CpiB: Zwischenbericht Strings (UTF32)

Patrick Walther / Claude Martin

Abstract

Unsere Erweiterung führt Strings in IML ein. Dies sind Zeichenketten mit beliebigen Unicode-Zeichen. Einzelne Zeichen werden gleich wie Integer behandelt. Für die Verkettung ist auch eine Datenstruktur ähnlich zu Arrays nötig. Im Source-Code sind Strings zwischen zwei "-Zeichen (x22) definiert. Als Escape-Zeichen wird ein Backslash verwendet.

Lexikalische Syntax

Ein neues Attribut für String-Literale wird eingeführt. Ausserdem sind Brackets nötig und es werden zwei neue Terminale für die Stringlänge eingeführt.

```
(LITERAL, "...")
LBRACKET
RBRACKET
MAXLEN
STRLEN
```

Die vom Scanner erzeugten String-Token können aus dem String bereits eine Darstellung als UTF-32 erzeugen.

Grammatikalische Syntax

Deklaration:

```
var strIdent1 : string;
var strIdent2 : string;
var strIdent3 : string;
```

Initialisierung:

```
strIdent1 init := [6];
strIdent2 init := "bla";
strIdent3 init := strIdent1 + strIdent2;
```

Oder:

```
call testFunction(strIdent1 init [6]);
```

Zugriff auf einzelne Zeichen:

```
strIdent2[2] := strIdent2[3]; // strIdent2 ist nun „baa“
strIdent2[1] := 97; // strIdent2 ist nun „aaa“
```

Länge auslesen:

```
strIdent1.maxlen; // Maximal 6 Zeichen möglich
strIdent1.strlen; // 0, da keine Zeichen eigefügt wurden.
strIdent2.strlen; // 3 Zeichen
```

Änderungen an der Grammatik

Zusätzlich zum normalen „factor“ wurde ein neuer „arrFactor“ erstellt und die oben erwähnten Terminale ergänzt.

```
arrFactor := LBRACKET expr RBRACKET
```

Durch diese Änderung ergibt sich der folgende neue „factor“.

```
factor := LITERAL
        | arrFactor
        | IDENT [INIT [arrFactor] | exprList | arrFactor | MAXLEN | STRLEN]
        | monadicOpr factor
        | LPAREN expr RPAREN
```

Kontext-Einschränkungen

Das Resultat der „expr“ zwischen „LBRACKET“ und „RBRACKET“ muss ein IntVal sein. Dieser wird erst zur RunTime auf > 0 und <= MAXLEN geprüft.

Die Terminale „MAXLEN“ und „STRLEN“ dürfen nur auf eine Variable des Typs string folgen.

Das Nichtterminal „arrFactor“ darf nur auf eine Varible des Typs string oder auf die Initialisierung einer solchen folgen.

Die verschiedenen „FLOWMODE“, „MECHMODE“ und „CHANGEMODE“ werden von unseren strings vollständig unterstützt. Speziell ist der Wert bei Index 0. Dieser ist nicht überschreibbar.

UTF-32

Da Unicode vollständig unterstützt werden soll, muss eine Codierung verwendet werden, die alle diese Zeichen speichern kann. Dazu verwenden wir UTF-32.

Nachteil:

- Enormer Speicherverbrauch. 11 Bits pro Zeichen werden nicht verwendet.

Vorteil:

- Sehr einfaches Abzählen der einzelnen Zeichen:
  Anzahl Zeichen = Anzahl Integer = Anzahl Bytes * 4

String-Länge

Wir unterscheiden zwei Längen:

- MAXLEN: Länge des Arrays — maximale Anzahl möglicher Zeichen.
- STRLEN: Zeichenlänge des Strings — Anzahl tatsächlicher Zeichen.

Die Länge zu speichern hat diverse Vorteile. Zum einen kann zur Laufzeit die Länge einfach ausgelesen werden, ausserdem kann ein unnötiger Speicherverbraucht reduziert werden, da Strings von passender Grösser erzeugt werden können. Trotzdem sind sehr lange Strings möglich. Die maximale Grösse von 2^{32}-1 Zeichen würde 16 GiB benötigen. Somit entscheiden wir uns für diese Lösung. Abfrage ist möglich durch das Suffix „`.maxlen`“.

Die Anzahl der tatsächlichen Zeichen muss abgezählt werden. Zur Laufzeit kann nach dem ersten Zeichen welches 0 ist gesucht werden, aber maximal bis „`.maxlen`“. Abfrage durch Suffix „`.strlen`“. Es wird garantiert, dass „`.strlen`“ nie länger ist als „`.maxlen`“.

Da Unicode auch die Möglichkeit bietet diakritische Zeichen (z.B. Umlaute) einzeln zu speichern, ist der Vorteil in diesem Falle nicht mehr so gross. Lediglich die Anzahl Codepoints kann direkt mit „`.strlen`“ ausgelesen werden.

Indexierung und Iteration

Das erste Zeichen im String wird per Index 1 ausgelesen, das letzte ist auf Position „.maxlen“. Index 0 ist die Speicherposition der maximalen Länge. Somit ist „`.maxlen`“ lediglich ein Alias für `[0]`.

```
program strlen
global
   var idx : integer;
   var str : string;
{
   str init := "bla";
   idx init := 1;
   while idx <= str.strlen {
      ! str[idx];
   }
}
```

Escape-Sequenzen

Als Escape-Zeichen wird ein Backslash verwendet. Unterstützt werden folgende Zeichen: `\t`, `\n`, `\r`, `\b`, `\f` und `\R`

Die Bedeutungen sind gleich wie in Java. Hinzu kommt noch `\R`, welches mit `System.getProperty("line.separator")` ersetzt wird. Das heisst, dass zur Compile-Zeit ein für das System üblicher Zeilenumbruch eingesetzt wird.

Speicherbelegung

Bei einer Initialisierung durch ein String-Literal ist der Speicherverbrauch:
4 Bytes + (Anzahl Code-Points × 4 Bytes)
Die Byte-Reihenfolge ist Big-Endian und gleich wie beim 32-Bit-Integer.

- *Leere Zeichenkette:*
  `00000000`
- „A“:
  `00000001 00000041`
- „日本語“:
  `00000003 000065E5 0000672C 00008A9E`
- „å“:
  `00000001 000000E5 (å)`
  `00000002 00000061 0000030A (a + ˚)`

Operationen

Operation	Syntax	Beschreibung
Maximal-Länge	`str.maxlen`	Max. mögliche Zeichen.
String-Länge	`str.strlen`	Anzahl Zeichen bis 0-Terminierung.
Index-Zugriff	`str[i]`	1-basierter Array-Zugriff.
Konkatenation	`s3 := s1 + s2;`	Zusammenführen zweier Strings. *
Vergleich	`s1 = s2`	True, wenn beide äquivalent.
Ein-/Ausgabe	`? str;` `! str;`	Eingabe oder Ausgabe per Terminal.

* Ob diese „in-place“ zusammengeführt werden, oder ein Hilfs-String verwendet wird, muss noch genau festgelegt werden.

Vergleich zu anderen Programmiersprachen

Die sehr weit verbreitete Syntax für Strings mit Anführungszeichen und Backslash als Escape-Zeichen ist bei vielen Sprachen gleich. Als Datentyp für einzelne Zeichen (Character) wird einfach ein 32-Bit-Integer verwendet. Das

Vorzeichen spielt sowieso keine Rolle, so dass hier nicht unterschieden wird.

Vergleich zu Java

Die Deklaration von Strings im Source-Code ist identisch. In Java werden Strings jedoch nicht als Array behandelt sondern als Objekt (*String* oder andere Implementation vom *CharSequence*-Interface).

Java verwendet hierbei durchwegs UTF-16, mit dem Nachteil, dass die Anzahl Codepoints nicht der Länge des Arrays entspricht. Die möglichen Zeichen sind jedoch die gleichen.

In Java steht durch die String-Klasse eine grosse Auswahl an Methoden für Strings zur Verfügung. Dies bieten wir nicht an. Die Grundoperationen würden jedoch ausreichen solche Methoden zu schreiben.

Vergleich zu C

Die Verwendung von Strings im Source-Code ist ähnlich. Da es bei C üblich ist Pointer zu verwenden ist dies anders.

In C kann grundsätzlich jede Zeichenkodierung verwendet werden. So entspricht unsere Lösung der Verwendung von char32_t. Die Terminierung von Strings mit 0 ist sehr ähnlich. Da wir jedoch die maximale Länge speichern wird verhindert, dass zu weit gelesen wird.

Vergleich zu Delphi (Pascal)

Delphi verwendet eine andere Syntax für Strings. Es wird das einfache Anführungszeichen verwendet und zur Escape-Sequenz wird einfach das Zeichen doppelt angegeben.

Delphi kennt bei kurzen Strings auch den Ansatz die Länge am Anfang der Kette zu speichern. Delphi verhindert, dass dieser Wert manipuliert wird, so dass dieser immer mit der tatsächlichen Länge übereinstimmt. So beginnt ein String bei Delphi auch bei Index 1. Unsere Umsetzung ist im Ansatz ähnlich. Doch wir speichern die maximale Länge, während diese bei Delphi diese immer 255 ist und das erste Byte die String-Länge enthält.

Später wurden in Delphi Strings mit „Copy-On-Write“ und Referenzzähler für Garbage-Collection eingeführt.

Delphi-Beispiel:

```
use System;
// …
{$LongStrings Off}
var s : string; i : integer;
// …
s := 'Delphi';
s := Copy(s,2,3);
// s = 'elp';
i := Pos('hi',s);
// i = 5;
if i = 0 then
  ShowMessage('''hi'' wurde nicht gefunden.');
```

Unicode Zeichen als Alias für Standard-Symbole.

Beim Scannen werden gewisse Operatoren vom Scanner bereits erkannt und gleich wie die Standard-IML-Schreibweise eingelesen. Da die selben Token erzeigt werden, spielt es nach dem Scannen keine Rolle mehr was im Source-Code ist.

Folgende Aliase kann der Scanner bereits erkennen:

Terminal/Attribute	Standard-IML	Alias
CAND	`CAND`	∧
COR	`COR`	∨
BECOMES	`:=`	≔
GE; LE; NE	`>=; <=; /=`	≥; ≤; ≠
FUN	`FUN`	*ƒ*
DIV; TIMES	`DIV; *`	÷; ×
QUESTMARK; EXCLAMARK	`?; !`	→; ←

Aufteilung der Arbeit

Die einzelnen Programmieraufträge wurden abwechslungsweise vom einen Studenten implementiert und vom anderen wurden dann die Tests dazu geschrieben.

Die Konzepte und Inhalte wurden immer gemeinsam erarbeitet. Mit diesen Massnahmen haben wir sichergestellt, dass beide Studenten immer auf dem gleichen Wissensstand sind.

Zusammenarbeit

Für die Arrays werden wir mit dem Team Rao/Giedemann zusammenarbeiten.

Jedoch benötigen wir nur eindimensionale Arrays.

Anhang: Code-Beispiele

Hallo Welt!

```
program HalloWelt
{
   ! "Hallo Welt!"
}
```

Hallo User!

```
program HalloUser
global
   var name : string
{
   ! "Wie heisst du?";
   name init := [32];
   ? name;
   ! "Hallo " + name + "!"
}
```

GROSSBUCHSTABEN

Standard-IML

```
program GROSSBUCHSTABEN
global
   var idx : int32;
   var len : int32;
   var chr : int32;
   var str : string
{
   ! "Geben Sie einen Text ein.";
   str init := [255];
   ? str;
   idx init := 1;
   len init := str.strlen;
   while (idx <= len) {
      chr := str[idx];
      if (((chr >= 97) CAND (chr <= 122)) COR
         ((chr >= 224) CAND (chr <= 254) CAND (chr /= 247))) {
         str[idx] := chr - 32
      }
      idx := idx + 1
   }
   ! str
}
```

Unter Verwendung von Unicode-Symbolen:

```
program GROSSBUCHSTABEN
global
   var idx : int32;
   var len : int32;
   var chr : int32;
   var str : string
{
   ← "Geben Sie einen Text ein.";
   str init ≔ [255];
   → str;
   idx init ≔ 1;
   len init ≔ str.strlen;
   while (idx ≤ len) {
      chr ≔ str[idx];
      if (((chr ≥ 97) ∧ (chr ≤ 122)) ∨
         ((chr ≥ 224) ∧ (chr ≤ 254) ∧ (chr ≠ 247))) {
         str[idx] ≔ chr - 32
      }
      idx ≔ idx + 1
   }
   ← str
}
```