Spezifikation zur »Moses«-Sprache

Moses (oder Moisej) ist eine funktionale Programmiersprache. Benannt ist sie nach Moses Isajewitsch Schönfinkel (Моисей Исаевич Шейнфинкель).

Ein Moses-Programm besteht aus einem gültigen Moses-Ausdruck, der bei Ausführung des Programmes ausgewertet wird.

Moses hat die folgenden Keywords (eventuell werden einige der Keywords auch einfach gewöhnliche Identifier sein, die über das Prelude geladen werden):

and, false, not, or, true, Unit

Ein Moses-Programm kann Zeilenkommentare enthalten, die mit einem # (Ist es besser, // zu verwenden?) eingeleitet werden, welches nicht Teil eines Literales ist. Jegliche Zeichen zwischen dem Kommentarzeichen und dem Zeilenende werden ignoriert. Es gibt keine mehrzeiligen Kommentare.

Comment = "#", {UnicodeCharacter}, Newline;

Besteht eine Zeile nur aus Unicode-Whitespace, wird sie ignoriert.

Ein Literal hat folgende Struktur (Sollten imaginäre/komplexe Zahlen builtin sein?):

Literal = StringLiteral | CharacterLiteral | NumberLiteral | "unit";

StringLiteral = ('"' | '»'), {UnicodeCharacter}, ('"' | '«');

CharacterLiteral = ("'" | "›"), UnicodeCharacter, ("'" | "‹");

NumberLiteral = IntegerLiteral | FractionLiteral | DecFractionLiteral;

IntegerLiteral = ["-"], NonZeroDigit, {Digit | "\_"} | "0";

NonZeroDigit = "1" - "9";

Digit = "0" | NonZeroDigit;

FractionLiteral = IntegerLiteral, "%", IntegerLiteral;

DecFractionLiteral = ["-"], [IntegerLiteral], ".", IntegerLiteral;

Identifier bezeichnen entweder einen Typ (wenn sie mit einem Großbuchstaben beginnen) oder einen Funktionsnamen (wenn sie mit einem Kleinbuchstaben beginnen). In Identifiern erlaubte Zeichen sind Buchstaben laut Unicode, Zahlen und Unterstriche. Nicht erlaubt sind Keywords (ist Shadowing von Keywords möglich?).

Identifier = TypeIdentifier | FunctionIdentifier;

TypeIdentifier = UpperLetter, {Letter | Digit | \_};

FunctionIdentifier = LowerLetter, {Letter | Digit | \_}

Ausdrücke sind Literale, Blöcke, Definitionen, Operatorausdrücke, Funktionsaufrufe oder Records und können von Typannotationen gefolgt sein.

Expression = Literal | Block | TypeDefinition | FunctionDefinition | OperatorExpression | FunctionCall | Record | TypeExpression, [TypeAnnotation];

Ein Block ist eine Liste von Ausdrücken, der Wert eines Blockes ist der Wert seines letzten Ausdrucks. Ist ein Block leer, so hat er den Wert {} (Es gibt einen Typen Unit, der als einzigen Wert {} enthält). Blöcke dienen der Gliederung von Ausdrücken. Beachte: Eine Moses-Datei muss genau einen Ausdruck enthalten und wird deswegen automatisch als ein Block aufgefasst. (Sollte jeder Ausdruck eines Blockes mit einem Semikolon abgeschlossen werden oder ist es auf einfache und garantiert funktionierende Weise möglich, Trenner zwischen Ausdrücken auszulassen?)

Block = "(", Expression, ")" | "(", {Expression, ";"}, Expression, ")";

Eine Definition dient der Einführung einer neuen Funktion, Konstanten oder eines neuen Typs. Es gibt Summen- und Produktdatentypen. (Die Repräsentation von Typen erfordert eventuell Überarbeitung, damit sie in guter Korrespondenz zu Records steht.)

TypeDefinition = TypeIdentifier, "=", TypeExpression;

TypeExpression = TypeBlock | TypeProd | TypeExpression, "+",

TypeExpression;

TypeBlock = "(", TypeExpression, ")";

TypeProd = TypeIdentifier | TypeExpression, "\*", TypeExpression;

FunctionDefinition = FunctionIdentifier, {Record}, [TypeAnnotation], "=", Expression;

Ein Operatorausdruck besteht aus einem Operator und bei binären Operatoren zwei, bei unären einem Unterausdruck.

OperatorExpression = {Expression}, Operator, Expression;

Die Operatoren sind, nach Präzedenz absteigend sortiert (in der gleichen Zeile stehende Operatoren haben die gleiche Präzedenz):

"." (\* Indizierung, Slicing (Sollte dazu besser [] genutzt werden?) \*) |

"^" (\* Potenzieren \*) |

"\*" | "·" | "/" | "÷" | "mod" |

"+" | "-" |

"<<" | ">>" (\* Shifts \*) |

(\* Bitweise Operatoren fehlen vorerst \*)

"<" | "<=" | "≤" | ">" | ">=" | "≥" | "!=" | "≠" | "==" |

"not" (\* Unärer Operator \*) |

"and" |

"or";

Ein Funktionsaufruf ist ein Identifier einer Funktion, gefolgt von einem Ausdruck als Argument. Ist das Argument unit, kann es weggelassen werden.

FunctionCall = FunctionIdentifier, {Expression};

Ein Record ist ein Element eines Produktdatentypes. Ein Record kann Keyword- und positionale Attribute beinhalten, auf die mithilfe der Indizierung zugegriffen werden kann. Das leere Record ist der Wert des Typs Unit. Ein Attribut ist ein Keywordattribut, falls der zugehörige Ausdruck eine Definition ist; eine positionale 0-basierte Indizierung ist immer möglich.

Record = "{", {Expression, ","}, [Expression], "}":

Typannotationen liefern dem Compiler Hinweise zum Typ des vorhergehenden Ausdrucks und dienen außerdem als Cast, sofern der vorliegende Ausdruck einen Cast in den annotierten Typen zulässt.

TypeAnnotation = ":", TypeIdentifier