# CheckerLang Handbuch

## Einführung

CheckerLang ist eine einfache, aber mächtige Programmiersprache. Ihr Hauptzweck ist das Definieren und Ausführen von Format- und Inhaltsprüfungen. Sie kann aber auch für andere Zwecke eingesetzt werden.

Die Sprache ist darauf ausgelegt, dass einfache und übliche Prüfungen sehr einfach und leserlich geschrieben werden können, dass aber gleichzeitig genügend mächtige Konstrukte zur Verfügung stehen, dass auch komplexe Prüfungen erstellt werden können.

## Tutorial

Skripts bestehen normalerweise aus einem Ausdruck. Der Wert dieses Ausdrucks ist das Resultat der Prüfung. Typischerweise sollte dies ein Boolean sein, es kann aber ein beliebiger Datentyp sein.

Felder werden als Variablen definiert und können direkt angesprochen werden.

V1100 is numerical exact\_len 8

Liefert TRUE, wenn das Feld V1100 einen numerischen Inhalt von genau 8 Zeichen enthält.

V1202 is alphanumerical min\_len 4 max\_len 8

Liefert TRUE, wenn das Feld V1202 einen alphanumerischen Inhalt mit 4 bis 8 Zeichen enthält.

V1103 is in [‘1’, ‘2’]

Liefert TRUE, wenn das Feld V1103 einen der Werte ‘1’ und ‘2’ enthält.

if length(V1104) == 4 then (V1104 + '0101') is date  
if length(V1104) == 8 then V1104 is date  
else FALSE

Liefert TRUE, wenn das Feld V1104 ein gültiges Jahr oder ein gültiges Datum im Format yyyyMMdd enthält.

V1105 in Stammdaten\_PLZ

Liefert TRUE, wenn V1105 in der Liste der PLZ enthalten ist. Dazu muss die Liste der PLZ in der Variablen Stammdaten\_PLZ definiert sein.

V1401 is time

Liefert TRUE, wenn das Feld V1401 eine gültige Zeit im Format HHmm enthält.

0 <= int(V1501) < 1000

Liefert TRUE, wenn das Feld V1501 eine Ganzzahl zwischen 0 und 1000 (exklusive) ist.

V1304 is empty or V1304 in ['1', '3', '4', '5', '6']

Liefert TRUE, wenn das Feld V1304 entweder leer ist oder einen der Werte ’1’, ‘3’, ‘4’, ‘5’ oder ‘6’ enthält.

## Datentypen

String Zeichenketten. Können mit einfachen oder doppelten Anführungszeichen eingeschlossen sein.

Integer Ganzzahl mit Vorzeichen.

Decimal Dezimalzahl.

Date Datumwert inklusive Zeit.

Boolean Boolescher Wert TRUE und FALSE.

Pattern Regulärer Ausdruck. Im Skript werden diese von zwei /-Zeichen vor und nach dem Ausdruck eingeschlossen.

List Liste von Objekten. Eine Liste kann beliebige Datentypen inklusive anderer Listen enthalten.

Set Menge von Objekten ohne Duplikate. Ist effizient, um Mitgliedschaft zu prüfen.

Map Key-Value-Zuordnung. Ist effizient für den Zugriff. Nur-Lesen.

Func Funktion. Es gibt eingebaute Funktionen und selbst definierte Funktionen.

Error Error signalisiert einen Fehler. Zusätzlich kann einen Wert enthalten sein, der den Fehler beschreibt.

## Sprachkonstrukte

|  |  |
| --- | --- |
| do  ausdruck1;  ausdruck2;  …  end | Blockstruktur, mit der mehrere Ausdrücke nacheinander ausgeführt werden können. Auf globaler Ebene ist do-end optional. |
| if bedingung1 then resultat1 if bedingung2 then resultat2 … else resultatn | Wenn-Dann-Kontrollstruktur, mit der für verschiedene Bedingungen verschiedene Resultate erzeugt werden können.  Als resultat1… können entweder einzelne Ausdrücke oder ein Block verwendet werden. |
| for varname in list dosomething | Schleifenkonstrukt, das den Ausdruck einmal für jedes Listenelement ausführt, wobei das Listenelement als varname zur Verfügung steht. Statt einer Liste kann auch ein String verwendet werden.  Sinnvoll vor allem mit der Verwendung von break, continue und return.  Als dosomething kann entweder ein einzelner Ausdruck, oder ein Block verwendet werden. |
| while bedingung do  ausdruck1;  ausdruck2;  …  end | Schleifenkonstrukt, das die Ausdrücke solange ausführt, bis bedingung FALSE ergibt. |
| *Logisch*  bedingung1 or bedingung2  bedingung1 and bedingung2  not bedingung1 | Boolsche Algebra. Dabei hat not höchste Priorität, gefolgt von and und dann or. Somit wird «a or b and not c» interpretiert wie «a or (b and (not c))» |
| *Gleichheit*  wert1 == wert2  wert1 is wert2  wert1 != wert2  wert1 <> wert2  wert1 is not wert2 | Vergleich zweier Werte auf Gleichheit. Die Ungleichheitsoperatoren <> und != sind synonym. |
| *Relation*  wert1 < wert2  wert1 <= wert2  wert1 > wert2  wert1 >= wert2 | Grössenvergleich der Werte. Zwei Vergleiche können auch kombiniert werden: statt «a < b and b <= c» schreibt man «a < b <= c». |
| *Arithmetik*  wert1 + wert2  wert1 - wert2  wert1 \* wert2  wert1 / wert2  wert1 % wert2 | Für Ganzzahlen und Dezimalzahlen gewöhnliche Arithmetik. Der %-Operator stellt für Ganzzahlen den Rest dar (d.h. 3 % 2 == 1). Für Strings sind auch + und \* definiert. Für Listen + und -. |
| *Gruppierung*  ( ausdruck ) | Gruppiert einen Ausdruck, um die Präzedenz zu übersteuern. Beispiel: «(2 + 3) \* 4» |
| *Funktionsaufruf*  fn(arg) | Ruft die Funktion fn mit dem Argument arg auf. Je nach Funktion braucht es mehr oder weniger Argumente. Es gibt auch Funktionen, bei denen Parameter mit Defaultwerten versehen sind. Dann ist das entsprechende Argument nicht nötig. |
| *Dereferenzierung*  ausdruck[index] | Gibt das Element zum Index index zurück. Das ist für Strings und Listen definiert. |
| *Variablendefinition*  def varname = value | Definiert die Variable varname und setzt sie auf den Wert value |
| *Zuweisung*  varname = value | Einer bereits definierten Variablen varname kann ein neuer Wert value zugewiesen werden. |
| *Funktionsdefinition (Lambda)*  fn(arg1, arg2 = 12) body | Erstellt eine Funktion, die zwei Parameter arg1 und arg2 benötigt, wobei arg2 einen Defaultwert von 12 hat. Body kann entweder ein einzelner Ausdruck oder ein Block sein.  Normalerweise werden Funktionen mit dem Def-Konstrukt einer Variablen zugewiesen, es ist aber auch möglich, Funktionen anonym zu verwenden. |
| *Listengenerierung*  [ausdruck for varname in liste]  [ausdruck for varname in liste if bedingung] | Generierung von Listen («Liste comprehension»). Für jedes Element von liste wird der ausdruck evaluiert, wobei das Element an varname gebunden ist. Das Resultat wird in eine neue Liste aufgenommen. Falls eine Bedingung vorhanden ist, wird das Resultat nur aufgenommen, falls die Bedingung wahr ist. |

## Prädikate

Prädikate sind spezielle Sprachkonstrukte, die Bedingungen testen und einen Boolean-Wert zurückgeben. Sie sind definiert, um übliche Prüfungen einfach und gut lesbar zu implementieren. Grundsätzlich könnten sie auch als Funktionen definiert sein.

|  |  |
| --- | --- |
| wert is empty  wert is not empty | Liefert TRUE, wenn wert ein leerer String ist  Liefert TRUE, wenn wert kein leerer String ist |
| wert is negative  wert is not negative | Liefert TRUE, wenn wert < 0 ist  Liefert TRUE, wenn wert >= 0 ist |
| wert is numerical  wert is not numerical  wert is numerical min\_len 4 max\_len 6  wert is numerical exact\_len 5 | Liefert TRUE, wenn wert numerisch respektive nicht numerisch ist. Kann mit ‘min\_len’, ‘max\_len’ und ‘exact\_len’ ergänzt werden, um Längenein­schränkungen festzulegen. |
| wert is alphanumerical  wert is not alphanumerical  wert is alphanumerical min\_len 4 max\_len 6  wert is alphanumerical exact\_len 5 | Liefert TRUE, wenn wert alphanumerisch respektive nicht alphanumerisch ist. Kann mit ‘min\_len’, ‘max\_len’ und ‘exact\_len’ ergänzt werden, um Längenein­schränkungen festzulegen. |
| wert is date | Liefert TRUE, wenn wert ein String im Format ‘yyyyMMdd’ ist, der ein gültiges Datum darstellt. |
| wert is date with hour | Liefert TRUE, wenn wert ein String im Format ‘yyyyMMddHH’ ist, der ein gültiges Datum mit Stunde darstellt. |
| wert is time | Liefert TRUE, wenn wert ein String im Format ‘HHmm’ ist, der eine gültige Tageszeit darstellt. |
| wert starts with ‘a’ | Liefert TRUE, wenn wert mit ‘a’ anfängt. |
| wert ends with ‘a’ | Liefert TRUE, wenn wert mit ‘a’ endet. |
| wert contains ‘a’ | Liefert TRUE, wenn wert ‘a’ enthält. |
| wert matches //a// | Liefert TRUE, wenn wert dem regulären Ausdruck //a// entspricht. |
| wert is in liste | Liefert TRUE, wenn wert in der Liste enthalten ist. |

## Funktionen

Es ist möglich eigene Funktionen zu definieren. Diese werden durch das Schlüsselwort «fn» eingeleitet.

def double = fn(x) 2 \* x;  
double(5) <= 10

* TRUE

Diese können sogar Rekursion nutzen.

def fact = fn(n) if n == 0 then 1 else n \* fact(n - 1);  
fact(10)

* 3628800

Funktionen können auch anonym direkt in Ausdrücken verwendet werden (Lambda-Ausdrücke).

(fn(a, b) string(a) \* b)(123, 3)

* "123123123"

## Eingebaute Funktionen

Die eingebauten Funktionen sind in einem separaten Dokument beschrieben.

## Grammatik

Die Grammatik der Sprache sieht in einer vereinfachten BNF-Syntax wie folgt aus:

script ::= block | expression

block ::= 'do' expression ( ';' expression )\* 'end'

expression ::= 'for' identifier 'in' expression ( block | expression ) |  
'while' or-expr block |  
'def' identifier '=' expression |  
identifier '=' expression |

if-expr

if-expr ::= 'if' or-expr 'then' ( block | or-expr )   
 ( 'if' or-expr 'then' ( block | or-expr ) )\*   
 ( 'else' ( block | or-expr ) )? |  
or-expr

or-expr ::= and-expr ( 'or' and-expr )\*

and-expr ::= not-expr ( 'and' not-expr )\*

not-expr ::= 'not'? rel-expr

rel-expr ::= add-expr ( rel-op add-expr )\*

rel-op ::= '==' | 'is' | '!=' | '<>' | 'is' 'not' | '<' | '<=' | '>' | '>='

add-expr ::= mul-expr ( add-op mul-expr )\*

add-op ::= '+' | '-'

mul-expr ::= unary-expr ( mul-op unary-expr )\*

mul-op ::= '\*' | '/' | '%'

unary-expr ::= ( unary-op )? pred-expr

unary-op ::= '-' | '+'

pred-expr ::= primary-expr 'is' 'not'? 'empty' |  
primary-expr 'is' 'not'? 'zero' |

primary-expr 'is' 'not'? 'negative' |

primary-expr 'is' 'not'? 'numerical' ( 'min\_len' primary-expr )? ( 'max\_len' primary-expr )? ( 'exact\_len' primary-expr )? |

primary-expr 'is' 'not'? 'alphanumerical' ( 'min\_len' primary-expr )? ( 'max\_len' primary-expr )? ( 'exact\_len' primary-expr )? |

primary-expr 'is' 'not'? 'date' |

primary-expr 'is' 'not'? 'date' 'with' 'hour' |

primary-expr 'is' 'not'? 'time' |

primary-expr 'starts' 'not'? 'with' primary-expr |

primary-expr 'ends' 'not'? 'with' primary-expr |

primary-expr 'contains' 'not'? primary-expr |

primary-expr 'matches' 'not'? primary-expr |

primary-expr 'is'? 'not'? 'in' primary-expr

primary-expr ::= identifier deref-or-call |

'(' ( block | expression ) ')' deref-or-call |

list deref-or-call |

literal-string deref? |

literal-int |

literal-decimal |

literal-boolean |

literal-pattern |

lambda |  
'break' |  
'continue' |  
'return' expression |  
'error' expression

deref-or-call ::= ( deref | call )\*

deref ::= '[' expression ']'

call ::= '(' ( expression ( ',' expression )\* )? ')'

list ::= '[' ( if-expr ( ',' if-expr )\* )? ']' |  
'[' or-expr 'for' identifier 'in' or-expr ( 'if' or-expr )? ']'

lambda ::= 'fn' '(' ( identifier ( '=' expression )?   
 ( ',' identifier ( '=' expression )? )\* )? ')'   
 ( block | expression )

literal-int ::= ziffer+

literal-decimal ::= ziffer+ '.' ziffer+

literal-string ::= '"' zeichen\* '"' | ''' zeichen\* '''

literal-boolean ::= 'TRUE' | 'FALSE'

literal-pattern ::= '//' zeichen+ '//'

## Technische Einbindung

Die CheckerLang ist in C#/.NET implementiert und steht als Klassenbibliothek zur Verfügung. Für die eigentliche Sprache wurde .NET Standard 2.0 verwendet. Damit kann die Sprache sowohl für .NET Core, als auch für .NET Framework Anwendungen verwendet werden.

Grundsätzlich wird die Sprache verwendet, indem die Interpreter-Klasse instanziiert wird und danach Scripte interpretiert werden.

using CheckerLang;

...

var interpreter = new Interpreter();

...

string script = "...";

...

EnvironmentMap data = new EnvironmentMap();

data.Put(varname, value);

...

Value result = null;

try

{

result = interpreter.Interpret(script, data);

}

catch (SyntaxError)

{

// todo

}

catch (RuntimeError)

{

// todo

}

// result weiterverarbeiten

Diese Vorgehensweise stellt sicher, dass bei jedem Aufruf der Interpret-Methode ein neues Environment verwendet wird.

Die Solution enthält neben der Sprache selber noch drei Projekte, welche mögliche Anwendungen demonstrieren:

REPL ist eine klassische Shell, in der auf der Kommandozeile CheckerLang-Scripte eingegeben werden können und diese direkt evaluiert werden.

BatchCheck demonstriert, wie man effizient eine grosse Menge Checks (d.h. CheckerLang-Skripte) auf viele Datensätze anwendet. Im Beispielprojekt geschieht das über Dateien. Natürlich können analog sowohl die Checks wie die Datensätze aus der Datenbank genommen werden.

TestApp ist eine kleine GUI-Applikation, in der man Skripte eingeben und ausprobieren kann.

Die Klasse FreeVars kann benutzt werden, um herauszufinden, welche Variablen einem Skript übergeben werden müssen, damit es ausgeführt werden kann. Das kann benutzt werden, um Checks nur dann auszuführen, wenn die benötigten Variablen im Vergleich zum letzten Durchgang geändert haben.

Der folgende Code skizziert die Anwendung dieser Klasse und ihr Zusammenspiel mit der Parser-Klasse.

var node = Parser.Parse(new Lexer(new StringReader("...")));

var freeVars = FreeVars.Get(node, new Interpreter().GetBaseEnvironment());

// set each freeVar in the environment data

...

Value result = node.Evaluate(data);

Die Methode GetBaseEnvironment der Interpreter-Klasse liefert das Environment zurück, das die fest eingebauten (primitiven) Funktionen und die Funktionen der Basis-Library enthält. Diese Funktionen sind natürlich immer definiert und darum nie freie Variablen. Daher muss die FreeVars.Get-Methode Zugriff auf dieses Environment haben.