e2: Sprachspezifikation, v 1.0

Dieses Dokument beinhaltet eine **informelle** Beschreibung der Programmiersprache e2, für die im Rahmen der Projektübungen zu „Grundlagen des Übersetzerbaus“ ein Übersetzer implementiert werden soll.

1 Lexikalische und syntaktische Struktur

Die Grammatik der Programmiersprache e2 ist Anhang A zu entnehmen und beschreibt die Struktur lexikalisch und syntaktisch gültiger Programme in e2.

Leerraum (Leerzeichen, Tabulatoren, Zeilenumbrüche, ...) trägt keine Semantik. Dies gilt auch für Kommentare, die mit # eingeleitet werden und bis zum Ende der Zeile reichen.

2 Typen

2.1 Primitive Typen

e2 kennt die beiden folgenden primitiven Typen:

- `int`: vorzeichenbehaftete Ganzzahl in Zweierkomplementdarstellung der Größe 64 Bit
- `real`: Fließkommazahl nach IEEE 754 der Größe 64 Bit

Für die Typumwandlungen gilt:

- Bei der Typumwandlung von `int` nach `real` wird die Zahl in eine wertgleiche Zahl umgewandelt (sofern diese als `real` darstellbar ist; andernfalls ist das Verhalten undefiniert). Diese Umwandlung wird *implizit* an genau den Stellen durchgeführt, an denen sie notwendig ist.
- Bei der Typumwandlung von `real` nach `int` werden die Nachkommastellen der Zahl abgeschnitten. Falls das Ergebnis nicht als `int` darstellbar ist, ist das Verhalten undefiniert. Diese Umwandlung kann nur *explizit* durchgeführt werden.

2.2 Zusammengesetzte Typen

Neben den primitiven Datentypen kennt e2 auch die folgenden zusammengesetzten Datentypen:

- Arrays:
  - Die Array-Elemente werden beginnend mit Null indiziert. Ein Zugriff außerhalb der Grenzen des Arrays resultiert in undefiniertem Verhalten.
  - Arrays können beliebig viele Dimensionen haben. Die Anzahl der Indizes beim lesenden oder schreibenden Zugriff muss der Dimensionalität des Arrays entsprechen.
  - Es gibt *keine* dynamischen Arrays, d.h. die Länge jeder Dimension muss ein zur Übersetzungszeit bekannter (ganzzahliger, nicht-negativer) Wert sein. Für die Angabe der Länge kann ein komplexer Ausdruck verwendet werden, wenn dieser gemäß §5.1 zu einer Konstante vereinfacht werden kann.
- Funktionen:
  - Eine Funktion kann beliebig viele Parameter haben. Die Typen der Parameter müssen primitiv sein. Die Typen der beim Aufruf einer Funktion übergebenen Argumente müssen gemäß obiger Regeln zur Typumwandlung kompatibel sein.
  - Eine Funktion kann optional einen Rückgabetyp haben. Dieser muss ebenfalls primitiv sein. Der mittels `return`-Anweisung zurückgegebene Wert muss gemäß obiger Regeln zur Typumwandlung kompatibel sein.

3 Variablen und Sichtbarkeitsschachteln

Globale Variablen werden statisch alloziert. Der Initialwert ist `0` (`int`-Variablen) bzw. `0.0` (`real`-Variablen). Bei globalen Array-Variablen werden die einzelnen Elemente gemäß des Basistyps initialisiert.

Der initiale Wert lokaler Variablen ist unspezifiziert. Wird eine lokale Variable vor dem ersten Schreiben lesend verwendet, ist das Verhalten undefiniert.

Bezüglich der Sichtbarkeit von Variablen und Funktionen gilt:

- Globale Variablen und Funktionen teilen sich einen gemeinsamen Namensraum.
- Die Parameterliste einer Funktionsdefinition erzeugt eine neue Sichtbarkeitsschachtel, ebenso wie der Funktionsrumpf sowie die Blöcke der `if`- und `while`-Anweisungen.
- Eine lokale Variable ist in dem Block, in dem ihre Deklaration steht, sowie allen darin geschachtelten Blöcken gültig. Eine Deklaration in einem inneren Block verdeckt eine Deklaration mit gleichem Namen in einem umgebenden Block.
- Eine Funktion kann lexikalisch vor ihrer Definition aufgerufen werden.

4 Anweisungen

`e2` kennt die folgenden Arten von Anweisungen:

- Zuweisung (`:=`): Auf der linken Seite der Zuweisung muss eine Variable oder ein Array-Ausdruck vom Typ `int` oder `real` stehen. Der Typ des Ausdrucks auf der rechten Seite muss gemäß obiger Regeln zur Typumwandlung kompatibel zum Typ der linken Seite sein. Bei der Ausführung wird zuerst der Ausdruck auf der rechten Seite ausgewertet und das Ergebnis anschließend in die Variable bzw. das Array-Element auf der linken Seite geschrieben.
- Funktionsaufruf: Die Funktion wird mit den übergebenen Argumenten aufgerufen. Das Ergebnis des Funktionsaufrufs (falls vorhanden) wird verworfen.
- `if`-Verzweigung mit optionalem `else`-Teil: Bei der Ausführung wird zunächst die Bedingung ausgewertet. Falls diese erfüllt ist, werden die Anweisungen im `then`-Block ausgeführt. Andernfalls werden (falls vorhanden) die Anweisungen im `else`-Block ausgeführt.
- `while`-Schleife: Bei der Ausführung wird zunächst die Bedingung ausgewertet. Falls erfüllt, werden die Anweisungen im `do`-Block ausgeführt. Dies wird wiederholt, bis die Bedingung nicht mehr erfüllt ist.
- `return`-Anweisung: Der arithmetische Ausdruck wird ausgewertet und das Ergebnis sofort an den Aufrufer zurückgegeben. Bei Funktionen ohne Rückgabetyp entfällt der Ausdruck für den Rückgabewert. Erreicht die Ausführung einer Funktion deren Ende, ohne vorher eine `return`-Anweisung erreicht zu haben, wird implizit ein Standardwert (`0` für Rückgabetyp `int`, `0.0` für Rückgabetyp `real`) zurückgegeben.

Die Ausführung geschieht sequentiell in lexikalischer Reihenfolge (mit Ausnahme der Kontrollstrukturen).

5 Ausdrücke

`e2` kennt die folgenden arithmetischen Ausdrücke:

- Konstanten: Ein Literal mit Punkt (`.`) stellt eine Konstante vom Typ `real` dar, ein Literal ohne Punkt eine Konstante vom Typ `int`.

- Binäre Operationen (`+`, `-`, `*`, `/`) mit erwarteter Semantik: Multiplikation (`*`) und Division (`/`) haben eine höhere Präzedenz als Addition (`+`) und Subtraktion (`-`), die Auswertungsreihenfolge kann jedoch durch Paare runder Klammern beeinflusst werden. Die beiden Operanden müssen einen primitiven Typ haben. Ist mindestens ein Operand vom Typ `real`, so wird der andere Operand ggf. implizit in einen `real`-Wert konvertiert und das Ergebnis ist ebenfalls vom Typ `real`. Andernfalls ist der Typ des Ergebnisses `int`.
- Array-Zugriff (`[]`) zum Lesen oder Schreiben eines Array-Elements: Die Indizes müssen vom Typ `int` sein und ihre Anzahl muss zur Dimensionalität des Arrays passen. Die Index-Ausdrücke werden „von links nach rechts“ ausgewertet. Ein Zugriff außerhalb der Array-Grenzen resultiert in undefiniertem Verhalten.
- Funktionsaufruf: Anzahl und Typen der Argumente müssen zu der Anzahl und den Typen der Parameter bei der Definition der Funktion passen. Die Argumente werden „von links nach rechts“ ausgewertet.
- Typkonvertierung (`as`): Konvertiert das Ergebnis eines Ausdrucks gemäß obiger Regeln in den angegebenen Typ. Sowohl Quell- als auch Zieltyp müssen primitiv sein.

Des Weiteren kennt `e2` die folgenden logischen Ausdrücke:

- Vergleiche (`==`, `!=`, `<`, `<=`, `>`, `>=`) mit erwarteter Semantik: Die beiden Operanden müssen einen primitiven Typ haben. Ist mindestens ein Operand vom Typ `real`, so wird der andere Operand vor dem Vergleich ggf. implizit in einen `real`-Wert konvertiert.
- Verknüpfte Ausdrücke durch logisches „Und“ (`and`) bzw. „Oder“ (`or`): Der `and`-Operator hat eine höhere Präzedenz als der `or`-Operator, die Auswertungsreihenfolge kann jedoch durch Paare runder Klammern beeinflusst werden. Die Auswertung erfolgt nach Kurzschlusssemantik, d.h. der Ausdruck wird nur soweit ausgewertet, bis das Endergebnis feststeht.

5.1 Konstantenfaltung

Während der Übersetzung wird eine einfache Konstantenfaltung durchgeführt, um komplexe arithmetische (Teil-)Ausdrücke zu vereinfachen. Eine binäre Operation der Form „`lo ‘op‘ ro`“ wird genau dann vereinfacht, wenn sowohl `lo` als auch `ro` vom Typ `int` sind und eine der folgenden Bedingungen erfüllt ist:

- `lo` und `ro` sind Konstanten. Die Berechnung kann durch das Ergebnis der Operation ersetzt werden.
- `lo` (bzw. `ro`) ist 0 und die Operation `‘op‘` ist `+`. Die Berechnung kann durch `ro` (bzw. `lo`) ersetzt werden.
- `ro` ist 0 und die Operation `‘op‘` ist `-`. Die Berechnung kann durch `lo` ersetzt werden.
- `lo` (bzw. `ro`) ist 1 und die Operation `‘op‘` ist `*`. Die Berechnung kann durch `ro` (bzw. `lo`) ersetzt werden.
- `ro` ist 1 und die Operation `‘op‘` ist `/`. Die Berechnung kann durch `lo` ersetzt werden.

Die Ausdrücke werden dabei soweit wie möglich vereinfacht.

Das Ergebnis der Vereinfachung muss dem Wert entsprechen, der auch bei der Programmausführung berechnet würde. Dies gilt inbesondere für den Fall eines Überlaufs.

6 Laufzeitbibliothek

Die Ausführung des Programms beginnt stets in der parameterlosen `main`-Funktion mit dem Rückgabetyp `int`. Der Rückgabewert der `main`-Funktion wird als *exit code* an das Betriebssystem zurückgegeben.

Die folgenden Funktionen sind Teil der Laufzeitbibliothek von `e2` und daher implizit definiert:

- `writeChar(int): int`: Gibt das Argument als Zeichen auf der Standardausgabe aus und gibt die Anzahl der ausgegebenen Zeichen zurück (0 bei einem Fehler, 1 sonst).

- `readChar(): int`: Liest ein Zeichen von der Standardeingabe ein und gibt dieses als Ganzzahl zurück.
- `writeInt(int): int`: Gibt das Argument als Dezimalzahl auf der Standardausgabe aus und gibt die Anzahl der ausgegebenen Zeichen zurück (0 bei Fehler).
- `readInt(): int`: Liest eine Ganzzahl in Dezimaldarstellung von der Standardeingabe ein und gibt diese zurück. Dabei werden alle Zeichen bis zur ersten Ziffer oder dem ersten Minuszeichen ignoriert, bevor alle aufeinanderfolgenden Ziffern konsumiert werden. Das Verhalten bei einem Überlauf ist undefiniert.
- `writeReal(real): int`: Gibt das Argument in Kommadarstellung auf der Standardausgabe aus und gibt die Anzahl der ausgegebenen Zeichen zurück (0 bei Fehler).
- `readReal(): real`: Liest eine Fließkommazahl in Kommadarstellung von der Standardeingabe ein und gibt diese zurück. Dabei werden alle Zeichen bis zur ersten Ziffer oder dem ersten Minuszeichen ignoriert, bevor alle aufeinanderfolgenden Ziffern (sowie ggf. ein Punkt zur Trennung der Nachkommastellen) konsumiert werden. Das Verhalten bei einem Überlauf ist undefiniert.
- `exit(int): int`: Beendet das Programm sofort mit dem angegebenen *exit code*.
- `time(): int`: Gibt die Systemzeit als Zeitstempel mit Millisekunden-Auflösung zurück.

7 Korrektheit der Implementierung

Eine korrekte Implementierung eines Übersetzers für `e2` muss die folgenden Bedingungen erfüllen:

- Ein Programm, das die syntaktischen Regeln von `e2` verletzt, muss bei der Übersetzung zu einer Fehlermeldung führen.
- Ein Programm, das mindestens eine der oben aufgeführten semantischen Regeln verletzt, muss bei der Übersetzung zu einer Fehlermeldung führen. Dies gilt *nicht* für Programme, die lediglich undefiniertes Verhalten aufweisen.
- Ein Programm, das keine der obigen Regeln verletzt und kein undefiniertes Verhalten aufweist, darf bei der Übersetzung zu keiner Fehlermeldung führen und das beobachtbare Verhalten seiner Ausführung muss dieser Sprachspezifikation genügen.

8 Beispiel

Folgendes Beispielprogramm berechnet die 50. Fibonacci-Zahl und gibt diese auf der Standardausgabe aus:

```
var mem : int[51];

func fib(n : int): int
  if n == 0 or n == 1 then
    return n;
  end
  if mem[n] == 0 then
    mem[n] := fib(n-1) + fib(n-2);
  end
  return mem[n];
end

func main(): int
  writeInt(fib(50));
  writeChar(10);
  return 0;
end
```

A Grammatik

```
program: ( globalDeclaration )* EOF ;

globalDeclaration: variableDeclaration | functionDeclaration ;

variableDeclaration: 'var' IDENTIFIER ':' typeName ';' ;

functionDeclaration
  : 'func' IDENTIFIER '(' ( parameterDeclaration ( ',' parameterDeclaration )* )? ')'
      ( ':' typeName )? block 'end' ;

parameterDeclaration: IDENTIFIER ':' typeName ;

typeName: primitiveTypeName ( '[' arithmeticExpression ']' )* ;

primitiveTypeName: 'int' | 'real' ;

block: ( variableDeclaration )* ( statement )* ;

statement: functionCallStatement | assignStatement | ifStatement | whileStatement | returnStatement ;

functionCallStatement: functionCall ';' ;

assignStatement: lvalue ':=' arithmeticExpression ';' ;

ifStatement: 'if' conditionalExpression 'then' block ( 'else' block )? 'end' ;

whileStatement: 'while' conditionalExpression 'do' block 'end' ;

returnStatement: 'return' ( arithmeticExpression )? ';' ;

lvalue: IDENTIFIER | arrayAccess ;

conditionalExpression: orExpression ;

orExpression: andExpression ( orExpressionRest )* ;

orExpressionRest: 'or' andExpression ;

andExpression: compareExpression ( andExpressionRest )* ;

andExpressionRest: 'and' compareExpression ;

compareExpression
  : arithmeticExpression ( '==' | '!=' | '<' | '<=' | '>' | '>=' ) arithmeticExpression
  | '(' conditionalExpression ')'
  ;

arithmeticExpression: additiveExpression ;

additiveExpression: multiplicativeExpression ( additiveExpressionRest )* ;

additiveExpressionRest: ( '+' | '-' ) multiplicativeExpression ;

multiplicativeExpression: factor ( multiplicativeExpressionRest )* ;

multiplicativeExpressionRest: ( '*' | '/' ) factor ;

factor : variableAccess | numberLiteral | functionCall | arrayAccess | '(' arithmeticExpression ')' | '('
    arithmeticExpression 'as' primitiveTypeName ')' ;

variableAccess: IDENTIFIER ;

numberLiteral: NUMBER | CHAR_LITERAL ;

functionCall: IDENTIFIER '(' ( argument ( ',' argument )* )? ')' ;

argument: arithmeticExpression ;

arrayAccess: IDENTIFIER ( '[' arithmeticExpression ']' )+ ;

WHITESPACE: (' ' | '\t' | '\r' | '\n'); // ignored
COMMENT: '#' ~('\n'|'\r')* '\r'? '\n';  // ignored

NUMBER: [0-9]+ ('.' [0-9]*)? ;
CHAR_LITERAL: '\'' [ -~] '\'';

IDENTIFIER: [a-zA-Z_]+[a-zA-Z0-9_]*;
```