Lecture 1: Terms

In this lecture, we formally define what a *term* is in arithmetic: the sort of thing that expressions like $2x + 3$ represent. Our motivation as Computer Scientists is to define this formally enough that we could get a computer to work correctly with terms once we have learnt some more programming.

For example, looking at the C compiler's assembly output for a simple function, with optimisation[1] turned on:

```
int f(int x) {
    int y = 1 + 2 * x;          lea eax, [rdi + 1 + rdi]
    return y;                   ret
}
```

The calling convention on 64-bit Intel machines is that function arguments are passed in the `rdi` register and return values are passed in the `eax` register; the calculation that the compiler is doing here is effectively $x + 1 + x$ in a way that Intel processors have convenient hardware support for to do the calculation and store the result in the right place using a single assembly instruction. To do this, the compiler implementer must know among other things:

- The code `1 + 2 * x` means $1 + (2x)$ not $(1 + 2)x$.
- $2x$ is the same as $x + x$, and $x + 1$ is the same as $1 + x$.
- All this still holds for 64-bit unsigned integers (that wrap around when you reach 2^{64}).

Syntax of Terms

In computing, *syntax* refers to rules for creating and manipulating formally defined objects such as terms or programs.

First, we define the kinds of numbers we might want to build terms over:

Definition 1 (number sets)**.** We use the following notation for number sets[a]:

- The set $\mathbb{N}$, called the natural numbers, is the set $\{0, 1, 2, \ldots\}$.
- The set $\mathbb{Z}$, called the integers[b], is the set $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.
- The set $\mathbb{Q}$ is the set of all fractions[c], that is numbers of the form a/b where a, b are integers and $b \neq 0$. This includes the integers, as b can be 1.
- The set $\mathbb{R}$ is the set of all real numbers (the 'number line').
- The set $\mathbb{Z}_n$ for a positive natural number n is the set $\{0, 1, \ldots, n-1\}$ of all natural numbers strictly less than n. For example, $\mathbb{Z}_{2^{64}}$ is the set of numbers you can store in an `uint64` or unsigned 64-bit integer data type on a computer.

[a] We will explain in more detail what a set is in a later lecture.
[b] Named after the German word for numbers, *Zahlen*.
[c] Named after the word 'quotient'. Also, e.g. $1/2$ and $2/4$ are the same fraction.

If you read older mathematics books, some include 0 among the natural numbers and some do not; in this unit and in most modern books, the natural numbers always include 0.

We will make a difference from most other books by distinguishing between a *term* and a *term string*: the former is an abstract mathematical object, that one could implement in a programming

[1] To be precise, gcc 13 on x86-64 with `-O1`.

language as a class or data type; the latter is a string of symbols that one can write on paper or in a source code file. Thus, for us, $1 + 2x$ is not a term itself but a term string, that one can turn into a term by *parsing*.

Definition 2 (term)**.** We start with the following building blocks:

- A set of numbers (also called values).
- A set of variables (such as the lowercase letters).
- A set of one-input-one-output operations, also called *unary* operations, such as negation (the thing that turns 2 into (-2)).
- A set of two-input-one-output operations, also called *binary* operations. For example, $+, -, \times, \div$.

A term is anything you can build by applying the following rules a finite number of times:

R1. For any value in our set of numbers, we can build a value term.
For example, over the natural numbers, we can build a value term for the number 2.
R2. For any variable, we can build a variable term. For example, x.
R3. For any unary operation and any term T that we have already built, we can build a unary operation term.
R4. For any binary operation, and for any two terms T, U that we have already built, we can build a binary operation term.

If we draw value and variable terms with circles, and operation terms with rectangles and lines to the terms we are building them from, then we can draw terms as trees[2], for example:

In an object-oriented programming language, we could define a type (interface, trait, or class) for terms, and then implement value, variable and operation terms as subclasses. In a functional programming language, we could define terms as an algebraic data type (also known as a sum type) with patterns for value, variable and operation terms. With some experience, a programmer can translate the above definition of terms almost mechanically to an implementation using classes or other data types.

Syntax Rules

Along with rules for building terms, we might want to give rules for manipulating terms. For example, some of the usual rules of arithmetic, expressed over terms, are:

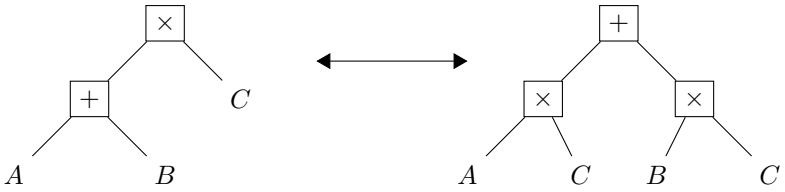
1. If you have an operation term with the $+$ or $\times$ operation, and its two child terms (that is, the terms it was constructed from using Rule R4) are A and B, then you can replace this term with

[2] In the mathematical sense, that is graphs with no loops.

a new term with the same operation, but the children in opposite order. In other words, you can swap out the following, whatever the child terms A and B are:

which we normally express as the commutative law $A + B = B + A$.

2. If you have an operation term with the subtraction operation '$-$' and two children A, B, and both children are 'the same', then you can replace the whole term with a value term with the value 0. We normally express this as $A - A = 0$.
3. If you have a multiplication term whose first child is an addition term, you can replace it with an addition term with two children, both of which are multiplication terms, and whose first children are the children of the former addition term, and whose second children are copies of the second child of the former multiplication term.
   In words, this is about as clear as mud, but as a picture:



this represents the distributive rule of arithmetic that $(A + B) \times C = (A \times C) + (B \times C)$.

To summarise, the *syntax* of terms includes rules for building terms — some of which build new terms out of other terms — and possibly rules for manipulating terms. From a purely syntax point of view, these are simply lists of rules, not yet linked to any deeper meaning.

Definition 3 (syntax rule)**.** A syntax rule is an expression of the form $T \vdash U$ where T, U are terms. The syntax rule itself is not a term. You read it 'T derives U' (or 'T yields U').

A syntax rule says that whenever you have a term matching the pattern of T (that is, you assign a term to each variable in T), then you can replace it with a term matching consistently the pattern of U. This holds even if T is a subterm of a larger term.

In a system with syntax rules, we also write $T \vdash U$ to mean 'You can turn T into U using the rules provided.' This always includes the special case $T \vdash T$ as you can turn a term into itself by applying none[a] of the rules.

[a] To mathematicians, 'applying no rules is an example of applying rules' is a standard way of thinking. For computer scientists, think of the reasoning like this: suppose there is some program that takes a term T and a list of steps, and applies all steps in the list in order. $T \vdash U$ means there is some list that turns T into U with this program. If T is already the same as U, then you can simply call the program with T and the empty list as input, and you will get T back again.

For example, the commutative law of addition as a syntax rule is $a+b \vdash b+a$ where a, b are variables. This means that in a term like $3 \times (2 + y)$, by matching the pattern $a = 2, b = y$ you can replace the original term with $3 \times (y + 2)$ as this is consistent with the same variable matching.

For another example, $x - x \vdash 0$ lets you match $5 + (4z - 4z)$ with $x = 4z$ to get $5 + 0$. There are no variables left to match on the right-hand side of the rule, which is not a problem.

A syntax rule is just a rule saying that you can replace one kind of term with another kind in a particular formal system, it is not a statement that the two terms involved are somehow related (but more on that later on).

A sequence of transformations of a term using syntax rules is called a *derivation*. In a derivation, you can use the symbol $\vdash$ again to mean 'I used a rule here'. For example, with the rules

$$a + b \vdash b + a \qquad (a + b) + c \vdash a + (b + c) \qquad a + (b + c) \vdash (a + b) + c$$

we could make the following derivation:

$$\begin{aligned} & ((3 + 4) + 7) + 6 \\ \vdash \quad & (3 + (4 + 7)) + 6 \\ \vdash \quad & (3 + (7 + 4)) + 6 \\ \vdash \quad & ((3 + 7) + 4) + 6 \\ \vdash \quad & (3 + 7) + (4 + 6) \end{aligned}$$

which, if you remember your number bonds that add up to 10, will help you add the numbers more quickly. If you write out the terms as trees, what you are doing here is rebuilding and reordering the trees.

Term Equality

There are at least three different ways in which terms can be 'the same', and we sneaked this concept into one of the rules mentioned above ($A - B = 0$ if A, B are 'the same'). We define two of these ways for now:

Definition 4 (identical terms)**.** Two terms A, B are *identical* if they are built from the same rules, in the same order, with the same values and variables. In this case we write $A = B$.

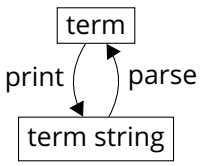
This says more or less the same as, if we get a computer to draw the trees of the two terms, then it creates the same image both times.

Definition 5 (syntactic equivalence)**.** If we have a definition of terms with a list of syntax rules, then two terms A, B are *syntactically equivalent* under these rules if we can turn A into B and back again using these rules, that is $A \vdash B$ and $B \vdash A$. In this case we write $A \mathrel{\widehat{=}} B$.

Note, this includes the special case that every term is syntactically equivalent to itself ($A \mathrel{\widehat{=}} A$).

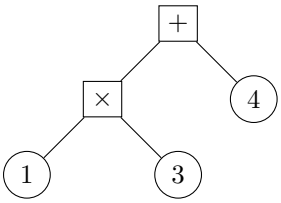
Printing and Parsing

We said that a term is an abstract mathematical object, that one can draw as a tree on paper or implement with classes or other data types in a programming language; a *term string* is a representation as a string of symbols such as $1 + 2 \times x$. Turning a term into a term string is called *printing*, and the other way round is *parsing*:

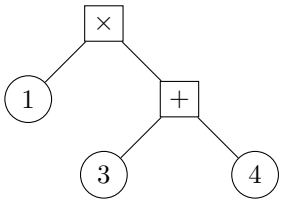


Every term can be printed, but not every string parses to a valid term. There are in fact several different sensible ways to print a term. If we use the usual *infix* notation, that is operators go be-

tween their operands as in $1+2$, then we also need to talk about brackets[3]. Brackets are not part of terms-as-mathematical-objects themselves, we don't need them as the tree structure makes clear what order a term is built in. But we do sometimes need them when printing terms:



$1 \times 3 + 4$



$1 \times (3 + 4)$

To be able to represent both these trees as a one-dimensional string, we need some way of distinguishing them; with the usual convention of BODMAS (division/multiplication before addition/subtraction, among other things) the unbracketed term refers to the tree on the left, and we need brackets to refer to the one on the right.

A similar problem occurs if the tree has the same structure as in the last diagram, but both nodes are addition nodes. We know in some sense that this does not matter for calculating, as addition is associative, but the two terms are not identical as mathematical objects. If the operation is subtraction instead of addition in both cases, you get different results in both cases again.

As a first attempt at printing terms, we can define the following rules that add brackets around every operation. The four rules for printing match the order and structure of the four rules for building terms in Definition 2.

Definition 6 (printing fully bracketed infix terms)**.**
To print a term in fully bracketed infix notation,

I1. If the term is a value term, print the value.
I2. If the term is a variable term, print the variable name.
I3. If the term is a unary operation term built from a term T, print an opening bracket, then the operation symbol, then print the term T, then print a closing bracket.
(This has the effect that a unary negation term, built from the value term 2, gets printed as `(-2)` as you would expect.)
I4. If the term is a binary operation term built from terms T, U then
   (a) First, print an opening bracket.
   (b) Then, print the term T.
   (c) Then, print the operation symbol.
   (d) Then, print the term U.
   (e) Finally, print a closing bracket.

A more interesting question (and an exercise for the workshop) is to adapt this printing algorithm to only print brackets when necessary. Assuming the only binary operations are $+, -, \times, \div$, this is not too hard, but requires some thought to get right.

We can also define rules for checking whether a string is a valid term string in fully bracketed infix notation. These rules directly lead to an algorithm for parsing fully bracketed terms:

[3]To be really precise, 'parentheses'. Bracket is a general term that can refer to round, square, angled, curly and other types. But we will not be pedantic about this point.

Definition 7 (checking fully bracketed infix terms)**.**
A string is a valid term string for fully bracketed terms in infix notation if it can be built from the following rules:

F1. A string representing a value is a valid term string
(for a natural number, this might be 'a non-empty sequence of digits').
F2. A string representing a variable name is a valid term string.
F3. If T is a valid term string, and $*$ is a unary operator symbol, then the string containing an opening bracket, the operator symbol, the string T and a closing bracket in this order is a valid term string. Written out, this string looks like `(*T)` where the $*$ and `T` are replaced with the operator symbol and the term string T, respectively. For example, (-2).
F4. If T and U are valid term strings, and $*$ is a binary operator symbol, then the string `(T*U)` is a valid term string (with everything inside the brackets replaced suitably).

The parsing algorithm this produces reads a string character by character, left to right, and builds up a term as it goes along. (We can allow spaces around operators and brackets if we want, the parsing algorithm can just skip these.) The general parsing rules are, assuming variable names cannot start with digits,

- If you see a digit, try and read a value term.
- If you see a letter, try and read a variable name.
- If you see an opening bracket followed by an operator symbol, try and parse a new term starting with the character after the operator symbol, then try and read a closing bracket. If this succeeds, make a unary operator term with the new term you read as its child term.
- If you see an opening bracket followed by something that is not an operator symbol, try and read a first term, then an operator symbol, then a second term, then a closing bracket. If this succeeds, make a binary operator term whose children are the two terms you read in the process.
- If any of these steps ever fails, or you read something else (like a closing bracket when you were not looking for one) then the string is not a valid term string.

This works, but is not as elegant as it could be: your C compiler will accept `1+2*x` and doesn't need you to write `(1+(2*x))`. We could try and change the rules to read as follows, moving brackets out to a separate rule:

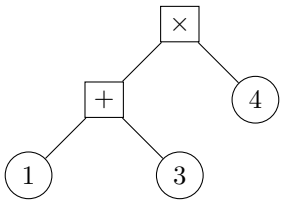
1. A string representing a value is a valid term string (for an integer, this might be 'a non-empty sequence of digits with an optional leading minus sign').
2. A string representing a variable name is a valid term string.
3. A unary operator symbol, followed by a valid term string, is also a valid term string.
4. A valid term string, followed by a binary operator symbol, followed by another valid term string, is also a valid term string. (This deals with the string `2+3` for example.)
5. An opening bracket, followed by a valid term string, followed by a closing bracket, is also a valid term string.

This is an adequate definition of valid term strings in infix notation, but it is not 'structural' in the sense that if you try and write a parser based directly on this definition, it will not always give you the correct term. For example, it would parse $1 + 2 \times 5$ the same way as $(1 + 2) \times 5$ because there is nothing in the rules to stop you from always reading left-to-right. This is ugly.

How to implement a proper parser for infix terms is a bigger question than we can address in this unit, and you might want to wait until you have a year's programming experience to understand the subtleties involved. A good presentation of the answer can be found in the free book 'Crafting Interpreters[4]' by Robert Nystrom.

Reverse Polish Notation (RPN)

There are different ways we can write the same term as a string. This is one reason that we make a distinction between the term itself as an abstract mathematical object, and a term string. For example, for this term



different ways of writing it include:

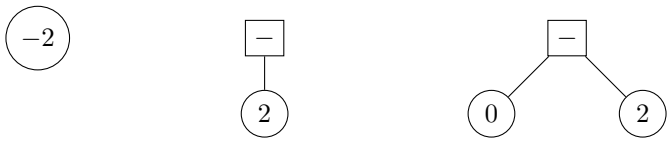
- Infix: $(1 + 3) \times 4$. This notation has the advantage that we are familiar with it from school, but it requires rules like BODMAS and brackets to be able to represent any possible term, and writing a parser for it is more complicated than with some other notations.
- Fully bracketed infix: $((1 + 3) \times 4)$. Solves the parsing problem, but is less human-friendly.
- Function notation: just like we write $f(x)$ or $g(x, y)$ for functions with the function name before the brackets, we could write this term as $\times(+(1, 3), 4)$. More or less the same advantages and disadvantages to fully bracketed infix for computers, but will make most humans hate you.
- S-expressions: $(\times(+1\ 3)\ 4)$ this time the operation symbols go inside the brackets, and we use spaces instead of commas to separate arguments. S-expressions are a general way of representing data structures, and are used a lot in some programming languages like Lisp/Scheme, where they are used to represent both data and code/programs.
- Polish notation: named after the mathematician and logician Jan Łukasiewicz, who realised that if you know exactly how many operands each operator has and you write operation symbols in front, then you can do without brackets. Our expression can be written as $\times + 1\ 3\ 4$, and read as 'multiply the sum of 1 and 3 with 4'. The term bracketed infix as $1 + (3 \times 4)$ would be written as $+ 1 \times 3\ 4$, with the order of operators swapped.
- Reverse Polish notation: this time the operators go after the operands. The term in the diagram is $1\ 3 + 4\ \times$. Note that this is not Polish notation written backwards: in both cases, the values 1,3,4 appear in the same left-to-right order.

Reverse Polish notation (RPN) is especially easy for computers to work with. Some of the first electronic tabletop calculators that supported proper arithmetic terms, as well as some of the first electromechanical computers, needed to be operated in RPN. Indeed, one way for a compiler to parse terms in infix notation is to first convert them to RPN using something called the 'shunting-yard algorithm'.

Unlike infix notation, in a RPN term string two values can follow each other, so spaces or some other kind of separator are needed. For example, `1 2` is two values 1 and 2, but `12` is the single value 12.

[4] `https://craftinginterpreters.com/`

Also in RPN, like with every no-bracket notation, you cannot use the same symbol for different operations with different numbers of arguments. The following three terms are not identical:



The first is the integer 'minus two'. The second is the unary operation 'negation' on the integer '(plus) two'. The third is the binary operation 'subtraction' on the two arguments zero and two. Under the usual laws of arithmetic, they are all syntactically equivalent (and they will all evaluate to 'minus two' when we define that), but they are three different terms. There are various ways of distinguishing these three terms in RPN, for example using a different symbol such as $\sim$ for unary negation, and putting a space between a minus sign representing a subtraction operation and its operand.

Parsing RPN generally works in two stages: first split the input string into *tokens* that are either values, variable names or operators, then parse the list of tokens. Of course, that can be done without reading the string more than once, by producing the tokens 'on the fly'.

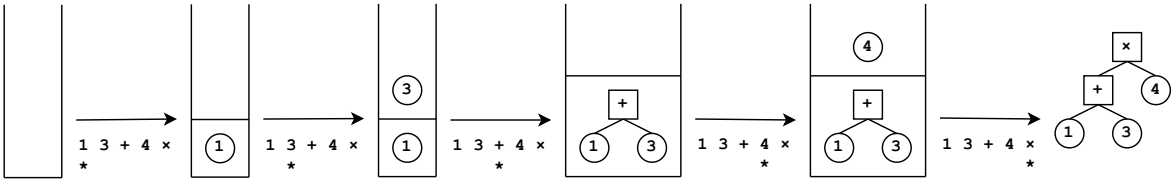
Almost everything to do with RPN works with a stack data structure, that is a 'last in, first out' list where you have three basic operations: *push* adds a new item to the end of the list, *pop* removes the last item from the end of the list, and *empty* checks if the list is empty or not (it is normally an error to try and pop an empty stack).

Definition 8 (RPN parsing)**.** To parse an RPN expression, start with an empty stack and read the tokens one by one from left to right:

RPN1. If you see a value token, push a value term on the stack.
RPN2. If you see a variable token, push a variable term on the stack.
RPN3. If you see a unary operation token, first check if the stack is empty: if so, the string is not a valid RPN term string. Otherwise, pop a term T off the stack, then push a new unary operation term with the old term T as its child.
RPN4. If you see a binary operation token, try and pop two terms T, U off the stack; check if the stack is empty first in both cases, if so then the string is not a valid term string. Then push a new binary operation term on to the stack with children U, T, in the opposite order to the one in which you popped them off the stack earlier.

Parsing succeeds if, when the string ends, there is exactly one term left on the stack.

For example, to parse the string `1 3 + 4 ×` the stack works like this, with the $*$ symbol indicating the token currently being looked at:



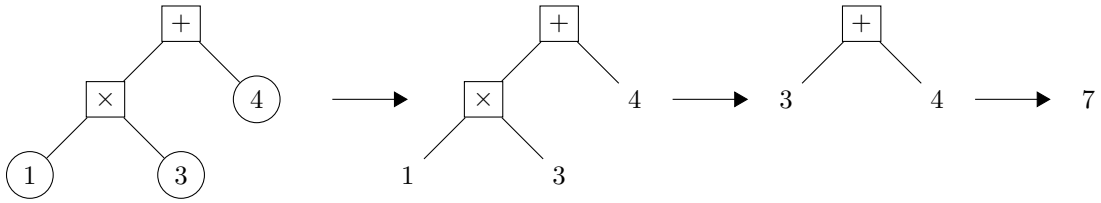
Semantics

In linguistics, *semantics* is the study of 'meanings'; in mathematics, the semantics of a term is whatever it evaluates to.

Definition 9 (evaluation without variables)**.**
The evaluation $e(T)$ of a term T without variables is:

E1. To evaluate a value term, simply take its value.
E2*. We are currently not discussing variable terms.
E3. To evaluate a unary operation term with child T, first evaluate the child. Then apply the operation to the value you get.
E4. To evaluate a binary operation term with children T, U, first evaluate both children. Then apply the operation to the two values.

For example, in the following diagram, nodes with squares or circles around them represent terms and nodes with nothing around them represent values:



Terms in arithmetic can also fail to evaluate, for example if you try and divide by zero. One way to fix this is to define a separate symbol $\perp$ pronounced 'bottom' or 'error', and say that

- Whenever an operation fails, its result is $\perp$.
- The result of evaluating any unary or binary operation term where at least one child evaluates to $\perp$ is again $\perp$.

Terms with variables are a bit more complicated, as you can only evaluate them in an *environment* which is a mapping of variable names to values. For example, in the environment $\{x = 1, y = 2\}$ the term $x + 1$ evaluates to 2.

Definition 10 (evaluation of terms with variables)**.** An environment is a mapping of variable names to values.

E2. To evaluate a variable term in an environment that contains its variable name, take the value of this variable in the environment.

One could again define that if you evaluate a variable term in an environment that does not contain its variable name, then the result is $\perp$.

Evaluating gives us a new notion of saying two terms are 'the same':

Definition 11 (semantic equivalence)**.** Two terms T, U are semantically equivalent if they evaluate to the same value, that is not $\perp$, in every possible environment that defines all their variables. We write this as $T \equiv U$.

For terms without variables, this simply means they evaluate to the same non-$\perp$ value. For example, $1 + 2 \equiv 2 + 1$ and obviously also $1 + 2 \equiv 3$. But also, $x - x \equiv 0$, as whatever we set x to, we get zero.

Note that we do need the clause that we only look at environments that define the variables involved: in an environment that does not define x, the evaluation of $x - x$ is $\perp - \perp$ which becomes $\perp$ again, not 0. This matches how many programming languages work, where if you do not define x then $x - x$ will not compile (if the language is compiled) or will cause a runtime error (if interpreted). However, if you define a function `int f(int x) { return x - x; }` in C and compile with optimisations on, the compiler will simply emit code[5] to return 0 without caring what x you pass in: by the fact that x is a parameter to the function, it will definitely exist in this function's execution environment.

For another example, $1/0 - 1/0 \not\equiv 0$. The term (string) on the left, even though it is of the form $X - X$, still evaluates to $\perp$.

Obviously, syntax and semantics are related: the fact that we suggested $T + U \mathrel{\widehat{=}} U + T$ using a possible snytax rule $a + b \vdash b + a$ on terms comes from the fact that $T + U \equiv U + T$, since addition is commutative. More on this point when we come to logic, where there are only two possible values (true and false): we can not only use syntax rules to prove some terms are semantically equivalent, but we can also use semantics to verify that particular syntax rules are sound.

Completeness and Soundness

In arithmetic, it was a big open question for a while what the best list of syntax rules would be (ideally as short as possible) so that the syntax rules are

- *sound*: every syntax rule respects evaluation, in the sense that if for two terms we have $A \mathrel{\widehat{=}} B$ then we also have $A \equiv B$.
- *complete*: for any two terms A, B with $A \equiv B$, there is some way to derive each one from the other using the syntax rules, in other words $A \mathrel{\widehat{=}} B$.

A complete and sound set of syntax rules, in other words, has $A \mathrel{\widehat{=}} B$ and $A \equiv B$ turn out to be interchangeable. In 1931, the logician Kurt Gödel published his famous 'first incompleteness theorem', which implies (among other things) that we can never find a finite list of syntax rules for arithmetic on natural numbers (under certain technical conditions) that we can prove to be both complete and sound.

[5] most likely `mov eax, 0; ret`

Appendix

Here are some notes that are not examinable for Maths A, but relate to other units.

There is a formal way of defining syntax rules that you will encounter in some of your programming languages units, such as Programming Languages and Computation in Year 2. Here are our term rules in this formalism. The expression $x : T$ is called a *judgement* and is read 'x has type T'. The bar separates the pre-conditions above from the post-conditions below, and you read $\frac{X}{Y}$ mentally as 'if X, then Y'. If there is more than one pre-condition above a bar, you read 'and' between them.

$$R1\ \frac{v : \text{Value}}{v : \text{Term}} \quad R2\ \frac{v : \text{Variable}}{v : \text{Term}} \quad R3\ \frac{T : \text{Term} \quad * : \text{Unary}}{*T : \text{Term}} \quad R4\ \frac{T : \text{Term} \quad U : \text{Term} \quad * : \text{Binary}}{T * U : \text{Term}}$$

Thus for example R3 reads as 'if T has type Term, and $*$ has type Unary [operation] then $*T$ has type Term'. Technically what we are describing here is, in our presentation, term strings instead of terms themselves, but the structure of the terms is clear enough from these rules that it does not make a difference.

We can turn the rules into a program in Haskell easily enough, as you will see in the Functional Programming unit:

```
data UnaryOp = UnaryMinus deriving (Eq, Show)

data BinaryOp = Plus | Minus | Times deriving (Eq, Show)

data Term = ValueTerm Int
          | VariableTerm Char
          | UnaryTerm UnaryOp Term
          | BinaryTerm Term BinaryOp Term
          deriving (Eq, Show)
```

The data type Term contains exactly the four rules R1–R4 from our lecture; we declared types for unary and binary operations too.

We can print terms in different ways by writing printing functions with the same four cases that define terms in the first place:

```
printUnary UnaryMinus = "~"
-- deliberately using a different symbol to binary minus
printBinary Plus = "+"
printBinary Minus = "-"
printBinary Times = "*"

-- fully bracketed infix terms
printFBT :: Term -> String
printFBT (ValueTerm v) = show v    -- this turns a number into a string
printFBT (VariableTerm v) = [v]    -- this turns a char into a string
printFBT (UnaryTerm op t) = "(" ++ (printUnary op) ++ (printFBT t) ++ ")"
printFBT (BinaryTerm t op u) =
    "(" ++ (printFBT t) ++ (printBinary op) ++ (printFBT u) ++ ")"
```

```
printRPN :: Term -> String
printRPN (ValueTerm v) = show v
printRPN (VariableTerm v) = [v]
printRPN (UnaryTerm op t) = (printRPN t) ++ " " ++ (printUnary op)
printRPN (BinaryTerm t op u) =
    (printRPN t) ++ " " ++ (printRPN u) ++ " " ++ (printBinary op)
```

And we can evaluate terms, if we have a way to model environments: here we just use functions.

```
type Env = Char -> Int

evalTerm :: Term -> Env -> Int
evalTerm (ValueTerm v) env = v
evalTerm (VariableTerm v) env = env v
evalTerm (UnaryTerm op t) env = -(evalTerm t env)
evalTerm (BinaryTerm t op u) env =
    let
        a = evalTerm t env
        b = evalTerm u env
    in case op of
        Plus  -> a + b
        Minus -> a - b
        Times -> a * b

-- The following is a dirty way to declare a function evalT
-- that evaluates non-variable terms without needing an environment.
-- There are, of course, better ways to do this in Haskell.
emptyEnv :: Env
emptyEnv v = error "Don't do this"
evalT :: Term -> Int
evalT t = evalTerm t emptyEnv
```

For example, we can now write

```
t = BinaryTerm (VariableTerm 'a') Plus (ValueTerm 2)

printFBT t    -- gives "(a+2)"
printRPN t    -- gives "a 2 +"
e 'a' = 3     -- an environment (partial function) in which a=3
evalTerm t e -- gives 5
```