IML Compiler mit Spracherweiterung: Datentyp String

Pascal Büttiker
Samuel Stachelski

FHNW, Modul Compilerbau, 2013

18. November, 2013
Version 0.1

Abstract

Dieses Paper beschreibt die Entstehung eines IML Compilers inklusive einer IML-Erweiterung (Datentyp string). Das Ziel dieses in C++ geschriebenen Compilers ist es, IML Source-Code in Assembler (NASM) Code zu kompilieren, welcher danach wiederum für die jeweilige Platform in Maschinencode übersetzt wird.

Unsere Erweiterung führt den neuen Datentyp "string" ein. Strings sollen ASCII-Zeichen speichern können. Ein String muss zwar intern in einer From von einem Array gespeichert werden, aber da ist nicht relevant für die Anwendung der Spracherweiterung: Ein String ist ein homogener Datentyp, der nicht auf anderen Datentypen von IML basieren soll. Dementsprechend braucht er z. B. auch eigene Operatoren. Ausserdem soll es möglich sein, einen Ausdruck eines beliebigen Typs in einen String einzubauen, mittels Inline-Syntax.

Aufbau

Das Paper ist in zwei Hauptteile gegliedert, zum einen in das Konzept für die IML-String Erweiterung zum anderen gehen wir auf technische Eigenheiten und Herausforderungen ein, auf die wir beim Implementieren gestossen sind.

IML- String Erweiterung

Motivation

Mit Texten, bzw. Zeichenketten zu arbeiten ist zentraler Bestandteil von Programmen, z.B. wenn diese eine Interaktion mit Menschen haben. In IML gibt es bisher keinen Datentyp für Zeichenketten, daher möchten wir einen Datentyp „String“ einführen. Einen String soll man mit anderen Ausdrücken / Variablen beliebigens Typs zusammenhängen können, wobei daraus wieder ein String entsteht. Der Fokus steht dabei auf der Usability für den Programmierer, er soll mit Strings möglichst angenehm sowie effektiv arbeiten können.

Datenstruktur

Für uns war es essentiell, dass String ein elementarer Datentyp ist, also nicht auf anderen Konstrukten der Sprache basiert. In den meisten modernen Programmiersprachen verwendet man eine Kombination von Characters und Arrays, um Strings zu repräsentieren. Wir finden es deshalb viel interessanter, einen homogenen Datentyp String einzuführen der vollkommen in die Sprache integriert ist und (je nach Komplexität des Typs) keine Änderungen an der Sprachdefinition erfordert, bzw. nicht mehr als beispielsweise die Einführung eine numerischen Datentyps.

Funktionalität

Unser Ziel ist es, dem Programmierer zu ermöglichen Nachrichten zu erstellen und Auszugeben. Ausserdem soll er beliebige Variablen in Strings um wandeln und aneinander hängen können. Modifikationen von Strings sind kein Ziel, deshalb beschränken wir uns auf das Implizite Casting, den neuen &-Operator und die Inline-Syntax.

Lexikalische Syntax

Folgende neue Zeichen müssen dem Alphabet der IML Sprache hinzugefügt werden:

STRCONCATOPR	&
STRING_START / END	"
KEYWORD_INLINE_START / END	%

Da wir einen separaten Opterator für die Stringkonkatenation einführen werden, müssen wir für die Expressions einen zusätzlichen Term definieren:

```
(expr,
   [[N strTerm, N repConcOprExpr]]),
(repConcOprExpr,
   [[T STRCONCATOPR, N strTerm, N repConcOprExpr], []]),
(strTerm,
   [[N term1, N repBoolOprTerm1]]),
```

Grammatikalischer Syntax

Deklaration

```
var myString1 : string;
const myString2 : string;
var myString3 : string;
var number1: int32;
```

Initialisierung

Statischer String
```
myString1 init := "schnurzel ";
```

Mit anderen Variablen
```
myString2 init := myString1 + number1;
```

Im Funktionsaufruf
```
call foo(myString3 init);
```

Ein-/Ausgabe:
```
? myString1;
! myString1;
```

Konkatenation

Grundsätzlich erlaubt der neue binäre Verknüpfungsoperator (&) das Verknüpfen von zwei (beliebigen) Expressions zu einem neuen String:

string := <EXPR> & <EXPR>

Sollte eine <EXPR> nicht vom Typ String sein, wird diese implizit zu einem String gecasted.

```
myString1 := "dum" & "di";
myString1 := "dum" & number1;
myString1 := "foo" & (10 + 2);
myString1 := string1 & string2;
```

Der &-Operator wandelt jede Expression in einen String um, auch wenn kein Teil der Operanden ein String ist. Das ganze an einem Beispiel veranschaulicht:

```
myString1 := 5 & (4 + 5);
```

Wird zu einer gecasteten Konkatenation:

```
myString1 := cast_str(5) & cast_str(4 + 5);

myString1 := "5" & cast_str(9);
myString1 := "5" & "9";
myString1 := "59";
```

Inline-Syntax

Die Inline-Syntax ist ein „Syntactic-Suggar“ für String-Konkatination, und hat demnach das Ziel, eben diese zu vereinfachen. Anstatt dass man einen String mühselig über Konkatenation zusammenbauen muss kann man das gleiche über eine viel gänigere Syntax erreichen.

Ohne Inline-Syntax:

```
myString1 := "Hello " & name & " your age is " & age + 10;
```

Dies lässt sich mit der Inline-Syntax mit weitaus weniger kryptisch, äquivalent ausdrücken:

```
myString1 := "Hello %name% your age is %age + 10%!";
```

Um ein Prozent-Zeichen in einem String darzustellen muss man es lediglich verdoppeln.

```
myString1 := "We have 90%% of it!";
```

Die Strings werden bei Kompilierungszeit auf Inline-Syntax analysiert und ausgewertet, d.h. die Auswertung ist nicht dynamisch und ist während der Laufzeit nicht mehr verfügbar.

Kontext- und Typ-Einschränkung

Es gibt für alle Datentypen die Möglichkeit, eine Variable zum Datentyp String zu konvertieren. Anwendung findet diese Konvertierung nur in Expressions und nur implizit. Ansonsten würden die Freiheiten z. B. bei den Prozedur- und Funktionsaufrufen eingeschränkt mit dem Typ String (z. B. ist dann ein Aufruf mir Referenzparametern nicht mehr klar definiert).

Zeichensatz

Es wird der ASCII-Zeichensatz unterstützt, wobei nicht-druckbare Zeichen ignoriert werden.

Maximale Länge

Die maximale Länge wird provisorisch auf 256 Zeichen festgelegt. Zu lange Strings werden (wie bei einem int-Overflow) einfach abgeschnitten.

Speicherverbrauch

Pro String-Variable wird ein von Anfang die maximale Länge eines Strings reserviert. Dies verbraucht zwar Platz, erleichtert die Handhabung der Speicherverwaltung dafür extrem. Abgesehen von dynamischer Speicherallokation wäre eine einfache erste Optimierung, zumindest bei statischen Strings nur die benötigte Länge zu reservieren.

Vergleich mit anderen Programmiersprachen

Kriterium	Java	C
Realisierung des Datentyps String	Klasse, welche auf dem Grunddatentyp char-Array aufbaut	Typ String existiert nicht, in den Standard-Bibliotheken wird String als ein char-Array betrachtet, wobei der Wert 0 das Stringende kennzeichnet
Operationen auf Strings	Syntax ähnlich wie bei uns: Konkatenation mittels +-Operator, dabei implizite Konvertierung von anderen Datentypen Sehr viele Funktionen in der Klasse String	Standard-Bibliothek mit diversen Funktionen für char-Arrays
Inline-Syntax	Nicht unterstützt	Nicht unterstützt
Maximale Länge	Länge des verfügbaren Arbeitsspeicher dividiert durch 2 (da UTF-16) oder 2^31 Zeichen	Länge des verfügbaren Speichers
Speicherverbrauch	2 Bytes pro Zeichen + Overhead der Klasse	1 Byte pro Zeichen

Implementation

Unsere Implementation ist in C++ gehalten, und produziert aus IML Source-Code NASM Assembler Source Code. Dieser NASM Code kann dann für jede Platform nativ kompiliert werden.

Übersicht

Lexikalische Analyse

Wir haben uns entschieden, das String-Inline Feature (Inline-Syntax) direkt im Lexxer zu implementieren. Konkret bedeutet das, dass Strings die Inline-Syntax enthalten bei der Lexikalischen Analyse direkt in String-Konkatenation übersetzt werden.

Der Lexxer arbeitet mit verschiedenen States, dazu gehören:

STATE_EXPRESSION	Normaler State für Code. /* --> STATE_COMMENT_MULTI // --> STATE_COMMENT_LINE " --> STATE_LITERAL_STRING
STATE_COMMENT_LINE	Einzeiliger Kommentar NewLine --> <Previous-State>
STATE_COMMENT_MULTI	Mehrzeiliger Kommentar */ --> <Previous-State>
STATE_LITERAL_STRING	Normaler Text % --> STATE_INLINE_STRING " --> STATE_EXPRESSION
STATE_INLINE_STRING (extends STATE_EXPRESSION)	Inline-Expression % --> STATE_LITERAL_STRING

```
MyString1 := "10 + 5 = %10+5%";
```

Das Statement wird vom Lexxer wie folgt interpretiert:

Die Analyse verläuft normal bis zum ersten Quote. Beim ersten Quote (") wird in den String-Context gewechselt, d.h. alle folgenden Zeichen werden als String-Bestandteil behandelt, bis entweder eine abschliessendes Quote vorkommt oder mit dem Inline-Keyword (%) ein Kontext-Wechsel erzwingt wird.

Beim treffen auf das %-Inline-Keyword im String passiert also folgendes:

1. Der aktuelle String wird als beendet interpretiert.
2. Es wird ein &-Konkatenations Operator eingefügt
3. Es wird in in den STATE_INLINE_STRING gewechselt. (Normale Expression-Analyse)

Parser (Concrete-Syntax Tree Generator)

Aus unserer angepassten IML Grammatik haben wir mit „Fix & Foxi“ eine Parse-Tabelle generiert.

Hier ein Ausschnitt aus der generierten Parse-Tabelle:

	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			e	e

Man könnte nun für jede Zeile eine Klasse definieren, und diese nach folgendem Schema implementieren:

```
// E2 := E'
void E2() throws GrammarError {
	if (terminal == '+') {
		println("E' ::= +TE' ");
		consume('+');
		T();
		E2();
	} else if (terminal == ')' || terminal == '$') {
		println("E' ::= e");
	} else
		throw new GrammarError("E' ");
}
```

Wie man unschwer erkennen kann weist dieser Code höchste Redundanzen auf bezüglich seiner Struktur, und wenn man 50 Zeilen in der ParserTabelle hat ist auch ein gewisser (langweiliger) Aufwand damit verbunden, diese in diese Klassenform zu bringen.

Eine erste Idee dies zu optimieren ist, den Code automatisch zu generieren. Nach unserer Ansicht erübrigt dieses Vorgehen zwar das Schreiben von Boilerplate Code, trifft aber nicht des Pudels Kern: Die Code Struktur wird nach wie vor höchst redundant sein.

Daher ist unser Ansatz, diese Struktur selber als Klassen-Modell zu designen, und dann anhand einer Parse-Tabelle dynamisch zur Laufzeit zu erstellen.

Dynamischer Parser (Concrete Syntax-Tree Generator)

Die Parse-Tabelle wird als Datengrundlage von unserem Compiler zur Laufzeit eingelesen, und daraus dann ein dynamischer Parser erstellt:

IGrammarSymbol: Represents a TS or NTS
=>
NonTerminal implements IGrammarSymbol
NonTerminal.ProductionRule (: IProductionRule)
and
Terminal implements IGrammarSymbol

D.h. Terminal-Symbole (TS) und Nicht-Terminal-Symbole (NTS) sind beide Ableitungen von IGrammarSymbol, was uns ermöglicht das Composite-Pattern zu nutzen:

Jedes NTS hat eine eigene ProductionRule, was einer Zeile in der Parse-Tabelle entspricht.

```
interface IProductionRule {
  IParseTree produce(IContext ctx);
}
```

Die ProductionRule Klasse ist das Herzstück unserer dynamischen Implementation und sieht ca. so aus:

```
class ProductionRule
{
  NonTerminal nonterminal;
  Map<Terminal, List<IGrammarSymbol>> productionTable;

  public ISyntaxTree produce(IContext ctx){
    ISyntaxTree node = new SyntaxNode(nonterminal);
    List<IGrammarSymbol> production = productionTable[ctx.terminal];
    for(IGrammarSymbol s : production){
      if(s instanceof Terminal){
        if(ctx.terminal.equals(s)){
          ctx.consume(ctx.terminal);
          node.addChild(new SyntaxLeaf((Terminal)s));
        }
      }else if(s instanceof NonTerminal){
        ISyntaxTree sub = s.ProductionRule.produce();
        node.addChild(sub);
      }
    }
    return node;
  }
}
```

Assembler / „Assemblizer“

TODO