently, the set is augmented by each declared class (non-abstract). We exclude abstract classes since it is not possible to instantiate those.

Mapping types

A map $m : S \to T$ maps a value in S to a value in T. We denote $y = m(x)$, and call y the image of x under m. The set $S \to T$ is the type of all mappings $m : S \to T$.

$$S \to T = \{m \mid x \in S \implies m(x) \in T\}.$$

The cardinality of the mapping type $\#(S \to T) = (\#T)^S$. This is because, for each $s \in S$, there are $\#T$ possible choices for $m(s)$.

For example, consider the mapping type:

$$\{u, v\} \to \{a, b, c\}.$$

Its cardinality is 9, and the possible mappings are:

$$\begin{array}{lll} \{u \to a, v \to a\} & \{u \to a, v \to b\} & \{u \to a, v \to c\} \\ \{u \to b, v \to a\} & \{u \to b, v \to b\} & \{u \to b, v \to c\} \\ \{u \to c, v \to a\} & \{u \to c, v \to b\} & \{u \to c, v \to c\}. \end{array}$$

Arrays can be understood as mappings. Assume that we have an array with components of type T and index values of type S. Then, for every possible index in S, there exists a component t of type T. This gives a natural representation of the array as the mapping type $S \to T$. The basic operations we can do on arrays are:

- construction of an array from its components; and
- indexing, to select a component using a computed index value.

An array is a finite mapping. If an array is of type $S \to T$, S must be a finite range of consecutive values (starting from a lower bound and going to the upper bound, incrementing by 1 every time), called the array's index range. In some PLs, the index may be any range of integers. However, the index range is mostly $\{0, 1, \ldots, n - 1\}$, for some fixed n.

We now consider C arrays:

```c
enum Pixel {DARK, LIGHT};
typedef Pixel[] Row;

```

```
4 Row r = {DARK, LIGHT, LIGHT, DARK}
5 int i, j;
6 r[i] = r[j];
```

We construct the array `r` in line 4, and index `r[j]` in line 6. The values of this array type are:

$$\texttt{ROW} = \{0, 1, 2, \dots\} \to \texttt{PIXEL} = \{0, 1, 2, \dots, \} \to \{0, 1\}.$$

Functions can also be understood as mappings. They map arguments to results. For example, let f be a unary function whose argument is of type S and result is of type T. Then, f is of type $S \to T$. There are 2 operations on functions:

- construction (or definition) of a function.
- application, i.e. calling the function with an argument.

A function can represent an infinite mapping (if $\#S = \infty$). This is not an issue since the results are computed on demand. On the other hand, this is an issue for an array since all the components within the array are stored.

A binary function f has arguments of type S_1 and S_2, and result of type T. In most PLs, we view f as mapping a single pair of arguments (i.e. $S_1 \times S_2$) to a result. So, the function $f : (S_1 \times S_2) \to T$. Then, we can think of f as a unary function. This can be generalised to n-ary functions, i.e. $f : (S_1 \times \cdots \times S_n) \to T$.

Now, consider the following unary function in C:

```
int length (String s) {
    int n = 0;
    while (s[n] != NUL) {
        n++;
    }
    return n;
}
```

This function's type is $\texttt{STRING} \to \texttt{INT}$. The value of the function is an infinite mapping since there are infinite possible strings. Next, consider the following binary function in C:

```
String rep (int n, char c) {
    String s = malloc((n+1) * sizeof(char));
    for (int i = 0; i < n; i++) {
        s[i] = c;
    }
    s[n] = NUL;
    return s;
}
```

This function has type $(\texttt{INT} \times \texttt{CHAR}) \to \texttt{STRING}$. In a call, the function is applied to a pair, e.g. $(6, \texttt{!})$ yields `!!!!!!`.

Recursive types

A recursive type is one defined in terms of itself. It must be a disjoint-union type in which:

- at least one variant is recursive (the recursive type) and

- at least one variant is non-recursive (the base type).

The types `LIST` and `TREE` can be given as recursive types:

$$\texttt{LIST} = \texttt{VOID} + (\texttt{VALUE} \times \texttt{LIST}), \qquad \texttt{TREE} = \texttt{VOID} + (\texttt{VALUE} \times \texttt{TREE} \times \texttt{TREE}).$$

The cardinality of a recursive type T is always infinite, i.e. $\#T = \infty$. We can produce a value of type T for an arbitrarily large size by taking higher numbers of the recursive case.

A list is a sequence of 0 or more component values. It can be empty or non-empty. If it is not empty, then it can be thought of as containing a head (the first value), and a tail (everything else in the list, as another list). This leads to the recursive relation

$$\texttt{LIST} = \texttt{empty VOID} + \texttt{nonempty (VALUE} \times \texttt{LIST}).$$

For example, consider the following class declaration for integer lists in Java:

```java
class IntList {
    int head;
    IntList tail;
}
```

The non-recursive variant is the built-in `null` value.

A string is a sequence of 0 or more characters. The way they are treated depends on the PL:

- Python treats strings as primitive.
- Haskell treats strings as lists of characters, which gives them all the list operations (e.g. getting the head and the tail).
- C treats strings as an array of characters, which gives them all the array operations (e.g. indexing).
- Java treats strings as objects of class `String`.

Type systems

A type error occurs when a program performs a meaningless operation, e.g. adding a string to a boolean. A PL's type system groups values into types to help prevent type errors. It also allows programmers to describe data effectively. Possession of a type system distinguishes high-level PLs from low-level languages. In assembly/machine languages, the only 'types' are bytes, so meaningless operations cannot be prevented.

Before any operation is performed, its operands must be type-checked in order to prevent a type error. For example,

- in a not operation, we must check that the operand is a boolean;
- in an add operation, we must check that both operands are numbers;
- in an indexing operation, we must check that the left operand is an array and the right operand is an integer.

We can classify a PL with respect to its type system. There are 2 types of type systems- static and dynamic typing. In a statically typed PL:

- every variable has a fixed type (which is usually described by the programmer),
- every expression has a fixed type (which is usually inferred by the compiler), and
- all operands are type-checked at compile-time.

Some statically typed languages are: Pascal, Ada, C, Java and Haskell. On the other hand, in a dynamically typed PL:

- only values have fixed types,
- variables and expressions do not have fixed types, and
- operands must be type-checked when they are computed at run-time.

Some dynamically typed languages are: Smalltalk, Lisp, Prolog and (most) scripting languages such as Perl and Python.

Consider the following code in Java:

```
1  public static boolean even (int n) {
2      return n % 2 == 0;
3  }
4
5  public static void main(String[] args) {
6      int p;
7      if (even(p+1)) {
8          // code
9      } else {
10         // code
11     }
12 }
```

When compiling lines 1-3, the compiler doesn't know the value of `n`, but does know that its type is `INT`. Using this information, it knows that the expression in line 2 will be a `BOOL`. Similarly, in line 7, the compiler does not know the value of `p+1`, but it does know that its type is `INT`. This matches the parameter type for the function `even`, so does not cause a type error. Clearly, we do not need to know the values of the variables to ensure that there won't be any type errors at runtime.

Now, consider the same code in Python:

```
def even(n):
    return n % 2 == 0
```

Here, the type of the variable `n` is not known. So, the operation `%` must be protected by a run-time type check. This is because we do not declare the types of variables in Python, so the compiler cannot infer them in general. So, we need run-time type checks to detect type errors.

Using a dynamically typed language, we can make use of the same function declaration for different types. To see this, consider the following code in Python.

```python
# Return the minimum element of values
def minimum(values):
    min_val = values[0]
    for val in values:
        if val < min_val:
            min_val = val
    return min_val
```

Here, the parameter `value` can be a list, or tuple, and its components can be string, int or float. So, we can call the function `minimum` with the tuple `(3.0, 2.7, 4.1)`, list of integers `[2, 3, 5, 7]` or a list of string `[cat, dog, ant]`.

There are advantages and disadvantages to both static and dynamic typing:

- Static typing is more efficient. It only requires compile-time type checks. On the other hand, dynamic typing requires run-time type checks, which makes the program to run slower. Moreover, we also need to tag all the values, which uses more space.
- Static typing is more secure. The compiler can guarantee that the object program contains no type errors. Dynamic typing provides no such security.
- Static typing is less flexible. Certain computations cannot be expressed naturally. Dynamic typing is more natural when processing data whose types are not known at compile-time.

Expressions

An expression is a program construct that will be evaluated to yield a value. Simple expressions are literals and variables. Compound expressions are given below:

- A function call is an expression that computes a result by applying a function to argument(s).
- A construction is an expression that constructs a composite value from its components.
- A conditional expression is an expression that chooses one of its sub-expressions to evaluate.
- An iterative expression is an expression that performs a computation over a collection (e.g. an array or a list).
- A block expression is an expression that contains declarations of local variables.