Handbuch zum bmcc Compiler

Benjamin Dittes & Matthias Lange

7. November 2004

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument stellt das Handbuch zum bmcc Compiler dar. Dieser Compiler entstand im Zeitraum von April 2004 bis Juli 2004 im Rahmen des Hauptseminars Compilerbau an der TU-Dresden.

Im ersten Teil dieses Handbuchs folgt eine allgemeine Beschreibung des Projekts und eine Darstellung des Funktionsumfangs. Im zweiten Teil wird die eigentliche Bedienung des Programms näher erläutert. Im abschließenden dritten Teil wird beschrieben, wie mit den vorhandenen Möglichkeiten eigene Spezifikationen für den Compiler geschrieben werden können.

2 Allgemeine Beschreibung

Im Rahmen des Hauptseminars Compilerbau sollte ein Compiler für die Sprache PL/0 entwickelt werden. Vorgaben über den Funktionsumfang, die Benutzung von externen Tools (z.B. yacc und lex) oder andere Restriktionen gab es nicht. Ziel war es am Ende ein PL/0 Quellprogramm lexikalisch und syntaktisch zu analysieren und dann irgendeine Form von Zielcode zu erzeugen.

Schnell fassten wir den Entschluss einen Compiler zu schreiben, der von der Quellsprache und dem zu erzeugenden Zielcode unabhängig ist. Vielmehr sollten diese beiden Dinge durch ein externes XML-File spezifiziert werden und der Compiler anhand der Regeln dieser Datei arbeiten.

Das ist uns am Ende, wenn auch mit wenigen Abstrichen, auch gelungen. So ist es uns mit dem aktuellen Stand möglich, ein PL/0 Quellprogramm z.B. in C++ oder C# zu übersetzen. Auf diese Dinge wird Kapitel 4 noch näher eingehen.

Als Entwicklungsplattform nutzten wir Mac OS X 10.3. Der Compiler ist auch nur auf Systemen mit mindestens Mac OS X 10.3 lauffähig. Alle älteren Systeme werden nicht unterstützt.

3 Bedienungsanleitung

Um den Compiler nutzen zu können braucht man prinzipiell nur zwei XML-Dateien: eine Spezifikation der Grammatik (inkl. Semantikspezifikation) und eine lexikalische Spezifikation. Wie man eigene Spezifikationen erstellt, erklärt Kapitel 4 dieses Handbuchs. Dieser Distribution liegt eine Spezifikation für PL/0 bei, so dass man sofort starten kann.

3.1 Laden einer Grammatik

Um mit dem Programm sinnvoll arbeiten zu können muss zunächst eine Grammatik geladen werden. Dazu wähl man im File Menü den Punkt „Open Grammar...“. Das ist auch in Abbildung 1 dargestellt. Nun muss man zu einem XML-File navigieren welches die Grammatikspezifikation

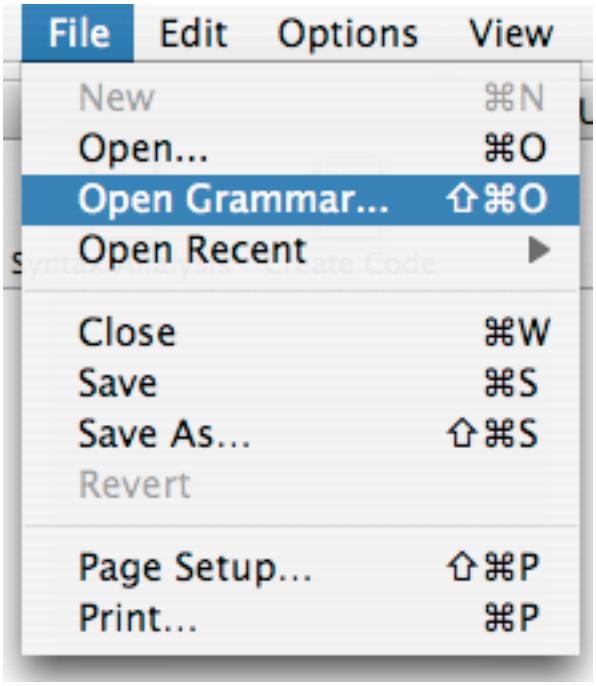

Abbildung 1: Laden einer Grammatik

enthält und diese öffnen. Bmcc lädt jetzt die Spezifikation. Um diese anzuzeigen muss man im View Menü auf „Show Grammar“klicken (Abbildung 2).

3.2 Durchführen der lexikalischen Analyse

Nachdem man erfolgreich eine Grammatik-Spezifikation geladen hat, kann man die lexikalische Analyse durchführen. Dazu bedarf es natürlich eines zu analysierenden Quellprogramms. Zum einen kann man ein bereits existierendes Quellprogramm aus einer Datei laden, zum anderen kann man eins in das Textfeld des Editors eingeben und dieses später bei Bedarf sichern. Dateien mit bereits existierenden Quellprogrammen müssen die Dateierweiterung .pl0 tragen, da sie sonst vom Bmcc nicht erkannt werden.

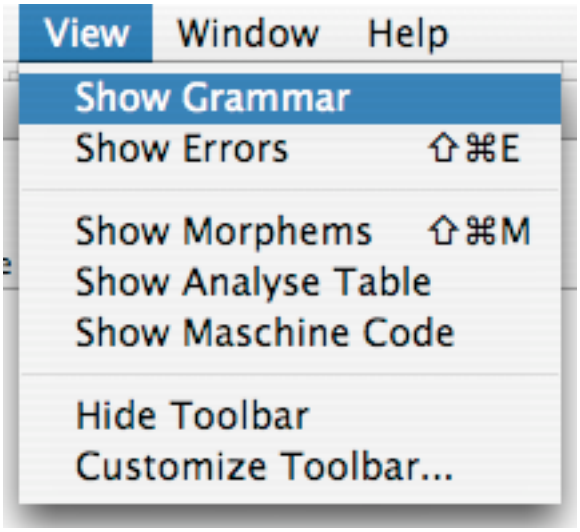

Abbildung 2: Ein Klick auf diesen Menüpunkt zeigt eine geladene Grammatik an

Um eine bereits vorhandene Datei zu öffenen muss man im File Menü auf „Open“klicken. In der sich öffnenden Dateiauswahlbox navigiert man zur gewünschten Datei und lädt diese. Das Quellprogramm wird jetzt im Editor angezeigt und man kann evtl. noch Änderungen vornehmen.

Um die lexikalische Analyse zu starten, muss man im Options Menü auf den Menüpunkt „Lexical Analysis“klicken (Abbildung 3). Vorhandene Fehler im Quellprogramm werden vom Compiler

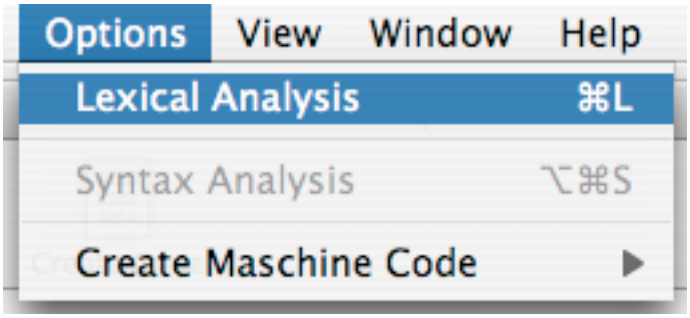

Abbildung 3: Ein Klick auf diesen Menüpunkt startet die lexikalische Analyse

gemeldet, so dass man sie beheben kann und den Vorgang erneut starten kann.

Das Ergebnis der lexikalischen Analyse (die Morpheme) lassen sich in einer Übersicht betrachten. Die erhält man, wenn man im View Menü auf „Show Morphems“klickt. Durch Auswahl eines Morphems in der Übersicht und Klick auf den Button „Show“wird die entsprechende Quelltextstelle markiert und angezeigt.

3.3 Durchführen der syntaktischen Analyse

Nach der lexikalischen Analyse kann man nahtlos mit der syntaktischen Analyse fortfahren. Dazu klickt man im Options Menü auf den Punkt „Syntax Analysis“. Enthält das Programm syntaktische Fehler, so werden diese angezeigt. Man kann diese dann im Editor korrigieren und muss

dann erneut eine lexikalische Analyse durchführen.

Die Analysetabelle kann man sich anzeigen lassen, wenn man im View Menü auf „Show Analyse Table“klickt.

3.4 Codeerzeugung

Nachdem man die syntaktische Analyse abgeschlossen hat kann man zur Codeerzeugung übergehen. Sie findet in drei Schritten statt, die man entweder einzeln oder alle auf einmal ablaufen lassen kann. Das Ergebnis der einzelnen Schritte kann man sich jeweils anschauen, wenn man im View Menü auf den Punkt „Show Machine Code“klickt.

Um die Codeerzeugung zu starten muss man im Options Menü auf den Punkt „Create Machine Code„klicken und dann den gewünschten Punkt auswählen.

Der endgültige Machinencode lässt sich an einem gewünschten Ort speichern, um ihn weiter zu verwenden. Da zu klicken Sie im Codefenster auf den Reiter „Complete“und dann auf den Knopf „Save“. Dann legen Sie den Speicherort und den Namen fest.

4 Beschreibung des XML-Formats

4.1 Die lexikalische Analyse

Die lexikalische Analyse basiert auf einer Menge von regulären Ausdrücken, die in Klassen eingeteilt sind, also z.B. in Schlüsselworte, Operatoren, Identifier und Nummern. Das XML-File beginnt also mit

<?xml version="1.0"encoding="ISO-8859-1"?> <classes>

Und endet mit

</classes>

Dazwischen folgt eine Menge von

<class name="<name>" syntax="<syntax>">

Tags, die wiederum ihre Einträge über

<keyword>keyword< /keyword>

Tags erhalten. <name> steht für den Namen der Klasse, <syntax> für deren Repräsentation in der Grammatikbeschreibung. Wenn für <syntax> „$" angegeben wurde, entspricht die Syntaxbeschreibung für jeden Eintrag seinem Substring des Programms. Dies ist bei allen Schlüsselwörtern und Operatoren wichtig, für Identifier ist <syntax> normalerweise auf z. B. „ident" fixiert.

Während der lexikalischen Analyse werden die Klassen und Einträge in der Reihenfolge durchsucht, wie sie im XML-File stehen und aus dem längsten bzw. ersten Match wird ein Token gebildet.

4.2 Die syntaktische Analyse

Da es sich um eine kontextfreie, tabellengestützte Analyse handelt muss eine LL(1)-Grammatik angegeben werden. Die geschieht durch ein Grammatik-XML-File. Außerdem enthält dieses File die Definition der Semantik, also die Regeln zur Bildung des Programms in der Zielsprache aus dem Syntaxbaum. Dies ist möglich, indem für jede Regel der Grammatik eine Zwillingsregel zur Erzeugung der Semantik angegeben wird. Diese enthält die gleichen Nichtterminale (in beliebiger Reihenfolge, man kann auch manche weglassen) – dies sorgt für das weitere Expandieren der Semantikregeln – und beliebige Terminale, die dann Teil des Outputs werden. Da dadurch nur eine so genannte „kontextfreie Semantik" beschrieben werden kann, ist es nötig, den Semantik-Output in mehreren Pässen (maximal drei) mittels einfacher semantischer Funktionen zum gewünschten Ergebnis zu führen. Genaueres zu diesen Funktionen findet sich nach der XML-Beschreibung. Ein Grammatik-XML-File beginnt mit dem Header, also z. B.:

<?xml version="1.0" encoding="ISO-8859-1" ?>
<grammar name="PL02C#" eps="_eps" start="Program" lexspec="lex_specification.xml">

Hier wird der Name der Grammatik, das Epsilonsymbol, das Start-Nichtterminal und das XML-File für die lexikalische Analyse angegeben. Nun folgt die Liste der Nichtterminale und Terminale:

```
<nterminals>
   <nterminal>Program</nterminal>
   <nterminal>Block</nterminal>
...
</nterminals>
<terminals>
   <terminal>CONST</terminal>
...
</terminals>
```

Dann beginnt mit dem XML-Tag <rules> die Liste der Regeln. Jede Regel beginnt mit

```
<rule name="<NTerm>">
```

und endet mit

```
</rule>
```

wobei <NTerm> das Nichtterminal ist, welches auf der linken Seite der Regel steht. Dann folgt eine Liste von <option>-Tags, die jeweils einen <syntax>- und einen <semantik>-Tag enthalten, also z. B.:

```
<rule name="Vardecl">
   <option>
      <syntax>_eps</syntax>
      <semantik></semantik>
   </option>
   <option>
      <syntax>VAR ident A ;</syntax>
      <semantik>_Define({1},int) _MakeAddress({1}) A</semantik>
   </option>
</rule>
```

Sowohl bei Syntax als auch bei Semantik werden die einzelnen Terminale und Nichtterminale durch Leerzeichen getrennt. Bis auf einige Besonderheiten ist nur durch die Listen <terminals> und <nterminals> ersichtlich, ob ein Wort Terminal oder Nichtterminal ist. Ausgenommen sind dabei semantische Befehle (alle Wörter, die mit ‚_', ‚$' oder ‚@' anfangen), diese werden immer als semantische Terminalzeichen betrachtet. Im obigen Beispiel sind also z. B. ‚VAR', ‚ident', ‚;' und ‚_Define({1},int)' Terminale, während ‚Vardecl' und ‚A' Nichtterminale sind.

4.3 Definition der Semantik

Wie bereits erwähnt, wird die Semantik eines Programms durch die Zwillingsregeln definiert, die für jede Option Pflicht sind. In diesen Semantikblöcken können beliebige Terminale, semantische Funktionen und natürlich die im jeweiligen Syntaxblock verwendeten Nichtterminale benutzt werden.

Zunächst zu den einfachen Erweiterungen. Mit einem ‚_' kann im Output ein Leerzeichen und mit ‚$' ein Zeilenumbruch erzeugt werden. Sollten (z. B. für eine Sprunganweisung) eindeutige Namen gebraucht werden, so kann dies durch ‚@<Name>' geschehen. Alle Vorkommen von ‚@<Name>' in einer Semantikoption werden durch (in dieser Regel gleiche, aber sonst) eindeutige Namen ersetzt, z. B. ‚<Name>154'. Beispiel:

<syntax>IF Condition THEN Statement</syntax>
<semantik>Condition JMNC_ _ResolveMarkToLine(@IfEnd) $
Statement _SetJumpMark(@IfEnd)</semantik>

Hier wird '@IfEnd' so vorverarbeitet, dass die Sprünge richtig ausgeführt werden.

In der folgenden Liste werden schließlich alle im Moment implementierten semantischen Funktionen beschrieben:

Tabelle 1: Implementierte Semantikfunktionen

EnterLevel(name)	legt Semantiklevel "name" an und speichert die Zeilennummer
LeaveLevel()	verlässt das aktuelle Level und betritt dessen Vater
Define(name, type)	legt Variable "name" des Typs "type" an
CheckDefine(name, type1:type2: ...)	prüft die Sichtbarkeit der Variable "name" vom Typ "type1", "type2", "..."
SetJumpMark(name)	verknüpft "name" mit aktueller Zeilennummer
ResolveMarkToLine(name)	liest registrierte Zeilennummer für "name" aus
ResolveLevelToLine(name)	liest registrierte Zeilennummer aus der Semantik-Node "name" aus
MakeAddress(name)	erstellt neue Adresse (beginnend bei 0) für "name"
ResolveAddress(name)	liest registrierte Adresse für "name" aus
ResolveLevelToString(name, sep)	erstellt qualifizierten Namen mit dem Separator "sep" in der Semantik-Node "name"
ResolveCurrentLevelToString(sep)	wie oben, nur in der aktuellen Semantik-Node
MakeAddressForCurrentLevel(Sep)	erstellt Adresse für qualifizierten Namen "sep"
ResolveLevelToAddress(Name,Sep)	liest Adresse für qualifizierten Namen "sep" aus Semantik-Node "name"
ResolveCurrentLevelToAddress(Sep)	wie darüber, nur aktuelle Semantik-Node
All(Commands,Type1:Type2:...)	Dupliziert 'Commands' für alle Variablen-einträge von eines der Typen
AllR(Commands,Type1:Type2:...)	genau wie All(...), nur umgekehrte Reihenfolge
@name	wird durch eindeutigen Namen ersetzt
_	Leerzeichen
$	Newline
{i}	i-tes Symbol der Syntaxoption