0.1 Abstract Syntax

0.1.1 Expressions

Programs in V are expressions. Each expression is a member of the abstract syntax tree defined below. The syntax tree will be constructed in parts, with an explanation of what each expression means and their uses. The full syntax tree can be obtained by simply joining all the separate sections.

Functions V, as a functional language, treats functions as first class constructions. This means that functions are regular expressions, and can be passed as arguments, bound to identifiers, etc. Below are both of the function expressions available in V, along with function application.

$$
\begin{array}{lll}
e & ::= & \cdots \\
 & | & func \\
 & | & e_1\ e_2 \\
 & | & x \\
 & & \\
func & ::= & \texttt{fn}\ x \Rightarrow e \\
 & | & \texttt{rec}\ f\ x \Rightarrow e \\
 & | & \texttt{rec}\ f : T\ \ x \Rightarrow e \\
 & & \\
x & ::= & \{x_0, x_1, \ldots\}
\end{array}
$$

All functions in V take exactly one parameter, and so function application evaluates the function by providing a single argument to it.

The first type of function ($\texttt{fn}\ x \Rightarrow e$) defines a simple unnamed function with a parameter x. x is an identifier from a set of name identifiers.

The body of the function is the expression e, which may or may not contain occurrences of the value passed as the argument of the function.

The other two types of functions are both variants of recursive functions available in V. These functions have a name, f, which is also a member of the set of name identifiers, and this name is used to allow recursive calls from withing their body (e). Like unnamed functions, they take exactly one parameter, x, which may or may not be present in their bodies.

The difference between these variants is in their type declaration: the first variant is implicitly typed, while the second is explicitly typed. In the second variant, the programmer specifies the return type of the function as T (types will be shown later).

There are also expressions to use functions.

The first of them is function application: the first of the expressions is a function, and the second is the argument being passed to the function.

The last expression is simply to allow the use of the parameter defined in a function. An identifier x is only considered valid if it has been bound before (either by a function or, as will be seen later, by let declarations of `match` expressions).

Built-in Functions *V* has a few built-in functions that provide basic behavior. These are:

e	::=	⋯	
	|	*Builtin*	
Builtin	::=	+	(add, binary)
	|	−	(subtract, binary)
	|	∗	(multiply, binary)
	|	÷	(divide, binary)
	|	−	(negate, unary)
	|	<	(less than, binary)
	|	≤	(less than or equal, binary)
	|	>	(more than, binary)
	|	≥	(more than or equal, binary)
	|	=	(equal, binary)
	|	≠	(not equal, binary)
	|	∨	(Or, binary)
	|	∧	(And, binary)
	|	get	(binary)
	|	set	(ternary)
	|	stack	(binary)
	|	distort	(ternary)

Every function has its arity declared. The arity of a function defines how many parameters it needs before it can be fully evaluated.

This behavior does not change the fact that functions receive only one parameter. These particular cases can be thought of as nested functions, each taking a single parameter, until all the necessary parameters have been received.

This means that partially applied built-in functions are also treated as functions, and, therefore, can be passed as arguments, bound to identifiers, etc.

Most of these functions are basic and require no explanation. The last 4, however, will only be explained later, after records and accessors have been introduced.

The boolean functions ∨ and ∧ are treated differently than others. Even though they are binary, they have a short-circuit mechanism. This means that, if the result of the application can be known by the first parameter (True in the case of ∨ or False in the case of ∧), the second parameter is not evaluated.

This is in contrast to all other functions in *V*, which evaluate their arguments before trying to evaluate themselves (This will be explained in more detail in **??**)

Constructors *V* has another type of special function: Data Constructors. Data Constructors are, as their name suggests, functions that construct values (data).

When fully evaluated, constructors define a form of structured data, storing the values passed to them as arguments. To access these values, it is possible to pattern match (see 0.1.1) on the constructor name when fully evaluated.

Like built-in functions, constructors can be in a partially evaluated state. Partially evaluated constructors are treated as normal functions and, therefore, do not allow matching on their name.

e	::=	con	
con	::=	n	(arity 0)
	|	b	(arity 0)
	|	c	(arity 0)
	|	nil	(arity 0)
	|	::	(arity 2)
	|	Tuple n	(arity n, $n \geq 2$)
b	::=	$true \mid false$	
n	::=	$\mathbb{Z}$	
c	::=	$'char'$	
$char$	::=	`ASCII characters`	

These functions are *extra* special, however, because they can have arity zero. This means that they "construct" values as soon as they are declared. The basic zero-arity constructors defined in the language are integers, booleans and characters.

The following two constructors (*nil* and ::) are related to lists. The first (with arity 0) is the empty list. The second constructor has arity 2, and extends a list (its second argument) by adding a new value (its first argument) to its head.

The last constructor is actually a family of constructors describing tuples. A constructor Tuple*n* defines a constructor that has n parameters and evaluates to a tuple with n elements. It is also important to note that $n \geq 2$. This means that only tuple constructors with 2 or more parameters are valid.

Records and Accessors The record system in *V* is composed of two parts: records and accessors.

Records are a type of structured data composed of associations between labels and values, called fields. Each label is part of an ordered set of labels l, and can only appear once in every record. For a record to be valid, the order in which its labels are declared must respect the order in the set of labels.

Accessors are terms that allow access to fields within a record. Accessors view records as trees, where each non-leaf node is a record, and each edge has a name (the label of the field). Accessors, then, define a path on this tree, extracting the node at the end of the path.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \{l_1 : e_1,\ \ldots\ l_n : e_n\} & (n \geq 1) \\
 & | & \#l & \\
 & | & \#(e_1,\ \ldots\ e_n) & (n \geq 2) \\
 & & & \\
l & ::= & \{l_1, l_2, ...\} &
\end{array}
$$

The most basic type of path is a simple label $\#l$. An accessor can also be made by combining multiple accesors. The $\#(e_1,\ \ldots\ e_n)$ expression creates an accessor for multiple fields of the same record. Another type of composition is vertical, and is obtained by using the built-in function "stack".

Furthermore, paths can be distorted with the "distort" built-in function, specifying a pair of functions to be applied when extracting or inserting values in the field.

More details about how accessors (and paths) work will be provided in a later section.

Let and Patterns The `let` expression is used to bind values to identifiers, allowing them to be reused in a sub-expression. A `let` expression is divided into 2 parts: the binding and the sub-expression. The binding itself also has 2 parts: the left-hand side, which is a pattern; and the right-hand side, which will be the value to be bound.

Patterns are used to "unpack" values, and can be either explicitly or implicitly typed.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \texttt{let}\ p = e_1\ \texttt{in}\ e_2 & \\
 & & & \\
p & ::= & patt & \\
 & | & patt : T & \\
 & & & \\
patt & ::= & x & \\
 & | & \_ & \\
 & | & con\ p_1,\ \ldots\ p_n & \text{(constructor pattern, } n = \text{arity } con) \\
 & | & \{l_1 : p_1,\ \ldots\ l_n : p_n\} & (n \geq 1) \\
 & | & \{l_1 : p_1,\ \ldots\ l_n : p_n, \ldots\} & \text{(partial record, } n \geq 1)
\end{array}
$$

Much like functions, `let` expressions allow the use of identifiers in an expression by attaching values to the identifiers. Differently from functions, however, a single `let` expression can bind multiple identifiers to values by using patterns. Patterns are matched against the values in the right-hand side of the binding, and can, depending on their structure, create any number of identifier bindings.

The pattern x is a simple identifier pattern. It matches any value in the right-hand side, and binds it to the identifier.

The _ also matches any value, but it does not create any new bindings (this is called the ignore pattern).

The *con* pattern matches a completely applied constructor. This pattern is a compound pattern, with the same number of components as the arity of the constructor.

The *con* pattern itself does not create any bindings, but its components might, since they are themselves patterns and, as such, will be matched against the components of the constructor.

The next pattern is a record pattern. This matches a record with *exactly* the same fields as the pattern. Since all labels in a record are ordered, the fields do not need to be reordered for the matching.

For matching any record with *at least* the fields $l_1, \ \ldots \ l_n$, one can use the pattern $\{l_1 : p_1, \ \ldots \ l_n : p_n, \ldots\}$. This pattern will match any record whose set of labels is a superset of $l_1, \ \ldots \ l_n$.

Match Expression A `match` expression attempts to match a value against a list of patterns. Every pattern is paired with a resulting expression to be evaluated if the pattern matches. Furthermore, it is possible to specify a boolean condition to be tested alongside the pattern matching. This condition will only be tested if the match succeeds, so it can use any identifier bound by the pattern. The matching stops at the first pattern that successfully matches (and any condition is satisfied), and its paired expression is then evaluated.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \texttt{match}\ e\ \texttt{with}\ match_1, \ldots\ match_n & (n \geq 1) \\
 & & & \\
match & ::= & p \rightarrow e & \\
 & | & p\ \texttt{when}\ e_1 \rightarrow e_2 &
\end{array}
$$

Exceptions This expression always evaluates to a runtime error.

$$
\begin{array}{lcl}
e & ::= & \cdots \\
 & | & raise
\end{array}
$$

Runtime errors usually happen when an expression cannot be correctly evaluated, such as division by zero, accessing an empty list, etc.

Sometimes, however, it can be necessary to directly cause an error. The *raise* expression serves this purpose.

0.1.2 Types

Since *V* is strongly typed, every (valid) expression has exactly one type associated with it. Some expressions allow the programmer to explicitly declare types, such as patterns and recursive functions. Other expressions, such as $e_1 = e_2$, or even constants, such as 1 or *true*, have types implicitly associated with them. These types are used by the type system (see **??**) to check whether an expression is valid or not, avoiding run-time errors that can be detected at compile time.

Types Below are all the types available in *V*. The first type is a fully applied constructor type. The second type is a function type. The third type is a record type and, finally, the last type is an accessor type.

$$
\begin{array}{lcll}
T & ::= & \cdots & \\
 & | & conT\ T_1,\ \ldots\ T_n & (n = \text{arity } conT) \\
 & | & T_1 \to T_2 & \\
 & | & \{l_1 : T_1,\ \ldots\ l_n : T_n\} & (n \geq 1) \\
 & | & T_1 \# T_2 & Accessor
\end{array}
$$

Most of the types are compound types, and the only scenario in which a type is not compound is for constructor types with arity 0. Function types specify the type of the single parameter (T_1) and the type of output (T_2).

Record types are also compound types, but they associate every component to its corresponding label. Just like record expressions, the labels must be ordered.

Finally, accessor types define the types of accessor expressions. They have two components: T_1, which is the type of the record being accessed; and T_2, the type of the value being accessed. It is read as T_1 *accesses* T_2.

Constructor Types These are types associated with constructors. Much like constructors, they can take any amount of arguments to be fully applied, and the number of arguments they take is described by their arity. Instead of taking values as arguments, however, constructor types take types as arguments.

$$
\begin{array}{lcll}
conT & ::= & \text{Int} & (\text{arity } 0) \\
 & | & \text{Bool} & (\text{arity } 0) \\
 & | & \text{Char} & (\text{arity } 0) \\
 & | & \text{List} & (\text{arity } 1) \\
 & | & \text{TupleT } n & (\text{arity } n,\ n \geq 2)
\end{array}
$$

Variable Types These types represent an unknown constant type. Explicitly typed expressions cannot be given variable types, but they are used by the type system for implicitly typed expressions. In the course of the type inference, the type system can replace variable types for their type.

It is important to realize that variable types already represent a unique type with an unknown identity. This means that a variable type may only be replaced by the specific type which it represents and not any other type. This distinction becomes important when talking about polymorphism, which uses variable types, along with universal quantifiers, to represent a placeholder for any possible type (this is discussed in greater detail in **??**).

$$
\begin{array}{lcl}
T & ::= & \cdots \\
 & | & X^{Traits}
\end{array}
$$

$$
X \quad ::= \quad X_1, X_2, ...
$$

0.1.3 Traits

Types can conform to traits, which define certain behaviors that are expected of said type. Regular types always have their trait information implicitly defined, since this information is included in the language. Variable types, on the other hand, can explicitly

state which traits they possess, restricting the set of possible types they can represent (this is represented by the superscript $Traits$ in a variable type X).

$$
\begin{array}{lcll}
Traits & ::= & \varnothing & \\
 & | & \{Trait\} \cup Traits &
\end{array}
$$

$$
\begin{array}{lcll}
Trait & ::= & Equatable & \\
 & | & Orderable & \\
 & | & \{l : Type\} & \text{(Record Label)}
\end{array}
$$

The information on which types conform to which traits is defined in the unification environment (see **??**). When a type T conforms to a trait $Trait$, the notation used is $T \in Trait$. The same notation can be used to describe when a type conforms to a set of traits $Traits$ (i.e. $T \in Traits$).

By default, the following rules hold for conformance:

Equatable If a type T is *Equatable*, expressions of type T can use the equality operators ($=, \neq$).

To define the set of types that belong to *Equatable*, the following rules are used:

$$\{\text{Int}, \text{Bool}, \text{Char}\} \subset Equatable$$
$$T \in Equatable \implies List\ T \in Equatable$$
$$X^{Traits} \in Equatable \implies Equatable \in Traits$$

Orderable If a type T is *Orderable*, expressions of type T can use the inequality operators ($<, \leq, >, \geq$). Any type that is *Orderable* is also *Equatable*.

To define the set of types that belong to *Orderable*, the following rules are used:

$$\{\text{Int}, \text{Char}\} \subset Orderable$$
$$T \in Orderable \implies List\ T \in Orderable$$
$$X^{Traits} \in Orderable \implies Orderable \in Traits$$

Record Label A type T_1 conforms to a Record Label Trait $\{l : T_2\}$ if it is a record that contains a field with the label l and the type T_2.

If the type conforms to the trait $\{l : T_2\}$, it can then use the accessor $\#l$. This ensures that accessor for a field can only be used on records that have that field.

To define the set of types that belong to a record label $\{l : T\}$, the following rules are used:

$$\{l_1 : T_1,\ \ldots\ l_n : T_n,\ \ldots\ T_k\} \in \{l : T\} \iff l_n = l \wedge T_n = T \qquad (1 \leq n \leq k)$$
$$X^{Traits} \in \{l : T\} \implies \{l : T\} \in Traits$$