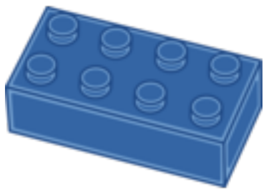

Bild: Language Reference Manual

Graham Gobieski

0. Contents

1. Introduction

Bild is a new, compiled programming lanaguage that seeks to take the best parts of imperative and functional languages and combine them into one concise language. Bild was inspired by Javascript, Swift, Haskell, OCaml, and Python and includes only the best features of each language. While the syntax of Bild will seem familiar to those who have programmed in any popular imperative langauge (C, Javascript, Java, etc.), Bild is inherently functional.

At the moment, the Bild compiler compiles Bild code to valid Java code and then relies on the Java Virtual Machine for portability, garbage collection, and runtime environment.

2. Lexical Conventions

2.1 Line Structure

Bild programs are composed of a set of logical instructions staments. Simple statements are separated by `;` and compound statements (including conditional statements) are separated by `{}`.

2.2 Tokens

There are several token classes in Bild: identifiers, keywords, literals, operators, and delimters. Whitespace and comments are ignored except as separators of tokens.

2.3 Comments

There are two types of comments: a single line comment and a multi-line comment. The single line comment begins with `//` and ends with the NEWLINE character. The multi-line comment begins wiht `/*` and ends with `*/`.

2.4 Identifiers

There are two types of identifiers: type identifiers and function identifiers. Identifiers are separated from other tokens via whitespace.

```
type-identifier = [A-Z]+[A-Za-z0-9_]*
identifier = [A-Za-z]+[A-Za-z0-9_]*
```

2.5 Key Words

The following are reserved keywords of Bild and are not valid for use as identifiers

```
if          import      while       raise
else        of          for         return
print       elif        is          in
try         break       fun         true
false       catch       when
```

2.6 Operators

The following tokens represent operators.

```
+           *           >=          ||
-           ==          <=          &&
\           !=          %           !
```

2.7 Literals and Constants

2.7.1 Integer Constants

An integer constant consists of a sequence of digits that does not begin with 0. All integers in Bild are taken to base 10. Negative integer constants consist of a sequence of digits prefixed by a dash.

```
integer-constant ::= [-]?[0-9]+
```

2.7.2 Double Constants

A double constant consists of an integer part, decimal point, fractional part, and optional exponential part, which consists of an integer prefixed by an 'e'. Either the integer or fractional part, but not both, may be missing. Both integer and fractional parts are themselves integers, separated by the decimal point. The decimal point must be present.

```
double-constant ::= [-]?([0-9]+.[0-9]*|[0-9]*.[0-9]+)(e[-]?[0-9]+)?
```

2.7.3 Boolean Constants

A boolean constant is either true or false and takes the following form.

```
bool-constant ::= 'true|false'
```

2.7.4 Character Constant

A character constant is a single character surrounded by single quotes. Bild contains several escape sequences which can be used as characters or within string literals:

- newline `\n`
- tab `\t`
- backslash `\\`
- single quote `\’`
- double quote `\"`

```
char-constant::= '[^']'
```

2.7.5 String Literals

A string literal is a sequence of characters surrounded by double quotes. String literals are immutable, and thus cannot be altered. Any operations performed a string literal will not affect the original literal but instead generate a new string.

```
string-literal ::= "[^"]"
```

2.7.6 Table Literals

A table literal is a comma-separated sequence delimited by curly braces. The comma-separated sequence can take two forms.

- A sequence of values of any type — there cannot be different types in this sequence. This generates a table where the values are keyed in sequential order from the integer 0 to sequence length-1.
- A sequence of key-value pairs written in the form key : val. As before, values cannot vary within a table. Keys are restricted to integer or string types, and a table can be keyed by either integers or by strings, but not both.

```
table-literal ::= {((string-constant|integer-constant:any-constant),*|any-constant,*)
                    (string-constant|integer-constant:any-constant)|any-constant))?}
any-constant ::= type-identifier|identifier|integer-constant
                |string-literal|table-literal|bool-constant
                |char-constant|integer-constant|double-constant
```

2.8 Delimiters

The following characters are delimiters.

```
(           )           [           ]
{           }           |           ;
?           +=          -=          *=
/=          %=          ,           @
:           ?
```

And the following characters have special meaning as part of other tokens

```
`           "           \
```

3. Syntax Notation

3.1 Meaning of Identifiers/Variables

Identifiers are names which can refer to functions, variables, and types. Each identifier is a string consisting of digits, letters, and underscores which does not begin with a digit.

Variables are storage locations that contain values. Depending on where in a program variables are initialized, they are either global or local to a particular scope. See Storage Scope for more details.

Bild is statically typed, which means that every variable has a type. The type of a variable determines the meaning and behavior of its values, and also the nature of storage needed for those values.

3.2 Storage Scope

The visibility of an identifier and liftetime of a variable's storage depends on where a variable is initialized. If a variable is initialized outside any block, it is a global variable. A global variable can be accessed by any part of the program below the global variable's initialization. It's storage stays alive throughout the entire execution of the program.

If a variable is initialized within any block, it is a local variable. A local variable can be accessed within the scope it is initialized, at or below its initialization. Its storage will be destroyed at the end of the scope.

3.3 Basic Types

There are six basic types in Bild:

- int
- double
- bool
- char
- string
- table

Integers are 32-bit signed two's complement integers. Doubles are double-precision 64-bit IEEE 754 floating points. Booleans are either true or false and can be evaluated as if they were numbers. We will refer to doubles, integers, and booleans as arithmetic types.

Characters are single unicode characters while strings are a sequence of 0 or more unicode characters. Both characters and strings are guaranteed to occupy O(n) space in memory, where n is the number of characters.

Arithmetic types, characters and strings are immutable, which means that their value cannot be changed once they are created. When a variable with an immutable type is assigned a new value, the old value and underlying storage are destroyed.

Tables, unlike the immutable types, are mutable. This means that variables do not contain tables, but rather contain references to tables. Assigning a table to a variable results in that variable storing a reference to that table. Similarly, when tables are passed as parameters to functions or returned from functions, the respective parameters and return values are references. In each of these operations, there is no copying of internal table data, only copying of references. Most importantly, elements in a table are of optional type. Please see 3.7 for a discussion of what an optional types means for use. This design decision prevents `NullPointerExceptions` and `IndexOutOfBoundExceptions` at runtime.

3.4 Tuples

A tuple is a custom type that is composed of one or more other types. More specifically tuple are paranthesized, comma-separated lists of built-in and custom types. They take the following form where `tuple-expression-list` is a comma separated list of inferred types:

```
tuple ::= (tuple-expression-list)
tuple-expression-list ::= expression|_
                        |(expression|_), tuple-expression-list
```

Tuples can be used as return types in place of tables or a simple data structures. However, tuples are immutable, which means that a tuple cannot be expanded after initialization, nor can the types held by the tuple be changed.

3.5 Function Types

Each function has a corresponding type that matches the return type of the function. This return type is inferred and in cases where a generic function is written and called with parameters of several different types, multiple functions and multiple function types are inferred and generated. Moreover, functions may be passed as parameters to other functions and can be stored as values in tables. Please see section 7.2 for more discussion.

3.6 Custom Types

Bild provides a platform for creating powerful custom types. These types have a similar syntax to types in OCaml and can be used wherever basic and function types are used. Please see section 7.3 for more discussion.

3.7 Optional Types

Any variable can be declared optional by placing a question mark `?` following the identifier. An variable of optional type does not necessarily have a value associated with it. As a result, variables of optional type must be checked for a value before use, using a try statement so that a `NullPointerException` and a `IndexOutOfBoundsException` can be avoided at runtime. Otherwise a warning will be generated by the compiler. To access a value, an exclammation point `!` must be placed after the variable when used in an expression.

Definition:

```
identifier? ::= anyType?
```

Access:

```
identifier!
```

try Statement:

```
try identifier expression : expression;
```

The first expression is evaluated if the variable represented by the identifier is not null; otherwise the second expression is evaluated. Please see section 4 for further discussion on try.

3.8 Automatic Conversions

3.8.1 Mixed Arithmetic Expressions

Bild will automically execute the following promotions of type in mixed arithmetic expressions:

- `int op double`: int will be promoted to double
- `char op double`: char will be promoted to double
- `bool op double`: bool will be promoted to double
- `char op int`: char will be promoted to int
- `bool op int`: bool will be promoted to int

3.8.2 Mixed String Expressions

Bild will automatically convert the non-string type to a string when found in the following situations:

- `string + double`: double will be converted to string
- `string + int`: int will be converted to string
- `string + bool`: bool will be converted to string; `"true"` or `"false"`
- `string + char`: char will be converted to string
- `string + table`: table will be converted to string; string versions of all elements will be concatenated together in a legible format.

4 Expressions

The precedence of expression operations is the same as the order of the major subsections of this section unless otherwise noted. Within each subsection, operators have the same precedence. Left or right associativity will be specified in each of the subsections for each operator.

```
expression ::= try? prefix-expression binary-expressions?
expression-list ::= expression | expression, expression-list
```

The `try` operator is an optional operator that checks to see if an optional (potentially null value) expression is valid; if it does exist the first condition is executed and otherwise a second expression can be specified.

4.1 Primary Expressions

Primary Expressions are identifiers, constants, literals, or expressions in parentheses.

```
primary-expression ::= identifier | constant | literal | (expression)
```

4.2 Prefix Expressions

```
prefix-operator ::= - | ++ | --
prefix-expression ::= prefix-operator? postfix-expression
```

4.4.1 Unary Minus

The operand of the unary `-` operator must be an arithmetic type and the result is the negative of its operand. An integral operand undergoes integral promotion. The type of the result is the type of the promoted operand. The precendence is lower than the every postfix expression except postfix increment and decrement.

4.4.2 Increment

Prefix increment is applicable to expressions that are of an arithmetic type. The operator increments a number by one.

4.4.3 Decrement

Prefix decrement is applicable to expressions that are of an arithmetic type. The operator decrements a number by one.

4.3 Postfix Expressions

Operators in postfix expressions group left to right.

```
postfix-operator ::= ++ | -- | ! | ?
postfix-expression ::= primary-expression
                        | postfix-expression[expression-list]
                        | postfix-expression(argument-expression-list?)
                        | postfix-expression(postfix-expression)
                        | postfix-expression postfix-operator
```

4.3.1 Tuple Access

A postfix expression followed by a postfix-expression in parantheses is a postfix expression denoting access to a component of a tuple. The expression in the parantheses must be an integer and must be in the range of the length of the tuple.

4.3.2 Table References

A postfix expression followed by postfix-expression in square brackets is a postfix expression denoting a subscripted table reference. The expression in square brackets must be a table key of type int or string, the postfix expression must be a table. The whole expression is of a basic type, a functional type, or a custom type(the value of the value associated with the key).

4.3.3 Function Calls

A function call is a postfix expression (function designator) followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which constitute the arguments to the function. Recursive functions are permitted.

The term *argument* refers to an expression passed by a function call, the term *parameter* refers to an input object or its identifier received by the function definition.

4.3.4 Increment

Prefix increment is applicable to expressions that are of an arithmetic type. The operator increments a number by one and occurs after the associated expression has been evaluated.

4.3.5 Decrement

Postfix decrement is applicable to expressions that are of an arithmetic type. The operator decrements a number by one and occurs after the associated expression has been evaluated.

4.3.6 Force Value

The exclammation point operator `!` strips the optional aspect of a type and forces a return of the value that the variable represents. If this is used in a unchecked situation where the optional has not been checked for nullity then a NullPointerException may be raised at runtime.

4.4 Binary Operators

Binary expressions include the standard, binary arithmetic expressions (`+`, `-`, `*`, `/`, `%`) as well as assignment operators and the tertiary conditional operator. Automatic conversions are applied as discussed in section 3.8.

```
binary-operator ::= + | - | / | * | %
assignment-operator ::= = | += | -= | *= | /= | %= | !=
relational-operator ::= == | != | <= | >= | > | <
logical-operator ::= && | ||


binary-expression ::= binary-operator prefix-expression
                        | assignment-operator try? prefix-expression
                        | ? try? expression : try? prefix-expression
                        | relational-operator prefix-expression
                        | logical-operator prefix-expression
                        | is prefix-expression

binary-expressions ::= binary-expression binary-expressions?
```

4.4.1 Assigment Operators

Assignment of an identifier to a value or function is considered an expression. Besides the standard equality, there exists five other assignment operators. Each of these additional operators simplies the following general expression:

```
identifier = try? identitifier op prefix-expression
=> identifier op= try? prefix-expression
```

4.4.2 Tertiary Operator

The tertiary operator is an instance of syntatic sugar and reduces the standard syntax for an if-else statement into a single line. The types of the blocks of the if and else must match.

4.4.3 Relational Operators

The relational operators group left-to-right. a<b (greater), <= (less or equal) and >= (greater or equal) all yield false if the specified relation is false and true otherwise The type of the result is bool. The usual arithmetic conversions are performed on arithmetic operands.

4.4.4 Equality Operators

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence. (Thus a<b == c<d is true whenever a<b and c<d have the same truth-value.)

4.4.5 Boolean Operators

4.4.5.1 Logical AND

The `&&` operator groups left-to-right. It returns true if both its operands are true, false otherwise.

`&&` guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to false, the value of the expression is false. Otherwise, the right operand is evaluated, and if it is equal to false, the expression's value is false, otherwise rue.

The operands must be of type int, double, or bool, but don't have to be of the same type. The result is a bool.

4.4.5.2 Logical OR

The `||` operator groups left-to-right. It returns true if one of its operands is true, false otherwise.

`||` guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to true, the value of the expression is true. Otherwise, the right operand is evaluated, and if it is equal to false, the expression's value is false, otherwise true.

The operands must be of type int, double, or bool, but don't have to be of the same type. The result is a bool.

4.4.5.3 Logical is

The `is` operator groups left-to-right. It returns true if the first operand is the type of the second operand (inferred type or explicitly denoted).

`is` guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; Then the second operand is evaluated. If the types of the first and second operand match then the expression returns true, otherwise false.

The operands can be of any type. The result is a bool.

4.6 Constant Expressions

A constant expression is an expression that is just a constant. The type of the constant expression is the type of the constant, the value of the constant expression is the value of the constant.

5 Declarators

There are three types of declarators: function, type and import. The function and type declarators cannot be nested (they must be global); however, the type declarator can be nested as it is also considered a statement.

```
declarator ::= function-declarator | import-declarator | type-declarator
```

5.1 Function Declarators

Function declarators describe a function and may not be nested within other statements unless in a type declaration. In a type declaration, functions can only be nested one-level, which means that it is not possible to have a function within a function. The type of the return statement of the function is the function type. If no return statement is provided the function defaults to type *int*.

```
function-declarator ::= identifier(arg-list?) compound-statement
arg-list ::= identifier | identifier arg-list
```

5.2 Type Declaratiors

Type declarators are an important aspect of Bild. They are used to define a custom type and may be associated with functions. Inheritance is implemented; the associated methods of the types specified after the `of` token will be inherited by the new custom type. With this said, Bild is equipped with strict single inheritance. Inherited variables and non-inherited variables are considered private to the type and are not accessible by expressions not within the type body; they, however, are accessible by functions and statements within the body. Inherited functions are public and are accessible by expressions outside the type body. See section 7.2 for further discussion.

```
type-declarator ::= type identifier = the
type-list ::= type-identifier (of type compound-statement?)?
                | type-identifier (of type compound-statement?)? type-list
type-body ::= (statement|function-declarator)
                |(statement|function-declarator) type-body
type ::= type-identifier|basic-type|tuple-type
basic-type ::= int|double|bool|char|string|table of basic-type
tuple-type ::= (tuple-type-list)
tuple-type-list ::= basic-type|type-identifier|_
                    |(basic-type|type-idenditifer|_),tuple-type-list
```

5.3 import Declarators

Import declarators describe file that have relevant functions and variables to be included in the current file.

```
import-declarator ::= import identifier | (.|..)/(../)*import-path-identifier
import-path-identifier ::= identifier | idenitifer/import-path-identifier
```

6 Statements

Statements are executed in sequence and do not return a value. In Bild, a statement takes the following form:

```
statement ::= (expression-statement
                | compound-statement
                | conditional-statement
                | type-declarator
                | print-statement
                | return-statement
                | break-statement
                | raise-statement)?;
```

6.1 Simple Statements

6.1.1 Expression Statements

Expression statements will typically be a function call but can be any arbitrary expression.

```
expression-statement ::= expression
```

6.1.2 return Statement

Return statements are used to return a specific value from a function. As such they can only be used within function bodies. Moreover, function types are inferred using the return statements of a function. This means that the types of the return statements must be in agreement.

```
return-statement ::= return expression
```

6.1.3 break Statement

A break statement stops the closest loop in scope and transfers control to the statements directly following the loop.

```
break-statement ::= break
```

6.1.4 raise Statement

A raise statement raises an error, which can then be handled via a try-statement. Please see section 5.3.2 for a definition of type.

```
raise-statement ::= raise type
```

6.1.5 print Statement

The print statement evaluates the expression, converts it to a string, and prints to StdOut.

```
print-statement ::= print(expression)
```

6.2 Compound Statements

Compound statements are used to write several statements when one statement is expected.

```
compound-statement ::= statement-list
statement-list ::= {statement} | {statement statement-list}
```

6.3 Conditional Statements

Conditional statements describe any block that begins with some sort of boolean conditional.

```
conditional-statement ::= if-statement
                        | try-statement
                        | while-statement
                        | for-statement
                        | match-statement
```

6.3.1 if Statement

The first substatment is executed if the expression is true otherwise control is transferred to an optional else-if clause and then finally to an optional else clause if none of the previous else-if clauses evaluate to true. A exclammation point placed at the beginning of the expression in the conditional clause negates the boolean of the expression.

```
if-statement ::= if(!?expression) compound-statement else-statement?
else-statement ::= el if-statement | else compound-statement
```

6.3.2 try Statement

The runtime environment tries to execute the first substatement, but if an error is raised control is transferred to the second substatement.

```
try-statement ::= try compount-statement catch(type-identifier) compound-statement
```

6.3.3 while Statement

The *compound-statement* is executed until the expression is no longer true.

```
while-statement ::= while(expression) compound-statement
```

6.3.4 for Statement

The *compound-statement* is executed until the middle expression is no longer true. The first expression initializes a variable. The second expression checks to see if the value of the variable is still valid and the third expression changes the value of the variable is some way.

```
for-statement ::= for(expression; expression; expression) compound-statement
                | for(identifier in expression) compound-statement
```

6.3.5 match Statement

The *match-statement* is important to control-flow in Bild. The statement provides an easy way to check the type or value of a variable and act on the value and the type of the variable in question. Read about OCaml pattern matching for further explanation.

```
match-statement ::= match identifier with match-list
match-list ::= match-conditional compound_statement
        | match-conditional compound_statement match-list
match-conditional ::= (type-identifier | tuple-type
                        | tuple | identifier | _)(when expression)?
```

7 Data Model, Inheritance, and Generics

7.1 Data Model

Bild implements a similar data model as Java: technically all calls are call-by-value, but in many situations they may feel like call-by-reference. Basic types are always copied when passed to a function. However, tuples, custom types, and function types are implemented using references so when passed to a function, the reference is copied and any modifications done to the memory locations where the reference points, will be seen outside the scope of the function.

7.2 Generics

Bild is equipped with support for inferred generics; inferred generics if used in the right way are as powerful as normal, explicit generics. Each function, including those associated with a custom type, is inherently generic, which means that unless a constant or literal is specified in the return statement, the function type will be generated by analyzing the calls to the function. Moreover, if a function takes parameters and then is called by two set of parameters that differ in type, two separate function definitions will be generated during code generation. This has the effect that every function is generic, since what is passed to the function determines the resulting function type.

In addition to generic functions, custom types may also be generic by reusing the *type-identifier* within the type definition. For example:

```
type a = B of a | C of int
```

In this example the use of `a` twice allows one to pass any type to `b`. In other words, `b` can hold any type, including basic types, function types, tuples, and other custom types. The type of b is determined when it is constructed.

7.3 Inheritance

In Bild, a custom type may inherit from (an)other type(s) by specifiying the other type(s) after the `of` token. For example:

```
type a = B {
    fun hello(){
        print("Hello World!");
    }
}
type b = C of B {
    fun portHello(){
        print("Oi!");
    }
}
```

In this situation `C` inherits the function `hello` from `type a` since `C` is specified as being of type `B`.

In situations where multiple types are specified following the `of` token, such as when a tuple type is specified, nothing is inherited, since the tuple is a type in its self and does not have any associated methods. More formally this means that Bild implements strict single inheritance of types.

8 I/O

8.1 Standard I/O

```
print(expression)
```

The `print` statement takes an expression, converts it to a string and outputs it to StdOut.

```
read(string) | read()
```

The `read` built-in function takes either a single `string` argument or takes no arguments. In the case of a single argument, `read` attempts to read from the file specified via the `string`. If it is unable to do so, it raises an error. In the other case where no arguments are specified, `read` attempts to read from StdIn. Regardless `read` returns a string.

8.2 File I/O

```
write(string)
```

The `write` built-in function takes a single `string` argument specifying what file should be written to. The `write` function automatically overwrites a file and will create a file when the file in the string specified does not exist. However `write` may raise an error when the path specified is not valid.

9 Standard Library

9.1 String

```
length(string)
```

Takes a `string` and returns the length (the number of characters) as a `int`.

```
charAt(string, int)
```

Takes a `string` and `int` and returns an optional character at the specified position.

```
toInt(string)
```

Takes a `string` and parses the string, returning an optional `int` that the string represents.

```
toDouble(string)
```

Takes a `string` and parses the string, returning an optional `double` that the string represents.

9.2 Table

```
length(table)
```

Takes a `table` and returns the length (the number of elements) as a `int`.

```
keys(table)
```

Takes a `table` and returns a `table` populated with the keys of the `table`.

9.3 I/O

Please see section 8 for discussion on I/O built-in functions.

10 Example Programs

GCD Program with Optionals

```
fun GCD(a, b){
    c = a;
    while(a != 0){
        c = a;
        a = b % a;
        b = c;
    }
    return b;
}
i1 = try toInt(read())! : print("Not a number");
i2 = try toInt(read())! : print("Not a number");
print(GCD(i1,i2));
```

Custom Types

```
type suit = SPADE | HEART | DIAMOND | CLUB
type face = KING | QUEEN | JACK | ACE | NOFACE
type order = Order of int
type card = Card of (order, face, suit)
jackOfHearts = Card(Order(10), King, DIAMOND)
```

Inheritance and Generics

```
type employee = E{
    sal = 0;
    setSalary(aSal){sal = aSal;}
    getSalary(){return sal;}
}
type hourly_employee = HE of E{
    hours = 40;
    setSalary(aSal){sal = aSal/40;} //Override the method
}
/*This function will be interpreted as producing both an int and a string, so two
different function definitions will be generated*/
fun addSals(s1, s2){return s1 + s2;}
em1 = E;
em1.setSalary(500);
em2 = HE;
em2.setSalary(500);
sum = addSals(em1.getSalary(), em2.getSalary());
print(sum); //prints 512
concat = addSals(""+em1.getSalary(), ""+em2.getSalary());
print(concat); //prints 50012
```

11 Complete Grammar

11.1 Types

```
type-identifier = [A-Z]+[A-Za-z0-9_]*
identifier = [A-Za-z]+[A-Za-z0-9_]*
integer-constant ::= [-]?[0-9]+
double-constant ::= [-]?([0-9]+.[0-9]*|[0-9]*.[0-9]+)(e[-]?[0-9]+)?
bool-constant ::= 'true|false'
char-constant::= '[^']'
string-literal ::= "[^"]"
table-literal ::= {((string-constant|integer-constant:any-constant),*|any-constant,*)
                    (string-constant|integer-constant:any-constant)|any-constant))?}
any-constant ::= type-identifier|identifier|integer-constant
                |string-literal|table-literal|bool-constant
                |char-constant|integer-constant|double-constant
tuple ::= (tuple-expression-list)
tuple-expression-list ::= expression|_
                        |(expression|_), tuple-expression-list
```

11.2 Expressions

```
expression ::= try? prefix-expression binary-expressions?
expression-list ::= expression | expression, expression-list
primary-expression ::= identifier | constant | literal | (expression)
prefix-operator ::= - | ++ | --
prefix-expression ::= prefix-operator? postfix-expression
postfix-operator ::= ++ | -- | ! | ?
postfix-expression ::= primary-expression
                        | postfix-expression[expression-list]
                        | postfix-expression(argument-expression-list?)
                        | postfix-expression(postfix-expression)
                        | postfix-expression postfix-operator
binary-operator ::= + | - | / | * | %
assignment-operator ::= = | += | -= | *= | /= | %= | !=
relational-operator ::= == | != | <= | >= | > | <
logical-operator ::= && | ||
binary-expression ::= binary-operator prefix-expression
                        | assignment-operator try? prefix-expression
                        | ? try? expression : try? prefix-expression
                        | relational-operator prefix-expression
                        | logical-operator prefix-expression
                        | is prefix-expression
binary-expressions ::= binary-expression binary-expressions?
```

11.3 Declarators

```
declarator ::= function-declarator | import-declarator | type-declarator
function-declarator ::= identifier(arg-list?) compound-statement
arg-list ::= identifier | identifier arg-list
type-declarator ::= type identifier = the
type-list ::= type-identifier (of type compound-statement?)?
                | type-identifier (of type compound-statement?)? type-list
type-body ::= (statement|function-declarator)
                | (statement|function-declarator) type-body
type ::= type-identifier|basic-type|tuple-type
basic-type ::= int|double|bool|char|string|table of basic-type
tuple-type ::= (tuple-type-list)
tuple-type-list ::= basic-type|type-identifier|_
                     | (basic-type|type-idenditifer|_),tuple-type-list
import-declarator ::= import identifier | (.|..)/(../)*import-path-identifier
import-path-identifier ::= identifier | idenitifer/import-path-identifier
```

11.4 Statements

```
statement ::= (expression-statement
                | compound-statement
                | conditional-statement
                | type-declarator
                | print-statement
                | return-statement
                | break-statement
                | raise-statement)?;
expression-statement ::= expression
return-statement ::= return expression
break-statement ::= break
raise-statement ::= raise type
print-statement ::= print(expression)
compound-statement ::= statement-list
statement-list ::= {statement} | {statement statement-list}
conditional-statement ::= if-statement
                            | try-statement
                            | while-statement
                            | for-statement
                            | match-statement
if-statement ::= if(!?expression) compound-statement else-statement?
else-statement ::= el if-statement | else compound-statement
try-statement ::= try compount-statement catch(type-identifier) compound-statement
while-statement ::= while(expression) compound-statement
for-statement ::= for(expression; expression; expression) compound-statement
                    | for(identifier in expression) compound-statement
match-statement ::= match identifier with match-list
match-list ::= match-conditional compound_statement
                | match-conditional compound_statement match-list
match-conditional ::= (type-identifier | tuple-type
                        | tuple | identifier | _)(when expression)?
```

12 References

12.1 Javascript Language Reference Manual

Tables in Bild are handled in a similar way to the way they are in Javascript. Mainly, keys ares stored as strings (although they can be accessed with integers or strings) and the table mutable.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

12.2 Swift Language Reference Manual

Much of the syntax, especially optional types, is drawn from Swift.

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/zzSummaryOfTheGrammar.html

12.3 OCaml Language Reference Manual

Almost all of the functional aspects of Bild are drawn from OCaml, including types and pattern matching.

http://caml.inria.fr/pub/docs/manual-ocaml/

12.4 Python Language Reference Manual

Type inference (ducking) is taken from the Python language.

https://docs.python.org/2/reference/

12.5 Java Language Reference Manual

Bild uses the Java runtime environment, the Java Virtual Machine, for garbage collection and other runtime responsibilities.

https://docs.oracle.com/javase/specs/jls/se7/html/