Die Lisp-nahe Zwischensprache

Ulrich Hoffmann
Heinz Knutzen

Christian-Albrechts-Universität zu Kiel

Arbeitspaket: Definition der Zwischensprache
Status: Entwurf
Dokumentbezeichner: APPLY/CAU/IV.2/1
Erstellt am: 03.01.92
Korrigiert am: 09.06.92

Synopsis

Dieses Dokument beschreibt die im Apply-System verwendete Lisp-nahe Zwischensprache (LZS). Sie soll als gemeinsame Zwischensprache für verschiedene Compiler-Phasen verwendet werden. Sie unterstützt damit insbesondere die Entwicklung aufeinander abgestimmter Komponenten durch räumlich getrennte Projektpartner.

Diese Arbeit ist im Rahmen des Verbundvorhabens APPLY entstanden, das vom Bundesminister für Forschung und Technologie gefördert wird. Die Projektpartner sind die Christian-Albrechts-Universität Kiel, die Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD), das Fraunhofer Institut für Software- und Systemtechnik (ISST) und VW-GEDAS.

Christian-Albrechts-Universität Kiel
Institut für Informatik und Praktische Mathematik
Forschungsgruppe „Programmiersprachen und Übersetzerkonstruktion"
Preußerstr. 1-9, W–2300 Kiel 1
Fax: 0431/5661-43
Ansprechpartner: Dr. Friedemann Simon
Tel.: 0431/5604-25
E-mail: fs@informatik.uni-kiel.dbp.de

Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD)
Institut für Angewandte Informationstechnik
Forschungsbereich Künstliche Intelligenz
Postfach 1240, W–5205 Sankt Augustin 1
Fax: 02241/14-2618
Ansprechpartner: Prof. Dr. Thomas Christaller
Tel.: 02241/14-2678
E-mail: christal@gmdzi.gmd.de

Fraunhofer Institut für Software- und Systemtechnik (ISST)
Kurstr. 33, O–1086 Berlin
Fax: 0037-2/20372-301
Ansprechpartner: Dr. Horst Friedrich
Tel.: 0037-2/20372-342
E-mail: hf@city.zki-berlin.adw.dbp.de

VW-GEDAS
Pascalstr. 11, W–1000 Berlin 10
Fax: 030/39007999
Ansprechpartner: Dipl.-Ing. Jürgen Beste
Tel.: 030/39007-331
E-mail: beste%sun1@hwsw.gedas.de

Inhaltsverzeichnis

1 Einleitung

Ein Schwerpunkt des APPLY-Projekts ist, durch Kompilation effizient ausführbare Applikationen zu erzeugen. Die Kompilation erfolgt nicht in einem Schritt vom Lisp-Programm in eine ausführbare Form, sondern es wird eine Lisp-nahe Zwischensprache (LZS) definiert, in der wesentliche Analysen und Optimierungen vorgenommen werden. Die Zwischensprache LZS wird in diesem Arbeitspapier beschrieben.

Die Verwendung einer Zwischensprache bietet allgemein den Vorteil, daß verschiedene Compiler-Oberteile und Unterteile geschrieben werden können, die die in der Zwischensprache vorgenommenen Optimierungen gemeinsam nutzen. Im APPLY-Projekt werden Compiler-Oberteile für EuLisp [Padget und Nuyens, 1992] und eine mächtige Teilsprache CommonLisp$_0$ von CommonLisp [Steele Jr., 1990], [Knutzen, 1992] sowie Codegeneratoren für die Sprache C und für verschiedene Maschinensprachen entwickelt. Siehe dazu Abbildung 1.

Da LZS eine gemeinsame Zwischensprache für EuLisp und für CommonLisp ist, müssen sowohl die CommonLisp als auch die EuLisp Konstrukte in der Zwischensprache darstellbar sein. Bei Unterschieden in beiden Sprachen enthält LZS entweder ein allgemeines Konstrukt, das beide Fälle erfaßt, oder zwei unterschiedliche Konstrukte, auf die die jeweiligen Compiler-Oberteile abbilden. Besonderheiten bei der Abbildung der einzelnen Sprachkonstrukte von EuLisp und von CommonLisp in die Zwischensprache sind in Abschnitt 5 beschrieben.

Optimierungen eines Programms und die ihnen zugrundeliegenden Analysen können grundsätzlich in einen machinenabhängigen und einen maschinenunabhängigen Teil aufgeteilt werden. Für Lisp-Programme ist es üblich und sinnvoll, maschinenunabhängige Optimierungen durch Transformationen in einer Lisp-nahen Darstellung durchzuführen. Dazu gehören Optimierungen wie Konstantenausbreitung, Elimination ungenutzter Variablen, Zusammenfassen mehrfacher Berechnungen, Inline Compilation und die Elimination überflüssiger Typprüfungen.

Es gibt in LZS nur relativ wenige verschiedene Sprachkonstrukte, da in der Quellsprache vorkommende Makros bereits expandiert sind und weil Sprachkonstrukte mit ähnlicher Bedeutung durch ein gemeinsames Zwischensprachkonstrukt dargestellt werden. Beispielsweise werden die Special-Forms `let` und `let*` und Lambda-Applikationen durch ein `let*`-Konstrukt dargestellt. Die geringe Anzahl der Sprachkonstrukte bewirkt, daß die Anzahl unterschiedlicher Optimierungsverfahren gering gehalten und diese damit besonders effizient ausgestaltet werden können.

LZS bietet die Möglichkeit, für Übersetzungs- und Optimierungsschritte relevante Informationen den einzelnen Komponenten eines Programms in sog. *Annotationen* zuzuordnen.

Um diese Zusatzinformationen konzeptuell vom eigentlichen Programm zu trennen, setzen sich die Komponenten von LZS aus zwei Teilen zusammen:

1. Der Strukturteil stellt die abstrakte syntaktische Struktur des zu übersetzenden Programms dar und erlaubt das strukturierte Durchlaufen des Programms während der verschiedenen Phasen der Übersetzung.

   Die gesamte Bedeutung des Programms ist durch die im Strukturteil enthaltenen Informationen und durch die Bedeutung der importierten Programmteile bestimmt.

Der Aufbau des Struktur-Teils ist für alle Compiler-Phasen gleich, so daß sie prinzipiell beliebig miteinander kombiniert werden können.

Der Strukturteil wird in Abschnitt 2 festgelegt.

2. Der Annotationsteil enthält übersetzungsrelevante Zusatzinformationen über das Programm, wie etwa Seiteneffekt- oder Typinformationen, die grundsätzlich aus dem Programm in der Zwischensprach-Darstellung durch Analysen gewonnen werden. Jede Compiler-Phase wird andere Informationen über das Quellprogramm benötigen; LZS kann daher im Annotationsteil für jede Phase anders instantiiert werden. Die Konfiguration von LZS für die einzelnen Übersetzerphasen findet also im Annotationsteil statt.

   Mögliche Instantiierungen des Annotationsteils werden in Abschnitt 3 beschrieben.

Die Trennung in einen Strukturteil und einen Annotationsteil führt dazu, daß LZS zu einer *Schemasprache* wird, die für jede Übersetzungsphase eine eigene Ausprägung besitzt. Die Struktur aller Ausprägungen ist durch den einheitlichen Strukturteil festgelegt; die möglichen Annotationen sind durch die jeweilige Instantiierung des Annotationsteils bestimmt.

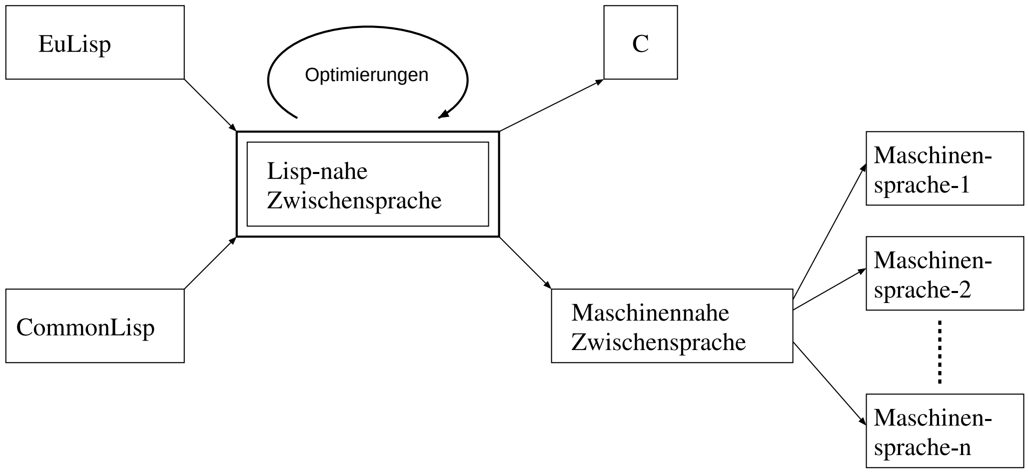

Abbildung 1:

2 Der Strukturteil

Der Strukturteil spiegelt die Struktur des Quellprogramms wider.

Die Programmkonstrukte der Zwischensprache lassen sich in Definitionen und Ausdrücke einteilen. In Definitionen wird die Bedeutung angewandter Vorkommen von *Programmobjekten*, z. B. von Variablen, Funktionen oder Symbolen, festgelegt. Ausdrücke beschreiben die Berechnung von Werten und das Ausführen von Seiteneffekten und greifen dabei auf die Definitionen zurück. Definitionen und Ausdrücke werden in der Zwischensprache durch LZS-Konstrukte dargestellt, die aus benannten Komponenten zusammengesetzt sind.

Gegenüber dem originalen Quellprogramm werden in LZS zahlreiche Informationen explizit dargestellt. Z. B. :

- Die Bindungsrelation

  Sie wird im Original-Programm durch Namensgleichheit und Position im Quelltext ausgedrückt. Im Strukturteil werden definierende und angewandte Vorkommen von Programmobjekten durch Verweise auf dasselbe zugehörige Konstrukt dargestellt. Der Strukturteil stellt so den *Programmgraphen* (Programmbaum und Bindungsrelation) eines Programms dar.

- Die Hintereinanderausführung

  Auf Quellsprachen-Ebene existieren mehrere Stellen (z. B. `cond`-Klauseln, Funktionsrümpfe), an denen implizit eine Hintereinanderausführung erfolgt. In der Zwischensprache werden an diesen Stellen explizit `progn`-Konstrukte eingefügt, um die Hintereinanderausführung explizit zu machen.

- Unterscheidung statischer und dynamischer Variablen

  Die Unterscheidung von dynamisch und statisch gebundenen Variable, die in CommonLisp-Quellprogramm durch `special`-Deklarationen ausgedrückt werden, wird in der Zwischensprache durch eine unterschiedliche Repräsentation dynamisch und statisch gebundener Variablen erreicht.

- Symbolische Daten

  Daten, die auf Quellsprachen-Ebene mit `quote` vor dem Ausführen geschützt dargestellt werden, haben in LZS eine Reräsentation durch eigene Konstrukte.

- Formale Parameterlisten

  Auf Quellsprachen-Ebene werden unterschiedliche Arten von Parametern (z. B. keyword, optional, rest) durch Schlüsselworte bzw. ihre Position in der Parameterliste angegeben. In der Zwischensprache sind die unterschiedlichen Parameter-Arten explizit voneinander getrennt.

Dieses Dokument legt die Bedeutung von Systemfunktionen nicht fest, sondern nur die Art und Weise, mit der Programmobjekte definiert werden und aus diesen wiederum neue Programmobjekte konstruiert werden können. Die Bedeutung eines LZS-Programms ist durch die Bedeutung der Systemfunktionen und die Bedeutung seines Strukturteils bestimmt. Die Bedeutung der Systemfunktionen wird aus den Quellsprachen CommonLisp und EuLisp übernommen.

Systemfunktionen können konzeptuell so aufgefaßt werden, als ob sie in einem ausgezeichneten System-Modul definiert worden sind, das von jedem Modul importiert wird. Die Systemfunktionen werden teilweise in Lisp, teilweise in maschinennäheren Sprachen implementiert.

In der folgenden Beschreibung finden sich Auszüge der LZS-Grammatik, die in Abschnitt 6 vollständig dargestellt ist.

2.1 Module

Die Zwischensprache ermöglicht, motiviert durch das Modulkonzept von EuLisp, die Darstellung von Modulen.

Ein LZS-Programm ist eine Menge von Modulen. Jedes Modul enthält Definitionen für Programmobjekte, die zur Verwendung in anderen Modulen exportiert werden können. Anderen Modulen werden die Programmobjekte über eine *Exportspezifikation* bekannt gegeben, gemäß derer sie die Programmobjekte importieren können. Die Exportspezifikation dient so zum Austausch von Informationen zwischen Modulen. Das Modul ist im APPLY-System die kleinste Übersetzungseinheit.

Programmobjekte der Zwischensprache sind in zwei Arten eingeteilt:

1. *Definierte Programmobjekte* werden im aktuellen Modul definiert. Wenn sie nur lokal im Modul benutzt und nicht exportiert werden, so sind schon während der Übersetzung des aktuellen Moduls alle angewandten Vorkommen der Programmobjekte bekannt.

2. Die Definitionen *importierter Programmobjekte* befinden sich in anderen Modulen, die vom aktuellen Modul importiert werden. Aus der Exportspezifikation dieser Module sind Übersetzungs-Informationen (z. B. Typsignatur, Seiteneffektfreiheit) der Programmobjekte bekannt.

Definierte und importierte Programmobjekte können aus Modulen exportiert werden. Bei definierten Programmobjekten wird aus ihren Definitionen die Exportspezifikation des Moduls generiert; für importierte Programmobjekte, wird die Exportspezifikation des ursprünglich exportierenden Moduls übernommen. In den LZS-Konstrukten für definierte und importierte Programmobjekte findet sich die Komponente `exported`. Sie gibt an, ob das entsprechende Programmobjekt aus dem aktuellen Modul exportiert werden soll.

```
module ::= [ function-list, class-list, named-constant-list, variable-list, symbol-list,
             init-function || ... ].
```

Ein Modul besteht aus jeweils einer Liste der im Modul auftretetenden Funktionsdefinitionen, Klassen-, Konstanten-, Variablen- und Symboldefinitionen.

Die auf oberster Ebene in Quellprogrammen auftretenden Ausdrücke, die keine Definitionen oder Deklarationen sind, die sog. *Top-Level-Forms*, bilden den Rumpf einer ausgezeichneten Hauptfunktion `init-function`, die die Initialisierung des Moduls übernimmt. Nach dem Start eines Programms werden die Hauptfunktionen aller Module ausgeführt, aus denen das Programm besteht.

Im folgenden werden die einzelnen Programmobjekte und ihre Darstellung in der LZS erläutert und motiviert.

2.2 Literale

Im Quellprogramm auftretende literale Konstanten werden in LZS in mehrere unterschiedliche Arten eingeteilt. Sie werden im folgenden kurz als *Literale* bezeichnet.

1. Einfache Literale besitzen keine Feinstruktur. Einfache Literale sind Zahlen (integer, float, ...), Buchstaben (character) und die leere Liste.
2. Strukturierte Literale besitzen eine Feinstruktur. Strukturierte Literale sind Felder (array, string), Strukturen (literal-instance) und Paare (cons).
3. Symbole, die im Quellprogramm durch `(quote` symbol) dargestellt sind, besitzen eine Feinstruktur. Im Gegensatz zu strukturierten Literalen bezeichnen mehrere Vorkommen eines Symbols im Quellprogramm aber dasselbe Symbol und werden in LZS durch Verweise auf dieselbe Struktur dargestellt.

literal ::= symbol | simple-literal | annotated-structured-literal.

2.3 Variablen

Variablen repräsentieren Speicherplätze, in denen zur Laufzeit eines Programms Programmobjekte abgelegt werden können.

In LZS ist die Bindungsrelation zwischen angewandten und definierenden Variablenvorkommen explizit gemacht. Alle angewandten Vorkommen von Variablen verweisen auf das zugehörige definierende Vorkommen. Unterschiedliche Variablen, die im Quellprogramm gleiche Namen besitzen, werden in LZS durch unterschiedliche Objekte dargestellt. Dadurch sind LZS-Programme ausgezeichnet.

In LZS wird zwischen Variablen mit statischer und dynamischer Bindung unterschieden. Statische Variablen entsprechen den globalen bzw. lokalen statischen Variablen von EuLisp[Padget und Nuyens, 1992], und den lokal statischen Variablen von CommonLisp. Für die global statischen Variablen aus EuLisp gibt es in CommonLisp keine Entsprechung. Dynamische Variablen können zur Laufzeit des übersetzten Programms über Symbole angesprochen werden. In der Zwischensprache wird das Symbol in der Komponente `symbol` nachgehalten.

local-static ::= [|| ...].
global-static ::= [exported || ...].
dynamic ::= [symbol || ...].

Für dynamische Variablen findet sich die `exported`-Kennzeichnung an dem korrespondierenden Symbol, da dynamische Variablen und die zugehörigen Symbole nur gemeinsam exportiert werden können.

2.3.1 Einführen neuer Bindungen

Mit Hilfe des Konstrukts `let*` werden in LZS neue Variablenbindungen eingeführt. Da typischerweise mehrere Variablenbindungen auf einmal erzeugt werden, enthält das `let*`-Konstrukt eine Liste der zu bindenden Variablen (`variable-list`), statt nur einer einzigen Variablen, und eine Liste ihrer Initialisierungswerte (`initial-value-list`). Diese beiden Listen sind im Gegensatz zur Quellsprache getrennt, da sie im Übersetzungsprozeß in der Regel unabhängig voneinander bearbeitet werden.

let* ::= [variable-list, initial-value-list, body || ...].

Das LZS-Konstrukt `let*` deckt beide in den Quellsprachen vorkommenden Special-Forms `let` und `let*` ab, da Programme der Zwischensprache ausgezeichnet sind, und so jedes `let` als `let*` dargestellt werden kann. Werden dynamische Variablen in `let`-Konstruktionen gebunden, so kann es ggf. notwendig sein, Hilfsvariablen einzuführen, um Bindungsverfälschungen zu vermeiden.

Als Kontextbedingung muß sichergestellt sein, daß

1. die Variablen, die in `variable-list` aufgeführten werden, keine global-statischen Variablen sind und
2. angewandte Vorkommen der lokal statischen Variablen nur innerhalb ihres Gültigkeitsbereiches auftreten.

2.3.2 Multiple Werte

Um die in CommonLisp vorkommende Special-Form `multiple-value-call` und darauf aufbauende Makros wie z. B. `multiple-value-setq` auszudrücken, wird das LZS-Konstrukt `mv-lambda` verwendet.

mv-lambda ::= [parameters, mv-argument, body || ...].

Ähnlich wie das `let*`-Konstrukt, bindet `mv-lambda` Werte an Variablen. Dazu werden die multiplen Werte verwendet, die bei der Auswertung des in der Komponente `mv-argument` beschriebenen Ausdrucks anfallenden. Da die Anzahl der multiplen Werte möglicherweise erst zur Laufzeit bekannt wird, ist es notwendig, eine allgemeine formale Parameterliste anstatt einer Liste von Variablen zu verwenden, um überschüssige Werte an optionale oder Rest-Parameter binden zu können.

2.3.3 Wertzuweisungen

Zuweisungen von Werten an Variablen werden in LZS durch das Konstrukt `setq` beschrieben.

setq ::= [location, form || ...].

Die Komponente `form` ist ein Zwischensprachkonstrukt, das den Ausdruck beschreibt, dessen Wert an das durch `location` bestimmte Programmobjekt zugewiesen werden soll. `location` ist dabei entweder eine Variable oder eine benannte Konstante (vgl. 2.4).

Als Kontextbedingung muß sichergestellt sein, daß Zuweisungen an benannte Konstanten nur einmal in der Initialisierungs-Funktion des Moduls vorgenommen werden.

2.4 Benannte Konstanten

Die Konstanten der Zwischensprache entsprechen Konstanten in EuLisp (`defconstant`). Sie sind statischen Variablen mit unveränderbarer Bindung an einen Wert.

defined-named-constant ::= [value, exported || ...].

2.5 Funktionen

Die Quellsprachen CommonLisp und EuLisp erlauben die Definition von Funktionen und generischen Funktionen. Nicht generische Funktionen werden im folgenden als *einfache* Funktionen bezeichnet, um sie deutlich von generischen Funktionen zu unterscheiden.

Durch das Auszeichnen der LZS-Programme, können Unterscheidungsprobleme zwischen dem funktionalen und dem eigentlichen Wert von Variablen, wie sie in CommonLisp vorkommen, nicht auftreten. Dies erlaubt den Umgang mit funktionalen Programmobjekten ohne die Verwendung von `funktion` bzw. `funcall`.

In der Zwischensprache werden definierende und angewandte Vorkommen von Funktionen durch dasselbe Konstrukt dargestellt. Definierende Vorkommen treten nur in der Komponente `function-list` der Konstrukte `module` und `labels` auf und können so von angewandten Vorkommen unterschieden werden.

2.5.1 Einfache Funktionen

Genau wie andere Programmobjekte, werden auch einfache Funktionen entweder unmittelbar im Modul selbst definiert oder aus anderen Modulen importiert.

imported-simple-function ::= [parameters, body, exported || ...].

defined-simple-function ::= local-simple-function | global-simple-function.

local-simple-function ::= [parameters, body || ...].
global-simple-function ::= [parameters, body, exported || ...].

Einfache Funktionen werden durch ein LZS-Konstrukt beschrieben, das als Komponenten eine Beschreibung der formalen Parameterliste (`parameters`, vgl. 2.5.4) und eine Beschreibung des Funktionsrumpfes (`body`) enthält.

Die Umgebung, in der der Funktionsrumpf zur Laufzeit ausgewertet wird, ist durch die Bindungsrelation gegeben, die durch die Verweise der angewandten Programmobjekt-Vorkommen auf ihre definierenden Vorkommen dargestellt ist.

Im Modul definierte Funktionen werden in lokal und global definierte Funktionen unterschieden. Ihr Strukturteile unterscheiden sich nur durch die Komponente `exported`, da lokal definierte Funktionen nicht exportiert werden können. Während der Übersetzung werden lokal und global definierte Funktionen jedoch unterschiedlich behandelt. Dies drückt sich durch erweiterte Annotationsteile für lokal definierte Funktionen aus (vgl. Abschnitt 3).

Auch von importierten Funktionen kann der Funktionsrumpf aus der Exportschnittstelle bekannt sein, etwa wenn er genügend kurz ist. Dies eröffnet die Möglichkeit auch importierte Funktionen *inline* zu kompilieren. Ist der Rumpf unbekannt so enthält die Komponente `body` den ausgezeichneten Wert `not-specified`.

Die Bindungen zwischen Funktionsidentifikatoren und Funktionen sind konstant. Im Gegensatz zu Variablen, bei denen die Bindung an jedem Vorkommen unterschiedlich sein kann, ist es daher bei Funktionen nicht notwendig, seperate Informationen über den Wert für jedes einzelne Vorkommen der Funktionen nachzuhalten.

2.5.2 Lokal definierte Funktionen

Lokale Funktionsdefinitionen erfolgen durch das Konstrukt `labels`.

labels ::= [function-list, body || ...].

Die Komponente `function-list` enthält eine Liste der lokal definierten Funktionen, deren Gültigkeitsbereich sich über die Komponenten des `labels`-Konstrukts erstreckt. Als Kontextbedingung muß sichergestellt sein, daß angewandte Vorkommen der lokal definierten Funktionen nur im Gültigkeitsbereich des `labels`-Konstrukts auftreten.

Unbenannte Funktionen (Lambda-Ausdrücke) werden vom Compiler-Oberteil in äquivalente `labels`-Konstrukte transformiert.

Da Programme der Zwischensprache ausgezeichnet sind, deckt das `labels`-Konstrukt beide in den Quellsprachen vorkommenden Special-Forms `labels` und `flet` ab. Jedes `flet` läßt sich durch `labels` darstellen.

2.5.3 Generische Funktionen und Methoden

Die Definition generischer Funktionen und ihrer Methoden ist Bestandteil der Objekt-Systeme von CommonLisp und EuLisp. Auf Ihre Darstellung in der Zwischensprache wird gesondert in Abschnitt 4 eingegangen.

2.5.4 Parameterlisten

Die Beschreibung der formalen Parameterlisten ist an Ausdrucksmöglichkeiten von CommonLisp angelehnt. Die einzelnen Komponenten der LZS-Struktur beschreiben die unterschiedlichen Parameter-Arten (einfache Parameter, optionale Parameter, Rest-Parameter, Keyword-Parameter) vgl. [Steele Jr., 1990] S.75ff.

parameters ::= [variable-list, optional-list, rest, keyword-list, allow-other-keys || ...].

2.5.5 Funktions-Anwendungen

Die Anwendung von Funktionen (funktionalen Programmobjekten) auf Argumente wird in LZS durch das Konstrukt `application` beschrieben.

application ::= [form, argument-list || ...].

Die Komponente `form` beschreibt einen Ausdruck, dessen Auswertung ein funktionales Objekt liefert. (Funktionale Programmobjekte sind Daten erster Ordnung). Da LZS-Programme ausgezeichnet sind, kann es keine Verwechselungen zwischen dem funktionalen Wert und dem eigentlichen Wert von Variablen geben, die auf Funktionsposition der Applikation stehen.

Die Argumentliste ist eine Liste von Audrücken, deren Auswertung die an das funktionale Objekt zu übergebenden Werte liefert.

2.6 Klassen

Sowohl CommonLisp als auch EuLisp besitzen objekt-orientierte Erweiterungen (Objekt-Systeme) CLOS bzw. TELOS, deren Entwicklung noch nicht abgeschlossen ist. Ihre Darstellung in der Zwischensprache ist gesondert in Abschnitt 4 beschrieben.

2.7 Bedingte Ausdrücke

Bedingte Ausdrücke werden in der Zwischensprache durch das `if` und das `switch`-Konstrukt dargestellt.

2.7.1 Bedingte Ausdrücke mit zwei Zweigen

if ::= [predicate, then, else ‖ ...].

Das `if`-Konstrukt faßt drei Ausdrücke zusammen.

1. Der Prädikatausdruck `predicate` beschreibt die Bedingung des `if`-Konstrukts.
2. Der Ausdruck `then` beschreibt denjenigen Ausdruck, der ausgewertet wird, wenn die Bedingung wahr ist.
3. Der Ausdruck `else` beschreibt denjenigen Ausdruck, der ausgewertet wird, wenn die Bedingung falsch ist.

2.7.2 Fallunterscheidungen über einfache Werte

Fallunterscheidungen, bei denen mit Hilfe eines Schlüssels einer von mehreren Fällen ausgewählt wird, lassen sich als Ketten von bedingten Ausdrücken mit zwei Zweigen ausdrücken.

Fallunterscheidungen, bei denen der Schlüssel ein Symbol oder ein einfaches Literal ist, lassen sich effizient übersetzen, etwa mittels Sprungtabellen oder Hash-Verfahren. Die Zwischensprache enthält daher das Konstrukt `switch`, um solche Fallunterscheidungen ausdrücken zu können.

```
switch ::= [ form, case-list, otherwise || ... ].
      case-list ::= ( case* ).
      case ::= [ case-label, form || ... ].
      case-label ::= symbol | simple-literal.
      otherwise ::= form.
```

`form` beschreibt den Ausdruck, der den Schlüssel bestimmt. `case-list` ist eine Liste von Fällen, die durch einfache Literale oder durch Symbole markiert sind. Während der Laufzeit wird der Rumpfausdruck desjenigen Falls ausgewertet, dessen Markierung mit dem Schlüssel übereinstimmt. `otherwise` beschreibt den Ausdruck, der ausgewertet wird, falls keiner der anderen Fälle ausgewählt werden konnte. Als Kontextbedingung muß sichergestellt sein, daß die Markierungen der Fälle paarweise ungleich sind und den gleichen Typ besitzen.

Vom Compiler-Oberteil werden geeignete Quellsprachenausdrücke erkannt und in `switch`-Konstrukte der Zwischensprache überführt.

2.8 Sequentielle Auswertung

Auf Quellsprachenebene erlaubt Lisp häufig das Auftreten einer Folge von Ausdrücken, die nacheinander abgearbeitet werden (`cond`-Klauseln, Funktionsrümpfe, u. a.). In der Zwischensprache sind diese impliziten Vorkommen der sequentiellen Auswertung durch explizite `progn`-Konstrukte ausgedrückt. Dies ermöglicht eine einheitliche Handhabung der sequentiellen Auswertung.

```
progn ::= [ form-list || ... ].
```

Die Komponente `form-list` enthält eine Liste der sequentiell auszuführenden Ausdrücke.

2.9 Continuations

Die vorzeitige Beendung der Auswertung eines Ausdrucks wird in der Zwischensprache durch sog. *downward-continuations*, entsprechend den EuLisp-Continuations, ausgedrückt ([Padget und Nuyens, 1992]). Das Konstrukt `let/cc` definiert eine Continuation, deren Gültigkeitsbereich der Rumpf (`body`) des `let/cc` Konstrukts ist.

```
let/cc ::= [ continuation, body || ... ].
```

Eine Continuation ist ein funktionales Objekt mit einem Parameter. Wird während der Ausführung des Rumpfs die Continuation angewendet, so wird die Ausführung des `let/cc`-Ausdrucks beendet. Sein Wert ist dann das Argument, das der Continuation übergeben wurde.

In EuLisp wird durch das `let/cc`-Konstrukt die Continuation unveränderbar an eine Variable gebunden. Sie entspricht daher einer lokalen Konstanten. Vorkommen dieser Variablen im Rumpf repräsentieren die Continuation. In der Zwischensprache sind die Vorkommen der Continuation explizit gemacht, eine Variable wird daher nicht benötigt.

2.10 Marken und Sprünge

Die in CommonLisp auftretenden Sprachkonstrukte `tagbody` und `go` werden in der Zwischensprache durch äquivalente Konstrukte ausgedrückt.

```
tagbody ::= [ first-form, tagged-form-list || ... ].
    tagged-form-list ::= ( tagged-form* ).
    first-form ::= form.
    tagged-form ::= [ form, tagbody || ... ].

go ::= tagged-form.
```

Das `tagbody`-Konstrukt enthält eine Liste markierter Ausdrücke (`tagged-form`), die dort definierend vorkommen.

Innerhalb der Ausdrücke, die das `tagbody`-Konstrukt umfaßt, kann `tagged-form` angewandt vorkommen und markiert dann einen Sprung (`go`) zum entsprechenden definierenden Vorkommen.

Die Komponente `first-form` beschreibt einen evtl. vorhandenen nicht markierten Ausdruck zu Beginn des `tagbody`-Konstrukts. Sie ist mit NIL besetzt, falls ein solcher Ausdruck nicht existiert.

Als Kontextbedingung muß sichergestellt sein, daß angewandte Vorkommen von `tagged-form` nur im Gültigkeitsbereich des `tagbody`-Konstrukts auftreten.

Das Resultat des `tagbody`-Konstrukts ist im Gegensatz zu CommonLisp nicht NIL, sondern das Resultat der letzten `tagged-form`.

3 Der Annotationsteil

In diesem Abschnitt wird dargestellt, welchen Charakter LZS-Annotationen besitzen. Die tatsächlichen Annotationen der einzelnen Compiler-Phasen sind im wesentlichen durch technische Details bestimmt. Sie werden näher in den APPLY-Arbeitspapieren über die entsprechenden Compiler-Phasen dargestellt.

Der Annotationsteil enthält Informationen, die für die einzelnen Phasen der Übersetzung relevant sind.

Aus Arbeiten im Rahmen des CLICC-Systems [Knutzen, 1991; Burkart, 1991] sind Annotationen für einfache Analysen und Optimierungen sowie für die Code-Generierung bekannt.

Annotationen werden durch Analysen (z. B. Typ- oder Seiteneffektanalysen) gewonnen und von Optimierungen verwendet. Während der Analysen dienen sie dazu, Informationen zwischen angewandten und definierenden Vorkommen der Programmobjekte auszutauschen. Zusätzlich können Annotationen auch zur besseren Diagnose Rückverweise auf den Original-Quelltext beeinhalten. Der Code-Generator annotiert Programmobjekte mit symbolischen Namen, Adressen und Offsets der entsprechenden Zielsprachen-Objekte.

- angewandte Variablenvorkommen

  Mit Hilfe des LZS-Konstrukts `variable-reference` kann jedes angewandte Vorkommen einer Variablen getrennt annotiert werden. Während einer Analyse des Datenflusses können hier beispielsweise Verweise von angewandten Variablenvorkommen auf korrrespondierende Teile im Datenflußgraphen eingetragen werden.

- Hintereinanderausführung

  Bei der Annotation des LZS-Konstrukts, das die sequentielle Auswertung von Ausdrücken darstellt (`progn`) ist es wünschenswert, Informationen auszudrücken, die konzeptuell zwischen zwei Ausdrücken der `form-list` gelten. Solche Informationen können in einer zu `form-list` strukturgleichen Liste in den Annotationsteil des zugehörigen `progn`-Konstrukts eingetragen werden.

4 Objekt-System

Sowohl CommonLisp als auch EuLisp beinhalten ein Objekt-System (CLOS bzw. TELOS). Mit der Beschreibung sog. *Metaobjekt-Protokolle* (kurz: *MOP*) [Kiczales *et al.*, 1991] besitzen sie eine Möglichkeit zur Konfiguration, die qualitativ über die Erweiterungsmöglichkeiten anderer objekt-orientierter Sprachen hinausgeht.

Das Metaobjekt-Protokoll von CLOS bzw. TELOS besitzt ein Standard-Protokoll, das objekt-orientierte Programmierung erlaubt, wie sie aus klassischen objekt-orientierten Programiersprachen bekannt ist. Die Darstellung der Programmobjekte dieses Standardfalls in LZS unter Verzicht auf die Erweiterungsmöglichkeiten des MOP ist geradlinig und bereitet keine weiteren Schwierigkeiten. Sollen aber auch Erweiterungen durch das Metaobjekt-Protokoll erlaubt sein, so muß die Zwischensprache Beschreibungen dieser Erweiterungen aufnehmen können. Es ist zur Zeit Gegenstand der Untersuchungen, welche Strukturen die Zwischensprache im objekt-orientierten Teil enthalten muß, um Metaobjekt-Protokoll Anwendungen darstellen zu können.

Die Metaobjekt-Protokolle von CLOS und TELOS befinden sich gegenwärtig in der Standardisierungs- bzw. Entwicklungsphase. Die Zwischensprache soll in ihrem objekt-orientierten Teil so beschaffen sein, daß sich CLOS bzw. TELOS-Anwendungen im Rahmen der Komplettkompilation mit möglichst wenigen Einschränkungen darstellen lassen.

5 Beziehung zwischen CommonLisp, EuLisp und der Zwischensprache

In diesem Abschnitt wird erläutert, welche Besonderheiten bei der Umsetzung der Sprachelemente aus CommonLisp [Steele Jr., 1990] und EuLisp [Padget und Nuyens, 1992] in die Zwischensprache auftreten.

5.1 Module

Die LZS-Module lehnen sich an die Struktur der EuLisp-Module an.

In CommonLisp existiert ein Paket-Konzept, das das Zusammenfassen von Programmobjekten zu Paketen ermöglicht. Hierbei ist es erlaubt, zu beliebiger Zeit Definitionen in anderen Paketen vorzunehemen und auch uneingeschränkt beliebige Programmobjekte aus anderen Paketen zu referenzieren. Um eine getrennte Übersetzung von Paketen zu erlauben, bei der die geplanten Optimierungen Wirkung zeigen, muß das CommonLisp-Paket-Konzept eingeschränkt werden. Die in CommonLisp$_0$ [Knutzen, 1992] vorgenommenen Einschränkungen des Paket-Konzeptes erzwingen eine strukturierte, modulartige Verwendung von Paketen, die vom Compiler-Oberteil in `module`-Konstrukte der Zwischensprache überführt werden.

5.2 Wertzuweisungen

Die in CommonLisp mögliche Mehrfachzuweisung (`setq` v1 e1 v2 e2 ...) ist in der Zwischensprache durch die Hintereinanderausführung einfacher Zuweisungen dargestellt.

5.3 Formale Parameterlisten

Die formalen Parameterlisten in CommonLisp und EuLisp unterscheiden sich syntaktisch voneinander. EuLisp kennt keine Keyword-Parameter oder optionalen Parameter; Rest-Parameter werden mit Hilfe von Listen der Form (p1 p2 p3 .rest) dargestellt. Die Struktur der formalen Parameterlisten in der Zwischensprache ist durch CommonLisp geprägt, und erlaubt damit sowohl die Darstellung formaler Parameterlisten von EuLisp als auch der von CommonLisp.

5.4 Symbole

In CommonLisp sind Symbole dem Paket zugeordnet, in dem sie definiert wurden. Um diesen Sachverhalt in der Zwischensprache zu modellieren, existiert die Komponente `package` in der Zwischensprachstruktur `symbol`, die das dem Symbol zugeordente Paket angibt. Da Symbole auch exportiert werden können, enthält `symbol` zusätzlich auch die Komponente `exported`. In EuLisp befinden sich alle Symbole in einem Namensraum und haben damit keinen Bezug auf ein definierendes Modul. Um EuLisp-Symbole in der Zwischensprache darzustellen, werden die Komponenten `package` und `exported` von `symbol` nicht benötigt.

Anders als in EuLisp, das benannte Konstanten als statische Variablen mit unveränderlicher Bindung auffaßt, werden in CommonLisp benannte Konstanten durch Symbole mit unveränderbarem Wert implementiert. In der Zwischensprache gibt es daher die Komponente `constant-value` zur Kennzeichnung von Symbolen, die als benannte Konstante in CommonLisp-Quellprogrammen definiert sind.

5.5 Makros

Die auf Quellsprachen-Ebene existierenden Definitionen von Makros und ihre Anwendungen werden durch das Compiler-Oberteil erkannt und vollständig expandiert. LZS besitzt keine Makros.

5.6 Continuations

In EuLisp definiert man mit der `let/cc`-Special-Form sog. *downward-continuations*, die angewendet werden, um die Auswertung von Ausdrücken vorzeitig zu beenden. Diese Verwendung wird in CommonLisp durch die Special-Forms `catch` und `throw` bzw. `block` und `return-from` erreicht. Continuations können als Datenobjekte auch Variablen zugewiesen werden, und so im Programm herumgereicht werden. In die Zwischensprache wurde das Konzept der downward-continuations aufgenommen, da sich sowohl `catch` und `throw` als auch `block` und `return-from` durch Continuations implementieren lassen.

5.7 Typdeklarationen

In CommonLisp existiert die Möglichkeit, für Ausdrücke einen Typ zuzusichern (z. B. durch die Special-Forms `the` oder `declare`). In der Zwischensprache werden diese Deklarationen durch explizite Typüberprüfungen der Form `(unless` (Typprädikat Ausdruck) (Fehlerfunktion)`)` ausgedrückt, die einen Fehler auslösen, falls der Ausdruck nicht den geforderten Typ besitzt.

6 Grammatik

In diesem Abschnitt wird die Grammatik von LZS angegeben. Terminale Symbole sind dabei **fett** gedruckt. Ein hochgestellter Stern hinter einem terminalen oder nicht-terminalen Symbol kennzeichnet, daß das Symbol an dieser Stelle beliebig häufig stehen darf.

Lisp Datenstrukturen werden in ihrer klassischen Form dargestellt:

- Listen werden in **(** und **)** eingefaßt.
- Paare werden durch **(** expr **.** expr **)** dargestellt.
- Arrays werden durch **#(** bzw. **#n(** und **)** eingefaßt

Das spezielle Element **()** stellt das ausgezeichnete Datum *empty-list* dar und **T** den Wahrheitswert `true`. [und] bezeichnen Lisp-Klassen, wobei || **...** angibt, daß die jeweilige Klasse weitere Komponenten im Annotationsteil besitzen kann. Produktionen besitzen die Form `name ::= definition.`.

module ::= [function-list, class-list, named-constant-list, variable-list, symbol-list,
init-function || ...].
  function-list ::= (function*).
  class-list ::= (class*).
  named-constant-list ::= (named-constant*).
  variable-list ::= (variable*).
  symbol-list ::= (symbol*).
  init-function ::= defined-global-function.

form ::= self-evaluating | named-constant | variable-reference | application | special-form.

self-evaluating ::= function | class | continuation | literal.

body ::= form.

function ::= simple-function | generic-function.

simple-function ::= imported-simple-function | defined-simple-function.
  imported-simple-function ::= [parameters, body, exported || ...].
    exported ::= T | ().
  defined-simple-function ::= local-simple-function | global-simple-function.
    local-simple-function ::= [parameters, body || ...].
    global-simple-function ::= [parameters, body, exported || ...].

generic-function ::= imported-generic-function | defined-generic-function.
  imported-generic-function ::= NOT YET DEFINED.
  defined-generic-function ::= local-generic-function | global-generic-function.
    local-generic-function ::= NOT YET DEFINED.
    global-generic-function ::= NOT YET DEFINED.

parameters ::= [variable-list, optional-list, rest, keyword-list, allow-other-keys || ...].
    optional-list ::= (optional*).
        optional ::= [variable, form, supplied || ...].
        supplied ::= variable | **()**.
    rest ::= variable | **()**.
    keyword-list ::= (keyword*).
        keyword ::= [variable, form, supplied, symbol || ...].
    allow-other-keys ::= **T** | **()**.

class ::= defined-class | imported-class.
    defined-class ::= NOT YET DEFINED.
    imported-class ::= NOT YET DEFINED.

continuation ::= [|| ...].

literal ::= symbol | simple-literal | annotated-structured-literal.

symbol ::= imported-symbol | defined-symbol.
    imported-symbol ::= [constant-value, exported || ...]..
    defined-symbol ::= [name, package, constant-value, exported || ...].
        constant-value ::= self-evaluating | **unknown** | **no-constant**.
        package ::= "A LISP STRING".
        name ::= "A LISP STRING".

simple-literal ::= **()** | character | integer | float | ratio | complex.

annotated-structured-literal ::= [structured-literal || ...].
    structured-literal ::= array | literal-instance | cons | string.
        array ::= #(literal'*) | #nA(... ((literal'*))* ...).
        literal-instance ::= [class, value-list].
            value-list ::= (literal'*).
        cons ::= (literal' **.** literal').
        string ::= "A LISP STRING".

        literal' ::= symbol | simple-literal | structured-literal.

named-constant ::= defined-named-constant | imported-named-constant.
    defined-named-constant ::= [value, exported || ...].
    imported-named-constant ::= [value, exported || ...].
        value ::= self-evaluating | **unknown**.

variable-reference ::= [variable || ...].

variable ::= static | dynamic.
    static ::= imported-static | defined-static.

imported-static ::= [exported || ...].
defined-static ::= local-static | global-static.
local-static ::= [|| ...].
global-static ::= [exported || ...].
dynamic ::= [symbol || ...].

application ::= [form, argument-list || ...].
argument-list ::= (form*).

special-form ::= setq | let∗ | mv-lambda | labels | let/cc |
if | switch | progn | tagbody | go |.

setq ::= [location, form || ...].
location ::= variable-reference | named-constant.

let∗ ::= [variable-list, initial-value-list, body || ...].
initial-value-list ::= (form*).

mv-lambda ::= [parameters, mv-argument, body || ...].
mv-argument ::= form.

labels ::= [function-list, body || ...].
function-list ::= (function*).

let/cc ::= [continuation, body || ...].

if ::= [predicate, then, else || ...].
predicate ::= form.
then ::= form.
else ::= form.

switch ::= [form, case-list, otherwise || ...].
case-list ::= (case*).
case ::= [case-label, form || ...].
case-label ::= symbol | simple-literal.
otherwise ::= form.

progn ::= [form-list || ...].
form-list ::= (form*).

tagbody ::= [first-form, tagged-form-list || ...].
tagged-form-list ::= (tagged-form*).
first-form ::= form.
tagged-form ::= [form, tagbody || ...].

go ::= tagged-form.

7 Beispiel

Das folgenden Beispiel soll darstellen, wie ein einfaches Lisp-Programm in LZS dargestellt wird.

```
(defun fac (n)
   (cond
      ((= n 0) 1)
      (t (* n (fac (- n 1))))))

(fac 5)
```

Um eine als Text darstellbare Form der LZS-Repräsentation dieses Programms zu erhalten, müssen wieder Namen eingeführt werden. Sie drücken aus, daß im Strukturbaum des Programms an *verschiedenen* Stellen, *dieselben* Teilbäume vorkommen. Zwischensprachkonstrukte werden ausdrücklich in der Form $[_{\text{Konstruktname}} \ldots]$ mit ihrem Namen markiert.

$[_{\text{module}}$ (FAC MAIN), (), (), (), (), MAIN || ...]

FAC: $[_{\text{global-simple-function}}$ $[_{\text{parameters}}$ (N), (), (), (), () || ...]
($[_{\text{if}}$ $[_{\text{application}}$ =, ($[_{\text{variable-reference}}$ N || ...] 0) || ...]
1
$[_{\text{application}}$ *,
($[_{\text{variable-reference}}$ N || ...]
$[_{\text{application}}$ FAC, ($[_{\text{application}}$ −,
($[_{\text{variable-reference}}$ N || ...] 1) || ...])
|| ...])
|| ...]
|| ...])
|| ...]

MAIN: $[_{\text{global-simple-function}}$ $[_{\text{parameters}}$ (), (), (), (), () || ...]
($[_{\text{application}}$ FAC, (5) || ...])
|| ...]

N: $[_{\text{local-static}}$ || ...]

=: $[_{\text{imported-simple-function}}$ (), (), () || ...]

*: $[_{\text{imported-simple-function}}$ (), (), () || ...]

−: $[_{\text{imported-simple-function}}$ (), (), () || ...]

Index

Literatur

[Burkart, 1991] O. Burkart. Das Frontend eines Compilers von Common Lisp nach C. Diplomarbeit, Institut für Informatik und Prakt. Math. der Christian-Albrechts-Universität, Kiel, 1991.

[Kiczales *et al.*, 1991] G. Kiczales, J. des Rivieres und D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[Knutzen, 1991] H. Knutzen. Codegenerierung und Laufzeitsystem eines Compilers von Common Lisp nach C. Diplomarbeit, Institut für Informatik und Prakt. Math. der Christian-Albrechts-Universität, Kiel, 1991.

[Knutzen, 1992] H. Knutzen. CommonLisp$_0$-Sprachdefinition. APPLY-Arbeitspapier APPLY/CAU/II.2/1, Christian-Albrechts-Universität, Kiel, April 1992.

[Padget und Nuyens, 1992] J. Padget und G. Nuyens. The EuLisp Definition, Version 0.8. April 1992.

[Steele Jr., 1990] G. L. Steele Jr. *Common Lisp - The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.