1

Untyped lambda calculus

1.1 Input–output behaviour of functions

Many functions can be described by some kind of expression, e.g. $x^2 + 1$, that tells us how, given an *input value* for x, one can calculate an *output value*. In the present case this proceeds as follows: first determine the square of the input value and consequently add 1 to this. The so-called 'variable' x acts as an *arbitrary* (or abstract) input value. In a concrete case, for example when using input value 3, one must replace x with 3 in the expression. Function $x^2 + 1$ then delivers the output value $3^2 + 1$, which adds up to 10.

In order to emphasise the 'abstract' role of such a variable x in an expression for a function, it is customary to use the special symbol λ: one adds λx in front of the expression, followed by a dot as a separation marker. Hence, instead of $x^2 + 1$, one writes $\lambda x\,.\ x^2 + 1$, which means 'the function mapping x to $x^2 + 1$'. This notation expresses that x itself is not a concrete input value, but an abstraction. As soon as a concrete input value comes in sight, e.g. 3, we may give this as an argument to the function, thus making a start with the calculation. Usually, one expresses this first stage by writing the input value, embraced in a pair of parentheses, after the function: $(\lambda x\,.\ x^2{+}1)(3)$. (Compare with the case when one wishes to apply the function sin to argument π: this is conveniently expressed as $\sin(\pi)$.)

In what follows, we will concentrate on the general behaviour of functions. We will hardly ever take into account that we know how to 'calculate' in the real world, for example that we can evaluate $3^2 + 1$ to 10, and $\sin(\pi)$ to 0. Only later will we consider well-known elementary functions such as addition or multiplication of numbers, or call upon our knowledge about specific functions such as *square*: our initial intention is to analyse functions from an abstract point of view.

Our first attempts lead to a system called λ-*calculus*. This system encapsulates a formalisation of the basic aspects of functions, in particular their

construction and their use. In the present chapter we do not yet consider *types*, being an abstraction of the well-known process of 'classifying' entities into greater units; for example, one may consider $\mathbb{N}$ as the type of all natural numbers. So this chapter deals with the *untyped* λ*-calculus*. In all the following chapters, however, we shall consider *typed* versions of λ-calculus, varying in nature, which will end up in a system suitable for doing mathematics in a formal manner.

1.2 The essence of functions

From the previous section we conclude that in dealing with functions there are two *construction principles* and one *evaluation rule*.

The construction principles for functions are the following:

Abstraction: From an expression M and a variable x we can construct a new expression: $\lambda x\,.\,M$. We call this *abstraction of* x *over* M.

Application: From expressions M and N we can construct expression $M\,N$. We call this *application of* M *to* N.

If necessary, some parentheses should be added during the construction process.

Examples 1.2.1 – Abstraction of x over x^2+1 gives $\lambda x\,.\,x^2+1$.
- Abstraction of y over $\lambda x\,.\,x-y$ gives $\lambda y\,.\,(\lambda x\,.\,x-y)$, i.e. the function mapping y to: $\lambda x\,.\,x-y$ (which is itself a function).
- Abstraction of y over 5 gives $\lambda y\,.\,5$, i.e. the function mapping y to 5 (otherwise said: the 'constant function' with value 5).
- Application of $\lambda x\,.\,x^2+1$ to 3 gives $(\lambda x\,.\,x^2+1)(3)$.
- Application of $\lambda x\,.\,x$ to $\lambda y\,.\,y$ gives $(\lambda x\,.\,x)(\lambda y\,.\,y)$.
- Application of f to c gives fc. This can also be written, in a more familiar way, as $f(c)$, but this is not the style we use here.

Remarks 1.2.2 *(1) A 'free' usage of these construction principles allows expressions which do not have an obvious meaning, such as* xx *or* $y(\lambda u\,.\,u)$*. In this chapter, we treat these kinds of constructs just like the others, not worrying about their apparent lack of meaning.*

(2) The function 'square' now looks as follows: $\lambda x\,.\,x^2$*. The stand-alone expression* x^2 *is still available, but it is no longer a* function*, but an abstract* output value*, viz. the square of (an unknown, but fixed)* x*. The difference is subtle and may become clearer as follows: let's assume that* x *ranges over* $\mathbb{N}$*,*

the set of natural numbers. Then $\lambda x\,.\,x^2$ *is a function, taking natural numbers to natural numbers. But* x^2 *is not: it represents a natural number.*

(3) The λ *is particularly suited for the description of 'neat' functions, which can be described by a mathematical expression. It takes some effort to use the* λ*-notation to describe functions with a slightly more complicated description, such as, for example:*

- *the function 'absolute value' with definition:*
$$x \mapsto \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases} \text{ ,}$$
- *or the function on domain* $\{0,1,2,3\}$ *with codomain* $\{0,1,2,3\}$ *that is described by:* $0 \mapsto 2, 1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3$.

(In Exercise 1.14 we introduce an if-then-else function, which is helpful in such cases.)

Next to the two construction principles described above, our intuitive function notion gives rise to a *rule* for the 'evaluation' of expressions. The formalisation of the *function evaluation process* is called 'β-reduction'. (An explanation for this name, and a precise definition, will be given in Section 1.8.)

This β-reduction makes use of *substitution*, formally expressed by means of square brackets '[' and ']': the expression $M[x := N]$ represents 'M in which N has been substituted for x'. (Note, however, that substitution is more subtle than one might expect. See Section 1.6 for a precise definition.)

> *β-reduction:* An expression of the form $(\lambda x\,.\,M)N$ can be rewritten to the expression $M[x := N]$, i.e. the expression M in which every x has been replaced with N. We call this process *β-reduction of* $(\lambda x\,.\,M)N$ *to* $M[x := N]$.

Examples 1.2.3
- $(\lambda x\,.\,x^2{+}1)(3)$ reduces to $(x^2{+}1)[x := 3]$, which is $3^2{+}1$.
- $(\lambda x\,.\,\sin(x) - \cos(x))(3+5)$ reduces to $\sin(3+5) - \cos(3+5)$.
- $(\lambda y\,.\,5)(3)$ reduces to $5[y := 3]$, which is 5.
- $(\lambda x\,.\,x)(\lambda y\,.\,y)$ reduces to $x[x := \lambda y\,.\,y]$, which is $\lambda y\,.\,y$.

Reduction is also possible on suitable *parts* of expressions: when an expression of the form $(\lambda x\,.\,M)N$ is a subexpression of a bigger one, then this subexpression may be rewritten to $M[x := N]$, as described above, provided that the rest of the expression is left unchanged. The *full* former expression (with subexpression $(\lambda x\,.\,M)N$) is then said to reduce to the *full* latter expression (with subexpression $M[x := N]$).

The rules describing how reduction extends from subexpressions to bigger ones are called the *compatibility rules* for reduction (see Definition 1.8.1).

Example 1.2.4 By compatibility, $\lambda z\,.\,((\lambda x\,.\,x)(\lambda y\,.\,y))$ reduces to $\lambda z\,.\,(\lambda y\,.\,y)$.

Remarks 1.2.5 *We emphasise that the word 'application' is deceptive: application of M to N is* not the result *of applying M to N, but only a first step in this procedure: all we can say is that 'application' is the construction of a new expression, MN, which, in a later stage, may* perhaps *lead to the actual execution of a function. For example, the application of function $\lambda x\,.\,\sqrt{x}$ to 7 gives expression $(\lambda x\,.\,\sqrt{x})(7)$, in which the function has not yet been executed. It is only after the* reduction *of the latter term that we obtain the result of 'application of the function to 7', namely the 'answer' $\sqrt{7}$.*

The λ-notation is for functions of *one* variable. A function of two or more variables does not fit in this notation. One could make the choice to extend the notation for this purpose. For example, consider the function f of two arguments, defined as $f(x,y) = x^2+y$. We might express f as $\lambda(x,y)\,.\,(x^2+y)$, with a *pair* as input. In this book, however, we will only consider functions of one argument. From the following remark it follows that this is not a real restriction.

Remark 1.2.6 *The behaviour of a function of two (or more) arguments can be* simulated *by converting it into a composite of functions of a single argument. For example, instead of the two-place function $\lambda(x,y)\,.\,(x^2+y)$ one can write $\lambda x\,.\,(\lambda y\,.\,(x^2+y))$. The latter function is called the* Curried *version of the former one, after the λ-calculus pioneer H.B. Curry; the idea of 'Currying' already can be found in the work of M. Schönfinkel (see Schönfinkel, 1924).*

There are subtle differences between the two versions when we provide them with two input values, for example:
– give $f = \lambda(x,y)\,.\,(x^2+y)$ as argument the pair $(3,5)$, then $f(3,5)$ reduces to 3^2+5;
– similarly, we can give $g = \lambda x\,.\,(\lambda y\,.\,(x^2+y))$ these two arguments, but only successively *and in the 'correct' order, so first 3 and then 5; the result is $(g(3))(5)$, which reduces again to 3^2+5 (use the reduction rule twice).*

By the way: with function g we have the liberty to give only one argument and then stop the process: $g(3)$ has a meaning in itself, it reduces to $\lambda y\,.\,(3^2+y)$. This is not possible with function f, which always needs a pair *of arguments.*

1.3 Lambda-terms

The main concern of the discipline called *lambda calculus* is the behaviour of functions in the simplest, most abstract view. This means that we can even do without numbers, and consequently we neither consider, for the time being, the usual simple operations connected with numbers, such as addition and

multiplication, nor more complex ones: exponentiation, the sine. Hence, many of the examples from the previous section are no longer useable.

What remains?

– To start with: *variables* (x, y, ...).
– Moreover: the two construction principles mentioned in the previous section: *abstraction* and *application*.
– Finally: the 'calculation rule' called *β-reduction*.

In the rest of this chapter, we introduce the *untyped λ-calculus* as a formal system, giving precise definitions, including the important operations, and stating the main properties. We omit most of the proofs, for which we refer to the overview text of J.R. Hindley and J.P. Seldin (Hindley & Seldin, 2008) or the seminal work on untyped λ-calculus by H.P. Barendregt (Barendregt, 1981).

Remark 1.3.1 *Lambda calculus or λ-calculus was invented by A. Church in the 1930s (Church, 1933). (It is not completely clear why he used the Greek letter λ – which represents the letter l – for expressing abstraction; see Cardone & Hindley, 2009, Section 4.1, for more details.) Church's aim was to use his lambda calculus as a foundation for a formal theory of mathematics, in order to establish which functions are 'computable' by means of an algorithm (and which are not). See also Section 1.12.*

Expressions in the lambda calculus are called *λ-terms*. The following inductive definition establishes how the set Λ of all λ-terms is constructed. To start with, we assume the existence of an infinite set V of so-called *variables*: $V = \{x, y, z, \ldots\}$.

Definition 1.3.2 (The set Λ of all λ-terms)
(1) (Variable) If $u \in V$, then $u \in \Lambda$.
(2) (Application) If M and $N \in \Lambda$, then $(MN) \in \Lambda$.
(3) (Abstraction) If $u \in V$ and $M \in \Lambda$, then $(\lambda u\,.\,M) \in \Lambda$.

Saying that this is an *inductive definition* of Λ means that (1), (2) and (3) are the *only* ways to construct elements of Λ.

An alternative and shorter manner to define Λ is via *abstract syntax* (or a 'grammar'):

$$\Lambda = V|(\Lambda\Lambda)|(\lambda V\,.\,\Lambda)$$

One should read this as follows: following the symbol '=' one finds three possible ways of constructing elements of Λ. These three possibilities are separated by the vertical bar '|'.

For example, the second one is $(\Lambda\Lambda)$, which means the juxtaposition of an element of Λ and an element of Λ, enclosed in parentheses, gives again an

element of Λ. (Note that the two elements taken successively from Λ may be the same element or different elements; both possibilities are covered by the notation $\Lambda\Lambda$.) What we get in this manner is clearly the same as expressed in Definition 1.3.2 (2).

Examples 1.3.3 Examples of λ-terms are:

– (with Variable as construction principle): x, y, z,
– (with Application as final construction step): $(x\,x)$, $(y\,x)$, $(x(x\,z))$,
– (with Abstraction as final step): $(\lambda x\,.\,(x\,z))$, $(\lambda y\,.\,(\lambda z\,.\,x))$, $(\lambda x\,.\,(\lambda x\,.\,(x\,x)))$,
– (and again, with Application as final step): $((\lambda x\,.\,(x\,z))\,y)$, $(y\,(\lambda x\,.\,(x\,z)))$, $((\lambda x\,.\,x)(\lambda x\,.\,x))$.

Notation 1.3.4 *(The representation of λ-terms; syntactical identity; $\equiv$)*
(1) We use the letters x, y, z and variants with subscripts and primes to denote variables in V.
(2) To denote elements of Λ, we use L, M, N, P, Q, R and variants thereof.
(3) Syntactical identity of two λ-terms will be denoted with the symbol $\equiv$.

So $(x\,z) \equiv (x\,z)$, but $(x\,z) \not\equiv (x\,y)$. Note that '$M \equiv N$' expresses that the actual λ-terms represented by M and N are identical.

With the following recursive definition we determine what the *subterms* of a given λ-term are; these form a *multiset*, since identical terms may occur more than once (see examples later).

Definition 1.3.5 (Multiset of subterms; $\mathtt{Sub}$)
(1) (Basis) $\mathtt{Sub}(x) = \{x\}$, for each $x \in V$.
(2) (Application) $\mathtt{Sub}((MN)) = \mathtt{Sub}(M) \cup \mathtt{Sub}(N) \cup \{(MN)\}$.
(3) (Abstraction) $\mathtt{Sub}((\lambda x\,.\,M)) = \mathtt{Sub}(M) \cup \{(\lambda x\,.\,M)\}$.

We call L a subterm of M if $L \in \mathtt{Sub}(M)$.

From the above definition, the properties below follow.

Lemma 1.3.6 *(1) (Reflexivity) For all λ-terms M, we have $M \in \mathtt{Sub}(M)$.*
(2) (Transitivity) If $L \in \mathtt{Sub}(M)$ and $M \in \mathtt{Sub}(N)$, then $L \in \mathtt{Sub}(N)$.

Note that a certain λ-term can 'occur' several times as a subterm in a given term. For example, with $(x\,x)$ we have that $x \in \mathtt{Sub}((x\,x))$ for two reasons: the 'first' x in $(x\,x)$ is a subterm and also the 'second' x is a subterm. In such cases, one speaks about different *occurrences* of the subterm.

Examples 1.3.7 – The only subterm of y is y itself.
– The subterms of $(x\,z)$ are $(x\,z)$, x and z.

– Similarly, the λ-term $(\lambda x\,.\,(x\,x))$ has *four* subterms: (1) $(\lambda x\,.\,(x\,x))$ itself; (2) $(x\,x)$; (3) the *left* x in $(x\,x)$; and (4) the *right* x in $(x\,x)$. Note that the first occurrence of x in $(\lambda x\,.\,(x\,x))$, the one immediately following the λ, does not count as a *subterm*.
– $\mathtt{Sub}((\lambda x\,.\,(x\,x))(\lambda x\,.\,(x\,x)))$ consists of $((\lambda x\,.\,(x\,x))(\lambda x\,.\,(x\,x)))$, $(\lambda x\,.\,(x\,x))$ (twice), $(x\,x)$ (twice) and x (four times).

It is easy to find the subterms of a λ-term when this λ-term is given in *tree representation*. We do not describe specifically how such a tree representation can be constructed; an example should be enough. See Figure 1.1. The letter 'a' in this figure stands for 'application'.

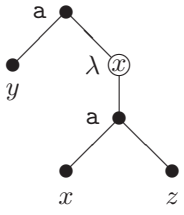

Figure 1.1 The tree of $(y\,(\lambda x\,.\,(x\,z)))$

A variable in a term M that immediately follows a λ symbol is drawn inside the corresponding node in the tree. The subterms of a λ-term M correspond to the *subtrees* in the tree representation of M. (We assume that the reader is familiar with the notion 'subtree'.) Check this in Figure 1.1. Note that the labels of the leaves in such a tree are always variables. And the other way round: a subterm consisting of a single variable corresponds to a labelled leaf. (Remember that a variable placed 'inside' a node is *not* a subterm; cf. Examples 1.3.7.)

There is also a notion of *proper* subterm, which excludes the Reflexivity in Lemma 1.3.6:

Definition 1.3.8 (Proper subterm)
L is a proper subterm of M if L is a subterm of M, but $L \not\equiv M$.

Example 1.3.9 The proper subterms of $(y(\lambda x\,.\,(x\,z)))$ are: y, $(\lambda x\,.\,(x\,z))$, $(x\,z)$, x and z.

Expressions constructed with Definition 1.3.2 have a lot of parentheses, which hampers readability. In order to be able to save on parentheses, the following conventions are followed:

Notation 1.3.10 – *Parentheses in an* outermost *position may be omitted, so* MN *stands for* λ*-term* (MN) *and* $\lambda x\,.\,M$ *for* $(\lambda x\,.\,M)$.

– *Application is left-associative, so MNL is an abbreviation for $((MN)L)$.*
– *Application takes precedence over abstraction, so we can write $\lambda x\,.\,MN$ instead of $\lambda x\,.\,(MN)$.*
– *Successive abstractions may be combined in a right-associative way under one λ, so we write $\lambda xy\,.\,M$ instead of $\lambda x\,.\,(\lambda y\,.\,M)$.*

These conventions are very useful, but also treacherous. As an example, note that $\lambda y\,.\,y\,(x\,y)$ should not be read as $(\lambda y\,.\,y)(x\,y)$, but as $\lambda y\,.\,(y(x\,y))$. Especially when substitution is involved (see Section 1.6), one must be careful.

1.4 Free and bound variables

Variable occurrences in a λ-term can be divided into three categories: *free* occurrences, *bound* occurrences and *binding* occurrences.

The last-mentioned category is the easiest to describe: these are the occurrences immediately after a λ. Other occurrences of variables in a λ-term are free or bound, which can be decided as follows.

In the construction of a λ-term from its parts (see Definition 1.3.2) we always start (see step (1)) with single variables. These are then *free*. In building more complicated terms via steps (2) and (3), it is only in the latter case that freeness may change: an occurrence of x which is free in M becomes *bound* in $\lambda x\,.\,M$. Otherwise said: abstraction of x over M *binds* all free occurrences of x in M; that is why the first x in $\lambda x\,.\,M$ is called a *binding* variable occurrence.

This discussion leads to the following recursive definition, in which $FV(L)$ denotes the set of free variables in λ-term L.

Definition 1.4.1 (FV, the set of free variables of a λ-term)
(1) (Variable) $FV(x) = \{x\}$,
(2) (Application) $FV(MN) = FV(M) \cup FV(N)$,
(3) (Abstraction) $FV(\lambda x\,.\,M) = FV(M) \setminus \{x\}$.

Examples 1.4.2

– $\begin{aligned} FV(\lambda x\,.\,x\,y) &= FV(x\,y)\backslash\{x\} \\ &= (FV(x) \cup FV(y))\backslash\{x\} \\ &= (\{x\} \cup \{y\})\backslash\{x\} \\ &= \{x, y\}\backslash\{x\} \\ &= \{y\}. \end{aligned}$

– $FV(x(\lambda x\,.\,xy)) = \{x, y\}.$

The last example demonstrates that Definition 1.4.1 collects the variables which are free *somewhere* in a λ-term. However, other occurrences of that variable in the same term may be bound. In the example term $x(\lambda x\,.\,x\,y)$, both x and y occur free, but only the first occurrence of x is free, the occurrence of x

just before y is bound. (The occurrence of x after the λ is a binding occurrence, being neither free nor bound.)

When inspecting the tree representation of a λ-term, it is easy to see whether a certain occurrence of a variable is free or bound: start with a variable occurrence, say x, at a leaf of the tree. Now follow the 'root path' upwards, that is: follow the branch from that leaf to the root (the uppermost node). If we pass an 'abstraction node' with the same x inside, then the original x is bound; otherwise it is free. Check these things for yourself with the tree representation of the term $x(\lambda x\,.\,x\,y)$.

Ending this section, we define an important subset of the set of all λ-terms by giving a name to terms without free variables:

Definition 1.4.3 (Closed λ-term; combinator; Λ^0)
The λ-term M is *closed* if $FV(M) = \emptyset$. A closed λ-term is also called a *combinator*. The set of all closed λ-terms is denoted by Λ^0.

Example: $\lambda xyz\,.\,xxy$ and $\lambda xy\,.\,xxy$ are closed λ-terms; $\lambda x\,.\,xxy$ is not.

1.5 Alpha conversion

Functions in the λ-notation (see Section 1.2) have the property that the *name* of the binding variable is not essential. The 'square function', for example, can be expressed by $\lambda x\,.\,x^2$ as well as by $\lambda u\,.\,u^2$. In both cases the expression means 'the function which calculates the square of an input value and gives the obtained number as its output value'. So the variable x (or u) serves as a temporary name for the input value, only meant to make it possible to *speak about* that value: the input called x gives output x^2, which describes the same procedure as 'input u gives output u^2'.

This is the reason why in the λ-calculus one is used to identify λ-terms which only differ in the *names* of the binding variables (together with the variables bound to them).

In order to describe this process formally, we define a relation called α-*conversion* or α-*equivalence*. It is based on the possibility of *renaming* binding (and bound) variables (cf. Hindley & Seldin, 2008, p. 278).

Definition 1.5.1 (Renaming; $M^{x\to y}$; $=_\alpha$)
Let $M^{x\to y}$ denote the result of replacing every free occurrence of x in M by y. The relation 'renaming', expressed with symbol $=_\alpha$, is defined as follows: $\lambda x\,.\,M =_\alpha \lambda y\,.\,M^{x\to y}$, provided that $y \notin FV(M)$ and y is not a binding variable in M.

One says in this case: '$\lambda x\,.\,M$ has been renamed as $\lambda y\,.\,M^{x\to y}$'.

The intended effect is that the binding variable x in $\lambda x\,.\,M$, plus all the corresponding bound x's occurring in M, are renamed as y. Note that the mentioned bound x's are precisely the *free* x's in M.

Now, what about the two conditions in this definition?

(1) *First condition:* $y \notin FV(M)$. If there *were* a free y in M, then this y becomes *bound* to the binding variable y in $\lambda y\,.\,M^{x\to y}$, which is not what we want: renaming should not influence the free/bound status of variables.

Example: Take $\lambda x\,.\,M \equiv \lambda x\,.\,y$, so $y \in FV(M)$. Then $\lambda y\,.\,M^{x\to y} \equiv \lambda y\,.\,y$. Now the same variable occurrence y is first free, and then bound, which conflicts with our intentions regarding 'renaming'. Note that $\lambda x\,.\,y$ is essentially different from $\lambda y\,.\,y$: in the first expression, every input delivers the fixed output y, while in the second case each input returns itself as output.

(2) *Second condition: y is not a binding variable in M.* If this were permitted, then this binding y could unintentionally bind a 'new' y replacing an x.

Example: Take $\lambda x\,.\,M \equiv \lambda x\,.\,\lambda y\,.\,x$; then $\lambda y\,.\,M^{x\to y} \equiv \lambda y\,.\,\lambda y\,.\,y$. In the first expression, the final x is bound by the first λx; in the second expression, the final y, replacing the x, is bound by the *second* λy. So again, renaming would essentially change the situation. In terms of 'behaviour': originally, a first input followed by a second input returns the first input; but after illegitimate renaming, a first input followed by a second input returns the *second* input.

In short: in the renaming of $\lambda x\,.\,M$ to $\lambda y\,.\,M^{x\to y}$, it is prevented that the 'new' binding variable y binds 'old' free y's; and that any 'old' binding y binds a 'new' y.

Renaming in Definition 1.5.1 applies to the full λ-term only. In order to allow it more generally, we extend this definition to the following one:

Definition 1.5.2 (α-conversion or α-equivalence, $=_\alpha$)
(1) (Renaming) $\lambda x\,.\,M =_\alpha \lambda y\,.\,M^{x\to y}$ as in Definition 1.5.1, under the same conditions,
(2) (Compatibility) If $M =_\alpha N$, then $ML =_\alpha NL$, $LM =_\alpha LN$ and, for arbitrary z, $\lambda z\,.\,M =_\alpha \lambda z\,.\,N$,
(3a) (Reflexivity) $M =_\alpha M$,
(3b) (Symmetry) If $M =_\alpha N$ then $N =_\alpha M$,
(3c) (Transitivity) If both $L =_\alpha M$ and $M =_\alpha N$, then $L =_\alpha N$.

So renaming, expressed in (1), is the basis of α-equivalence.

The compatibility rules (2) have the effect that one may also rename binding and corresponding bound variables in an arbitrary *sub*term of a given λ-term.

Reflexivity (3a), symmetry (3b) and transitivity (3c) make α-conversion into an *equivalence relation*.