XPASCAL
eine Erweiterung der Sprache Pascal mit exakter Arithmetik

Klaus Lagally

Institut für Informatik
der Universität Stuttgart

12. Oktober 1989

Zusammenfassung

XPASCAL ist ein experimentelles Programmsystem zur Unterstützung exakter Berechnungen in arithmetischen Zahlkörpern, das derzeit in der Abteilung Betriebssoftware am Institut für Informatik der Universität Stuttgart entwickelt wird. Bisher haben daran neben dem Verfasser die Studenten G. Neusetzer, U. Schoppe, G. Wahl, Th. Schöbel und S. Robitschko mitgearbeitet. Der vorliegende Bericht soll die derzeitigen Zielvorstellungen und den Stand der Realisierung aufzeigen und zu Kommentaren, Änderungswünschen und Verbesserungsvorschlägen einladen.

Inhaltsverzeichnis

Kapitel 1

Einführung

XPASCAL ist ein experimentelles Programmsystem, das die bequeme Formulierung und Ausführung von Berechnungen in arithmetischen Zahlkörpern, insbesondere Kreisteilungskörpern, unterstützen soll. Die zugrundeliegende Programmiersprache ist eine im wesentlichen aufwärtskompatible Erweiterung der Programmiersprache Standard-Pascal [1, 2]. XPASCAL soll den fachkundigen, mit den Grundlagen von Pascal vertrauten Anwender in die Lage versetzen, einfachere Anwendungsbeispiele aus der Zahlentheorie auf komfortable Weise durchrechnen zu können, ohne dabei

- allzu früh auf Grenzen des darstellbaren Zahlbereichs zu stoßen,
- durch Rundungsfehler verfälschte Resultate zu erhalten,
- sich um die maschineninterne Realisierung der Arithmetik zu kümmern,
- von der gewohnten Schreibweise allzusehr abweichen zu müssen.

Natürlich ist dieser Komfort nicht ohne erheblichen Aufwand an Speicherplatz und Rechenzeit zu haben. Anwendungen, die mit den gebräuchlichen Programmiersprachen bequem lösbar sind, sollten daher nicht mittels XPASCAL in Angriff genommen werden.

Von den bekannten Systemen zum symbolischen Rechnen (beispielsweise MACSYMA, REDUCE etc.) unterscheidet sich XPASCAL wesentlich durch die kompatible Einbettung der Operationen in eine verbreitete Hochsprache.

XPASCAL wird derzeit realisiert als eigenständiges System unter dem Betriebssystem UNIX[1]. Für einen sinnvollen Einsatz ist ein Rechnersystem mit einem Hauptspeicherausbau von ca. 4 MByte und einem Prozessor von mindestens der Leistung eines MC 68000 angemessen. Da XPASCAL ein in der Entwicklung befindliches, experimentelles System ist, sind Anregungen, Verbesserungsvorschläge und Änderungswünsche

[1] UNIX ist ein eingetragenes Warenzeichen von AT&T Bell Laboratories.

im ganzen Bereich von der Benutzerschnittstelle über die Sprachdefinition bis hin zur aktuellen Realisierung willkommen.

Kapitel 2

Sprachumfang

XPASCAL ist weitgehend an Standard-Pascal [1] angelehnt. Änderungen ergeben sich zum einen aus dem erwünschten erweiterten Leistungsumfang und zum anderen aus daraus folgenden implementierungsbedingten Einschränkungen. In der Regel sollten Standard-Pascal-Programme auch unter XPASCAL, wenn auch mit verringerter Effizienz, ablauffähig sein.

Für die Beschreibung gilt generell folgendes (vgl. [1]):

- "es ist ein Fehler" bedeutet: die Operation ist unzulässig; das Ergebnis ist undefiniert. Nicht alle Fehler werden zur Übersetzungszeit oder zur Laufzeit erkannt.
- "das Ergebnis ist implementierungsdefiniert" bedeutet: es wird ein wohldefiniertes, aber nicht näher festgelegtes Resultat aus mehreren Möglichkeiten ausgewählt. Das Ergebnis ist brauchbar, kann aber in einer anderen Implementierung abweichend ausfallen.
- Als Metasprache wird eine Variante von EBNF verwendet. Terminalsymbole werden in "`"`" eingeschlossen, Nichtterminalsymbole in "`<`" und "`>`". Wiederholungen werden mit "`{`" und "`}`" geklammert, optionale Angaben mit "`[`" und "`]`". Alternativen werden mit "`|`" abgetrennt.

2.1 Einschränkungen gegenüber Pascal

2.1.1 Programmkopf

Als Programm-Parameter sind nur File-Namen zulässig. Die Standard-Dateien *input* und *output* brauchen nicht angegeben zu werden, da sie immer angeschlossen werden.

2.1.2 Globale Sprünge

Explizite Aussprünge aus Prozeduren auf globale Marken sind verboten. Von der Verwendung lokaler Sprünge wird dringend abgeraten.

Durch Aufruf der vordefinierten Prozedur *halt* kann der Programmlauf definiert beendet werden.

2.1.3 Conformant arrays

Das "conformant-array"-Konzept aus Standard-Pascal ist vorerst aus Aufwandsgründen nicht realisiert.

2.1.4 Variante Records

Variantenteile von Verbunden dürfen keine Komponenten eines Typs enthalten, der über Standard-Pascal hinausgeht.

2.1.5 Files

Der Komponententyp eines File-Typs darf keine Elemente eines Typs enthalten, der über Standard-Pascal hinausgeht. Der Komponententyp darf darüberhinaus keine Verbunde mit Varianten, Files oder Pointer enthalten.

2.1.6 Pack und Unpack

Die Standardprozeduren *pack* und *unpack* sind nicht implementiert, weil hier auch gepackte Reihungen indiziert werden dürfen.

2.2 Erweiterungen gegenüber Standard-Pascal

2.2.1 Lexikalische Darstellung

- Eingabezeilen können durch das Zeichen "\" auf die nächste Zeile fortgesetzt werden. Diese Wirkung des Zeichens "\" tritt nur am Zeilenende ein.
- Innerhalb von Bezeichnern darf das Zeichen "_" (Unterstrich) vorkommen. Groß-und Kleinbuchstaben werden, außer in Zeichenketten, nicht unterschieden. Bezeichner dürfen beliebig lang sein, alle Zeichen dienen der Unterscheidung.

- Zahlen dürfen beliebige Stellenzahl haben. Innerhalb einer Ziffernfolge darf das Zeichen "_" (Unterstrich) zum Abteilen von Zifferngruppen verwendet werden. Bei **double-**Zahlen, das sind **real-**Zahlen mit höherer Genauigkeit, wird der Exponent mit dem Zeichen "d" eingeleitet.

- Die als Erweiterung (siehe 2.2.16) vorgesehenen Operationen **bitand, bitor, bitxor, bitnot** können als reservierte Wortsymbole oder mittels der Zeichen "`&`", "`|`", "`%`", "`~`" codiert werden.

2.2.2 Einfügen von Texten

Durch eine Einfügungs-Angabe

```
<insertion> =
   "#include" <file-ref>

<file-ref> =
   "'" <string> "'" | "<" <string> ">"
```

in der ersten Spalte einer Quellzeile kann der Inhalt einer Textdatei in die zu übersetzende Quelle eingefügt werden. Der `<string>` enthält den Dateinamen nach den Konventionen des unterliegenden Betriebssystems. Solche Einfügungen können auch geschachtelt werden.

Diese Möglichkeit ist vor allem im Zusammenhang mit getrennt übersetzten Programmteilen von Bedeutung (siehe 2.3).

2.2.3 Deklarationsreihenfolge

Die Reihenfolge der Deklarationen eines Blocks ist weitgehend beliebig, jedoch dürfen jeweils nur bereits textuell vorher definierte Objekte verwendet werden; Ausnahme: Prozeduren und Funktionen, die "forward" (bzw. "external") spezifiziert sind.

Diese Erweiterung ergibt sich fast zwangsläufig aus der Möglichkeit, Texte einzufügen.

2.2.4 Gepackte Reihungen

Gepackte Arrays können indiziert werden. Komponenten von gepackten Arrays dürfen als Var-Parameter auftreten.

2.2.5 Erweiterte Konstantendefinition

Auf der rechten Seite einer Konstantendefinition darf ein konstanter Rechenausdruck stehen. Er darf als Operanden nur vordefinierte oder im Programm bereits besetzte Konstanten enthalten.

Konstanten, die in Deklarationen vorkommen, müssen bereits zur Übersetzungszeit auswertbar sein.

2.2.6 Arithmetische Grund-Datentypen

XPASCAL enthält ein erweitertes System von arithmetischen Standard-Typen:

integer wie in Standard-Pascal: ein Ausschnitt aus den ganzen Zahlen im Intervall $(-maxint)$ bis $maxint$. $maxint$ ist eine implementierungsspezifische vordefinierte Konstante (derzeit $2^{31} - 1 = 2147243647$).

integer ist kein Ring; es ist, wie in Standard-Pascal, ein Fehler, wenn eine der arithmetischen Grundoperationen $+, -, *$ aus dem Bereich **integer** herausführen würde. Ist damit nicht zu rechnen, so sollte **integer** anstelle von **long** verwendet werden, da der Rechenaufwand wesentlich geringer ist.

int gleichbedeutend mit **integer**.

long die ganzen Zahlen, soweit sie mit dem verfügbaren Speicher darstellbar sind (andernfalls bricht der Programmlauf ab). Die Resultate der Grundoperationen sind exakt. **long** ist daher ein Ring.

rat die rationalen Zahlen, soweit sie mit dem verfügbaren Speicher darstellbar sind. Die Resultate der Grundoperationen sind exakt. **rat** ist ein Körper.

real wie in Standard-Pascal: eine maschinenspezifische Approximation an die reellen Zahlen. Die Resultate der Grundoperationen sind mit Rundungsfehlern behaftet; daher ist **real** weder ein Ring noch ein Körper, jedoch oft eine brauchbare Annäherung daran.

double eine mindestens ebenso gute Approximation der reellen Zahlen wie **real**; ihre Verwendung bedingt eventuell höheren Rechenaufwand.

2.2.7 Erweiterte Grundtypen

Für bestimmte zahlentheoretische Anwendungen können die Bereiche **int**, **long**, **rat**, **real** und **double** jeweils durch symbolische Adjunktion einer primitiven p^n-ten Ein-

heitswurzel ζ erweitert werden. So ergeben sich die zusätzlichen Grundtypen:

ext_int bedeutet **int**[ζ],

ext_long bedeutet **long**[ζ],

ext_rat bedeutet **rat**[ζ],

ext_real bedeutet **real**[ζ],

ext_double bedeutet **double**[ζ] .

Für die Genauigkeit der Resultate und die Ring- bzw. Körper-Eigenschaft gilt das über die jeweiligen Grundtypen Gesagte.

Die Elemente der Erweiterungstypen kann man sich vorstellen als Polynome von minimalem Grad in ζ mit Koeffizienten vom jeweiligen Grundtyp.

Beispiel:

Für $p = 2$ und $n = 2$ entspricht **ext_real** *den komplexen Zahlen,* **ext_long** *den ganzen Gaußschen Zahlen.*

Hinweis:

ext_real *und* **ext_double** *sind nur in diesem Falle sinnvoll!*

2.2.8 Standard-Konstanten

Die Größen

- *basic_prime:* die Primzahl p,
- *basic_exponent:* der Exponent n,
- *primitive_root:* die primitive p^n-te Einheitswurzel ζ,
- *root_order:* die Ordnung p^n von ζ,
- *ext_degree:* der Grad $\phi(p^n) = (p-1) \cdot p^{n-1}$ der Erweiterungen

sind standardmäßig als Konstanten verfügbar. *primitive_root* ist vom Typ **ext_int**, die anderen Konstanten sind vom Typ **int**. Beim Start der Ausführung eines XPASCAL-Programms werden die aktuellen Werte von p und n vom Benutzer erfragt, bzw. sind über Start-Optionen einstellbar.

In Deklarationen dürfen Konstanten, die von den Standard-Konstanten abhängen, nicht vorkommen, da diese nicht bereits zur Übersetzungszeit zur Verfügung stehen.

Hinweis:

In der Praxis wird man die vorgegebenen Bezeichnungen mittels einer Konstantendefinition im Benutzerprogramm zweckmäßig problembezogen umbenennen. Das Gleiche gilt sinngemäß für die Standard-Typbezeichner der Erweiterungstypen.

Beispiel für $p = 2$ und $n = 2$:

```
const i = primitive_root;
type complex = ext_real;
     gauss = ext_long;
```

oder auch:

```
type Z = long;
     Q = rat;
     C = ext_real;
     R = ext_long;
     K = ext_rat;
```

2.2.9 Nichtarithmetische Typen

- Die aus Standard-Pascal bekannten Datentypen **char**, **boolean** (synonym: **bool**) sowie die frei definierbaren Aufzählungstypen sind weiterhin vorhanden.
- Die Mechanismen zur Konstruktion von zusammengesetzten Typen sind unverändert übernommen. Dabei zählen die Typen **long**, **rat** und **double**, ebenso wie **real**, nicht zu den skalaren Typen.
- Die neu eingeführten Typen (außer **double**) dürfen nicht als Komponenten eines *file* auftreten oder darin enthalten sein, ebensowenig im Variantenteil eines varianten *record*. Diese Einschränkung betrifft nicht die Ein- und Ausgabe auf Textdateien (vgl. 2.2.15).

2.2.10 Typverträglichkeit

Gemäß Tabelle 2.1 wird bei Wertzuweisung und bei Wert-Parametern der Wert des zuzuweisenden Ausdrucks bei Bedarf auf den Zieltyp ausgeweitet.

Wegen der oben (Abschnitt 2.2.7) genannten Einschränkungen ist der Übergang **int** → **long** in Strenge kein Ringhomomorphismus, ebenso der Übergang **rat** → **real** kein Körperhomomorphismus. Dasselbe gilt für die entsprechenden Übergänge bei den erweiterten Grundtypen.

int	→	**long**	→	**rat**	→	**real**	→	**double**
↓		↓		↓		↓		↓
ext_int	→	**ext_long**	→	**ext_rat**	→	**ext_real**	→	**ext_double**

Tabelle 2.1: Typausweitung

2.2.11 Typumwandlung

- Eine Einschränkung auf einen weniger mächtigen Typ ist bei Bedarf explizit vorzunehmen mittels der Standardfunktionen:

  Short: long → int
  bzw. **ext_long → ext_int**

  Trunc, Round: real → int
  bzw. **ext_real → ext_int**
  bzw. **rat → long**
  bzw. **ext_rat → ext_long**

  Single: double → real
  bzw. **ext_double → ext_real**

  Dabei gilt generell, daß die Einschränkung eines vorher implizit ausgeweiteten Wertes auf den ursprünglichen Typ zum ursprünglichen Wert führt; ansonsten hängt das Resultat von der Operation ab:

  – Für erweiterte Typen ist das Resultat komponentenweise zu berechnen.
  – Für die Grundtypen gilt:

    Short: es ist ein Fehler, wenn der **long**-Operand außerhalb des Bereichs **integer** liegt; sonst wird der Operand auf **integer** gekürzt.

    Single: liefert eine geeignete **real**-Approximation des Arguments vom Typ **double**; bei mehreren gleichwertigen Möglichkeiten ist das Ergebnis implementierungsdefiniert. Es ist ein Fehler, wenn das Argument außerhalb des mit dem Typ **real** darstellbaren Zahlbereichs liegt.

    Trunc: liefert die dem Absolutbetrag nach größte ganze Zahl, deren Betrag nicht größer als der des Arguments ist. Es ist ein Fehler, wenn diese Zahl außerhalb des Zielbereichs liegt.

    Round: Es gilt die Definition

    $$\begin{aligned} \textit{if } x \geq 0 \textit{ then } \mathit{Round}(x) &= \mathit{Trunc}(x + 0.5) \\ \textit{else } \mathit{Round}(x) &= -\mathit{Round}(-x) \end{aligned}$$

    Es ist ein Fehler, wenn $\mathit{Round}(x)$ außerhalb des Zielbereichs liegt.

  - Ist bei einem Erweiterungstyp für eine Komponente ein Fehler aufgetreten, so ist das Gesamtergebnis fehlerhaft.
  - Ist ansonsten für eine Komponente ein implementierungsdefiniertes Resultat aufgetreten, so ist das Gesamtergebnis implementierungsdefiniert.

- Die Standardfunktion

  Coeff: ext_X $\times$ **int** $\rightarrow$ **X**, mit **X** $\in$ {**int**, **long**, **rat**, **real**, **double**}

  liefert beim Aufruf $Coeff(x, k)$ den Koeffizienten von ζ^k aus der Polynomdarstellung von x aus. Der zweite Parameter muß zwischen 0 einschließlich und dem Grad $\phi(p^n) = (p-1) \cdot p^{n-1}$ ausschließlich liegen, sonst ist das Ergebnis undefiniert.

- Die Standardfunktion

  Conj: ext_X $\times$ **int** $\rightarrow$ **ext_X**, mit **X** $\in$ {**int**, **long**, **rat**, **real**, **double**}

  liefert beim Aufruf $Conj(x, k)$ den Wert, der sich ergibt, wenn man in der Polynomdarstellung von x konsistent ζ durch ζ^k ersetzt. Durchläuft k alle Werte von 2 bis $p^n - 1$, die nicht durch p teilbar sind, so erhält man jede Konjugierte von x genau einmal. Es gilt immer:

  $$Conj(x, 1) = x$$

  Liegt k nicht im Bereich von 0 bis p^n ausschließlich, oder ist k durch p teilbar, so ist das Resultat undefiniert.

- Mittels der Funktion

  Expand: int $\rightarrow$ **long**
  bzw. **ext_int** $\rightarrow$ **ext_long**

  kann ein Wert explizit ausgeweitet werden, um exakte Arithmetik zu erzwingen. Dies führt zu erheblich höheren Rechenzeiten und sollte daher nur dann verwendet werden, wenn andernfalls mit der Überschreitung des Grundbereichs **int** bzw. **ext_int** gerechnet werden muß.

2.2.12 Rechenoperationen

Die in Standard-Pascal definierten Grundoperationen sind soweit wie möglich aufwärtskompatibel erweitert worden. Im einzelnen gilt folgendes:

$+, -, *$**: T1** $\times$ **T2** $\rightarrow$ **T3**
: Das Typenschema aus Tabelle 2.1 bestimmt zu zwei beliebigen arithmetischen Grundtypen T1, T2 der Operanden den kleinsten gemeinsamen Erweiterungstyp T3. Er ist der Typ des Resultats, und auf ihn werden die Operanden bei Bedarf angepaßt. Siehe dazu auch Tabelle 2.2, 2.3.

−: T → T, Vorzeichenumkehr
Der Typ wird nicht geändert.

/: T1 × T2 → T3
Der Divisortyp wird ggf. soweit möglich über **int** → **long** → **rat** bzw. **ext_int** → **ext_long** → **ext_rat** ausgeweitet. Der Ergebnistyp bestimmt sich anschließend über Tabelle 2.2, 2.3.

div, mod: T1 × T2 → T3
Diese Operationen sind nur sinnvoll, falls T1 (bis auf Umfangsbeschränkungen) ein Ring ist; daher muß T1 ∈ {**int**, **long**, **ext_int**, **ext_long**} sein. T2 kann nur **int** oder **long** sein. Der Ergebnistyp bestimmt sich nach Tabelle 2.2, 2.3. Bei T1 ∈ {**ext_int**, **ext_long**} sind die Operationen komponentenweise zu verstehen.

=, ≠: T1 × T2 → T3
Die Operanden werden nach Tabelle 2.2, 2.3 angepaßt; der Ergebnistyp ist **boolean**.

<, >, ≤, ≥: T1 × T2 → T3
Als Operanden sind nur Elemente der Grundtypen zulässig; ansonsten wie bei "=".

∗∗: T1 × int → ext_int
T1 darf nur die Konstante ζ (= *primitive_root*) sein; der zweite Operand muß zwischen 0 und p^n liegen, sonst ist das Ergebnis undefiniert. Diese Operation ist nur zur bequemen Notation von Konstanten eines Erweiterungstyps vorgesehen.

	int	**long**	**rat**	**real**	**double**
int	**int**	**long**	**rat**	**real**	**double**
long	**long**	**long**	**rat**	**real**	**double**
rat	**rat**	**rat**	**rat**	**real**	**double**
real	**real**	**real**	**real**	**real**	**double**
double	**double**	**double**	**double**	**double**	**double**

Tabelle 2.2: Typverträglichkeit bei Grundtypen

2.2.13 Standardfunktionen

- Die Funktionen **sin, cos, exp, ln, sqrt, arctan, ord, pred, succ, chr, odd** sind weiterhin in gleicher Bedeutung vorhanden.

	T2	ext_T2
T1	T3	ext_T3
ext_T1	ext_T3	ext_T3

Tabelle 2.3: Typverträglichkeit bei Erweiterungstypen

- **abs: T → T**
  ist für beliebige arithmetische Grundtypen definiert.
- **sqr: T → T**
  ist für beliebige arithmetische Typen definiert.
- **Denom: rat → long**
  bzw. **ext_rat → long**

  liefert jeweils die kleinste positive ganze **long**-Zahl, mit der das Argument multipliziert werden muß, um einen mit dem Typ **long** bzw. **ext_long** verträglichen Wert zu erhalten.
- Die Funktionen **Coeff, Conj, Trunc, Round, Short, Single** und **Expand** sind bereits in 2.2.11 beschrieben.

2.2.14 Standardprozeduren

- Die Prozeduren **new** und **dispose** sind unverändert vorhanden.
- Die Prozeduren **pack** und **unpack** sind nicht implementiert; sie sind auch überflüssig, da auch gepackte Reihungen indiziert werden können.
- Die Prozedur **halt** beendet den Programmlauf definiert.

2.2.15 Ein- und Ausgabe auf Textdateien

- Die Prozeduren **reset, rewrite, put, get, page** und die Funktionen **eof, eoln** sind unverändert vorhanden.
- Bei den Prozeduren **write** und **writeln** sind auch Argumente der neu eingeführten arithmetischen Grundtypen zugelassen. Das Standard-Ausgabeformat für sie ist noch nicht endgültig festgelegt; jedenfalls kann ein mit **write** ausgegebener Wert eines der Grundtypen mit **read** auf eine Variable desselben Typs unverändert eingelesen werden. Explizite Formatangaben sind derzeit bei den neu eingeführten Typen wirkungslos.

- Bei **read** und **readln** sind auch Argumente der Typen **long**, **rat**, **double** zulässig.

  – Als externe Darstellung wird bei **long** nach evtl. vorhandenen führenden Leerzeichen und Zeilenwechseln und nach einem optionalen Vorzeichen eine Ziffernfolge beliebiger Länge erwartet, die nötigenfalls mit dem Zeichen "\" auf Folgezeilen fortgesetzt werden kann. Die Folgezeilen dürfen führende Leerzeichen enthalten. Mit dem Zeichen "_" (Unterstrich) können Zifferngruppen zur Erhöhung der Lesbarkeit abgeteilt werden. Jedes andere Zeichen schließt die Zahl ab.

  – Bei **rat** wird eine **long**-Zahl im angegebenen Format erwartet, an die wahlweise mit dem Zeichen "/" eine zweite **long**-Zahl unmittelbar angeschlossen werden kann. Als Resultat wird der exakte Quotient der beiden Zahlen geliefert, bzw. dann, wenn "/" und die zweite Zahl fehlen, die auf den Typ **rat** ausgeweitete einzige Zahl.

  – Bei **double** wird die entsprechende externe Darstellung wie bei **real** erwartet; der Exponent kann in beiden Fällen wahlweise mit "e" oder "d" eingeleitet werden. Die Anzahl der angegebenen Nachkommastellen ist beliebig, es wird geeignet gerundet.

2.2.16 Optionale Erweiterungen

Die folgenden Erweiterungen können durch Angabe der Compiler-Option `{$X+}` (vgl. Abschnitt 3.2) am Anfang des Quelltextes zusätzlich eingeschaltet werden:

- Ein Funktionsresultat darf von beliebigem Typ sein.

- Die Infix-Operationen **bitand, bitor, bitxor, bitnot** bedeuten die bitweise Verknüpfung ihrer **integer**-Operanden und liefern ein **integer**-Resultat.

- Bei **case** ist ein **else**-Zweig möglich:

```
<statement> =
   "case" <expression> "of"
     { <caselabel> {"," <caselabel> }
                        ":" <statement> ";" }
     [ "else" <statement> ]
   "end"
```

  Fehlt der **else**-Zweig, so ist der Aufruf eines nicht definierten **case**-Zweiges ein Fehler.

- Die auch anderweitig verbreitete Dahl-Schleife

```
<statement> =
   "loop" { <statement> ";" }
      { "exit" "if" <condition> ";"
                  { <statement> ";" } }
   "end"
```

  wurde zusätzlich aufgenommen. Sie wird mit dem dynamisch ersten **exit**-Zweig verlassen, dessen `<condition>` erfüllt ist.

- **integer**- oder **long**-Zahlkonstanten können, eingeleitet mit "`$`", hexadezimal angegeben werden. Die Hexadezimalziffern 10 – 15 werden durch "`a`" bis "`f`" oder "`A`" bis "`F`" codiert. Fortsetzen mit "\" und Abtrennen mit "`_`" ist auch hier möglich.

2.3 Getrennte Übersetzung

XPASCAL gestattet es, Programmteile getrennt zu übersetzen und externe Moduln anzuschließen, die sich wie benannte Bibliotheken verhalten.

Um dabei noch möglichst weitgehende Prüfungen zur Übersetzungszeit zu ermöglichen, ohne sich allzuweit vom Pascal-Standard zu entfernen, wurde folgende Konstruktion gewählt [8]:

Eine Übersetzungseinheit ist entweder ein Hauptprogramm oder ein benannter externer Modul, der gewisse Konstanten, Typen, Prozeduren und Funktionen extern zugänglich macht. Zur Ausführungszeit werden alle von einem Programm angesprochenen Objekte, soweit vorhanden, automatisch dazugeladen.

2.3.1 Modulbeschreibung

Zu jedem Modul gibt es eine Modulbeschreibung, welche die Deklarationen der extern zugänglichen Objekte enthält. Diese Modulbeschreibung wird mittels des *include*-Mechanismus (siehe 2.2.2) sowohl in den Quelltext des Moduls selbst wie auch in alle Übersetzungseinheiten, die Eingänge des Moduls verwenden, im äußersten Block textuell eingefügt. So kann zur Übersetzungszeit die korrekte Versorgung weitgehend abgeprüft werden. Zur Ladezeit wird außerdem überprüft, ob in verschiedenen Übersetzungseinheiten eingefügte, gleich benannte Einschübe textuell identisch sind. Dies fängt den sonst oft schwer erkennbaren Fehler ab, daß inkompatible Versionen zusammengebunden werden.

Eine Modulbeschreibung hat folgende Syntax:

```
<module-description> =
   { <constant-definition>
   | <type-definition>
   | <external-specification>  }

<external-specification> =
   <procedure-heading> ";" "external" <module-name> ";"
   | <function-heading> ";" "external" <module-name> ";"

<module-name> =
   <identifier>
```

2.3.2 Aufbau eines Moduls

Die Syntax von `<program>` ist, bis auf die zusätzlich eingeführte Möglichkeit der `<external-specification>`, unverändert.

Ein externer Modul gehorcht nach textueller Einfügung aller *include*-Teile der Syntax

```
<module> =
   "module" <module-name> ";"
   { <constant-definition>
   | <type-definition>
   | <external-specification>
   | <function-declaration>
   | <procedure-declaration> }
```

Globale Variablen im äußersten Block sind in einem externen Modul, (der nicht das Hauptprogramm ist,) nicht zugelassen; ein solcher Modul enthält auch keine ausführbaren Anweisungen außerhalb von Prozeduren und Funktionen.

Alle im äußersten Block eines Programms oder Moduls definierten oder importierten Namen müssen voneinander verschieden sein. Dagegen ist es zulässig, daß derselbe Prozedur- oder Funktionsname von verschiedenen Moduln exportiert wird; diese können dann aber nicht gleichzeitig verwendet werden, weil sonst Namenskonflikte durch Importe entstehen.

2.3.3 Zuordnung von Externbezügen und Eingängen

Ist der `<module-name>` in einer "external"-Angabe verschieden vom Namen des gerade übersetzten Moduls bzw. Programms, so handelt es sich um einen Externbezug. Der angesprochene Modul wird zur Laufzeit automatisch dazugeladen.

Ist der `<module-name>` in der "external"-Angabe identisch mit dem eigenen Modulnamen, so handelt es sich um die Angabe einer nach außen zugänglichen Prozedur oder Funktion. Die Extern-Spezifikation wird dann genau wie eine "forward"-Spezifikation behandelt; d.h. es muß die Deklaration der Prozedur bzw. Funktion im gleichen Objekt nachfolgen (ohne nochmalige Angabe der Parameter bzw. des Resultat-Typs). Natürlich kann ein Modul außerdem lokale Konstanten, Typen, Prozeduren und Funktionen besitzen und verwenden.

Kapitel 3

Praktische Verwendung

XPASCAL ist als ein zweistufiges System realisiert. Eine Vorübersetzungsphase überführt das vom Anwender vorgegebene Quellprogramm aus der erweiterten Pascal-Sprache in eine Zwischenform, die anschließend, auch wiederholt mit wechselnden Parameterwerten und unterschiedlichen Daten, durch einen Interpreter ausgeführt werden kann. Dies kann auch auf einem anderen Rechensystem geschehen, da der Interpreter in Standard-Pascal geschrieben und leicht portabel ist. Die vom Vorübersetzer erzeugten Zwischendateien sind auch leicht übertragbar.

3.1 Übersetzungsphase

Der Übersetzer wird unter UNIX aufgerufen mit dem Kommando

`xpas1` *Name*.`PAS`

Der Übersetzer erwartet ein Quellprogramm *Name*.PAS und erzeugt daraus ein Protokoll in einer Textdatei *Name*.LST, die auch alle gegebenenfalls anfallenden Fehlermeldungen enthält, sowie im Erfolgsfalle eine Zwischendatei *Name*.IMC mit dem zu interpretierenden Zwischencode.

Weiterhin wird wahlweise eine Textdatei *Name*.CDE erzeugt, die den Zwischencode aus *Name*.IMC in lesbarer Form enthält. Dies dient nur der Kontrolle des Übersetzungsvorganges, ebenso wie die Datei *Name*.IDL, welche außerdem bei Laufzeitfehlern dem Interpreter Zugriff auf die Bezeichner im Quellprogramm ermöglicht.

Verwendete *include*-Dateien müssen im gleichen Dateiverzeichnis vorhanden sein oder über absolute oder relative Pfadnamen angesprochen werden.

3.2 Compiler-Optionen

Ein Kommentar der Gestalt `<Kommentaranfang> "$" <Optionen> <Rest>` mit

```
<Kommentaranfang> ::= "{" | "(*"
       <Optionen> ::= <Option> { "," <Option> }
         <Option> ::= <Grossbuchstabe> ("+" | "-")
           <Rest> ::= { <Zeichen ausser Kommentarende> } <Kommentarende>
  <Kommentarende> ::= "}" | "*)"
```

kann zur Steuerung von Übersetzer-Optionen verwendet werden. Mit "+" wird die Option jeweils eingeschaltet, mit "-" ausgeschaltet.

Als Optionen sind bislang festgelegt:

A: Parserlauf protokollieren. Für Testzwecke reserviert.
Voreinstellung: aus.

B: Semantikteil protokollieren. Für Testzwecke reserviert.
Voreinstellung: aus.

C: Zuweisungen auf Unterbereiche und Array-Indizierung überprüfen.
Voreinstellung: aus.

D: Deklarierte Größen protokollieren.
Globale Größen und Standard-Größen werden nicht gedruckt. Siehe auch Optionen **G** und **Z**. Für Testzwecke reserviert.
Voreinstellung: aus.

E, F: Nicht belegt.

G: Bei Option **D** und **R** werden auch die globalen Größen gedruckt. Für Testzwecke reserviert.
Voreinstellung: aus.

H: Listing seitenweise abteilen.
Voreinstellung: ein.

I, J: Nicht belegt.

K: Zwischencode lesbar ausgeben.
Voreinstellung: ein.

L: Listing ein- und ausschalten.
Voreinstellung: ein.

M, N: Nicht belegt.

O: Zwischencode optimieren.
Voreinstellung: ein.

P: Pointer bei Dereferenzierung auf Wert $\neq$ **nil** überprüfen.
Voreinstellung: aus.

Q: Nicht belegt.

R: Erreichbare Größen protokollieren.
Globale Größen und Standard-Größen werden nicht gedruckt. Siehe auch Optionen **G** und **Z**. Für Testzwecke reserviert.
Voreinstellung: aus.

S: Auf Einhaltung des Pascal-Standards prüfen. (Nicht realisiert).
Diese Option darf nur vor dem **program**-Statement gesetzt werden.
Voreinstellung: aus.

T: Stringverarbeitung wie in Turbo-Pascal aktivieren.
Nicht realisiert.

U, V: Nicht belegt.

W: Namensliste absetzen.
Voreinstellung: ein.

X: Erweiterten Standard aktivieren:

- **loop**-Anweisung,
- **else**-Zweig bei **case**,
- **bitand, bitor, bitxor, bitnot**-Operationen,
- beliebiger Resultattyp bei Funktionen,
- Hexadezimalkonstanten

sind zugelassen.
Diese Option darf nur vor dem **program**-Statement gesetzt werden.
Voreinstellung: aus.

Y: Intern reserviert.

Z: Bei Option **D** und **R** werden auch die Standard-Größen gedruckt. Für Testzwecke reserviert.
Voreinstellung: aus.

3.3 Ausführungsphase

Der Interpreter wird unter UNIX aufgerufen mit dem Kommando

`xpas2` *Name* { *filename* | *option* }

Die *filename*-Angaben werden in der Reihenfolge ihres Auftretens den Programm-Parametern zugeordnet.

Als Optionen sind derzeit zulässig:

`-p`*Zahl* **:** Vorgabe der Primzahl p

`-n`*Zahl* **:** Vorgabe des Exponenten n

`-o` **:** Einschalten Testmodus (siehe [10], dort sind auch weitere Testmöglichkeiten beschrieben).

Die Reihenfolge der Optionsangaben ist bedeutungslos, ebenso können Optionsangaben und File-Namen beliebig gemischt werden.

Der Interpreter erwartet eine Eingabedatei *Name*.IMC mit dem Zwischencode des Hauptprogramms. Bei seinem Start erfragt er, soweit nicht durch Start-Optionen voreingestellt, vom Anwender die aktuellen Werte für die Primzahl p und den Exponenten n, welche die jeweils gültigen Erweiterungstypen für den Lauf des Programms festlegen. Die Standard-Konstanten sind dann mit den aktuell gültigen Werten verfügbar.

Werden im Hauptprogramm Routinen aus anderen Moduln angesprochen, so werden diese, soweit vorhanden, automatisch dazugeladen und angeschlossen. Für jeden angesprochenen Modul *Modul* muß die Datei *Modul*.IMC vorhanden sein. Fehlt ein angesprochener Modul, so wird der Lauf abgebrochen, ebenso dann, wenn sich beim Laden Inkonsistenzen der Schnittstellen herausstellen.

Bei Laufzeitfehlern wird die Fehlerstelle und die aktuelle Aufrufverschachtelung angegeben. Sofern die Datei *Name*.IDL zur Verfügung steht, kann auf die aktuellen Werte der sichtbaren Variablen unter Bezug auf die im Quellprogramm verwendeten Bezeichner zugegriffen werden. (Bislang noch nicht realisiert).

3.4 Implementierungs-Einschränkungen

- Mengentypen können maximal 256 Elemente enthalten.
- Die Ordinalwerte der Grenzen des Grundtyps eines Mengentyps müssen im Bereich von 0 bis 255 einschließlich liegen.
- Aufzählungstypen können maximal 256 Elemente enthalten.

- Zeichenketten dürfen maximal 255 Zeichen lang sein.
- In Deklarationen auftretende Konstanten dürfen nicht größer als 32767 sein.
- *maxint* hat den Wert $2^{31} - 1 = 2147243647$.
- Es dürfen maximal ca. 2000 verschiedene Bezeichner, Konstanten und Strings auftreten.
- Die Summe der Längen voneinander verschiedener Bezeichner, Konstanten und Strings darf einen Wert von ca. 25000 nicht überschreiten.
- Der Wert von p^n (siehe 2.2.8) muß kleiner als $2^{15} = 32768$ sein. In der Praxis führen schon wesentlich kleinere Werte zu unerträglich hohem Zeit- und Speicherbedarf.

Kapitel 4

Stand der Realisierung

Zum Zeitpunkt der Abfassung dieses Berichts (18. Dezember 1998) liegt eine erste ablauffähige Version des Gesamtsystems vor.

Der Übersetzer ist bis auf einige unwesentliche Details fertiggestellt. Er ist in MODULA-2 [3] geschrieben und wurde unter Verwendung des Compiler-Generators COCO [4] entwickelt [5, 6]. Die semantische Analyse lehnt sich an Ideen aus [12] an. Der Übersetzer wurde auf einem ATARI 1040 ST entwickelt und auf eine SUN 3-Workstation unter UNIX übertragen.

Der Interpreter für den Standard-Pascal-Kern ist fertiggestellt [9], ebenso die Erweiterung auf den vollen Sprachumfang [10]. Implementierungssprache ist Standard-Pascal. Die Realisierung der Langzahlarithmetik und der Erweiterungstypen ist ebenfalls vollendet [11]. Da sich der Hauptspeicherbedarf der Langzahlarithmetik als unerwartet groß herausgestellt hat, wurde eine Neuimplementierung dieses Teiles mit anderen Datenstrukturen durchgeführt, die nahezu abgeschlossen ist [7]. Das Modulkonzept ist spezifiziert und implementiert [8], die Versionsüberprüfung ist noch nachzurüsten.

Die oben erwähnte Fehleranalyse bei Laufzeitfehlern ist als Fernziel ins Auge gefaßt.

Literaturverzeichnis

[1] K. Däßler, M. Sommer: PASCAL: Einführung in die Sprache. Norm-Entwurf DIN 66256.
Springer Verlag 1983.

[2] K. Jensen, N. Wirth: Pascal User Manual and Report.
Springer Verlag 1978.

[3] N. Wirth: Programming in MODULA-2.
Springer Verlag 1983.

[4] P. Rechenberg, H. Mössenböck: Ein Compiler-Generator für Mikrocomputer.
Hanser Verlag 1985.

[5] G. Neusetzer: Implementierung eines Syntax- und Semantikprüfers für Pascal mittels einer erweiterten Attribut-Grammatik.
Studienarbeit Nr. 643, 1987.
Institut für Informatik, Universität Stuttgart.

[6] G. Neusetzer: Entwicklung einer Pascal-Variante für zahlentheoretische Anwendungen und eines Compilerkerns dafür.
Diplomarbeit Nr. 676, 1988.
Institut für Informatik, Universität Stuttgart.

[7] S. Robitschko, Neuimplementierung der Langzahl-Arithmetik für XPASCAL (Arbeitstitel).
Diplomarbeit, 1989.
Institut für Informatik, Universität Stuttgart.

[8] Th. Schöbel: Ein Modulkonzept für XPASCAL.
Studienarbeit Nr., 1989.
Institut für Informatik, Universität Stuttgart.

[9] U. Schoppe: Prototyp eines Pascal-P-Code Interpreters.
Studienarbeit Nr. 694, 1988.
Institut für Informatik, Universität Stuttgart.

[10] U. Schoppe: Ein Code-Interpreter für XPASCAL.
Diplomarbeit Nr. 568, 1988.
Institut für Informatik, Universität Stuttgart.

[11] G. Wahl: Entwurf und Implementierung einer Langzahl-Arithmetik für Kreisteilungskörper.
Diplomarbeit Nr. 546, 1988.
Institut für Informatik, Universität Stuttgart.

[12] D. A. Watt: An Extended Attribute Grammar for Pascal.
SIGPLAN Notices **14**, 60 - 74, Feb. 1979.

Anhang A

Programmbeispiel für erweiterte Arithmetik

```
program xtest(input, output);

(* Demonstrationsprogramm
   fuer erweiterte Arithmetik
   in Kreisteilungskoerpern *)
(* Standard-Bezeichnungen komfortabel umbenennen ! *)

(* Standard-Konstanten, beim Start vorgegeben *)

const zeta = primitive_root;    (* Einheitswurzel der Ordnung p**n *)
        p  = basic_prime;       (* Primzahl *)
        n  = basic_exponent;    (* Exponent *)
        pn = root_order; (* Ordnung von zeta *)
        f  = ext_degree; (* Grad der Erweiterung *)

(* Standard-Typen *)

type Z = long_integer;
     Q = long_rational;
     R = extended_integer;      (* Z[zeta] *)
     K = extended_rational;     (* Q[zeta] *)

(* lokale Groessen *)

var i: integer;
    a, b: Z;
    s, t: Q;
```

```
    w, x, y: R;
    u, v: K;

(* Beispiele fuer nicht standardmaessig vorgegebene Operationen *)

function Compl(x: K): K;
(* Produkt der Konjugierten *)
var i: integer;
    y: K;
begin y := 1;
      for i := 2 to pn do
          if i mod p <> 0 then y := y * Conj(x, i);
      Compl := y;
end;

function Norm(x: K): Q;
begin Norm := Coeff(x * Compl(x), 0);
end;

function Rez(x: K): K;
(* multiplikatives Inverses *)
var y: K;
    d: Q;
begin y := Compl(x);
      d := Coeff(x * y, 0);
      Rez := y / d;
end;

(* Beginn Hauptprogramm *)

begin writeln('Test erweiterte Arithmetik');
      writeln('p = ', p:3 , ',  n = ', n:3);
      writeln('(1 - zeta)**f = p * u');

      x := 1 - zeta;
      y := 1;
      for i := 1 to f do y := y * x;
      writeln('(1 - zeta)**f = ', y);

      u := y / p;
      writeln('u = ', u);

      a := Denom(u);
```

```
      if a = 1 then writeln('u ist ganzes Element') else
                   writeln('u ist kein ganzes Element');

      x := Trunc(u);
      writeln('Trunc(y / p) = ', x);
      writeln('y div p      = ', y div p);
      if (y div p) <> x then writeln('Fehler bei Trunc oder div');

      w := Trunc(a * u);
      writeln('Zaehler von u ist ', w);
      writeln('Nenner  von u ist ', a);

      s := Norm(u);
      writeln('Norm(u) = ', s);

      if s = 1 then writeln('u ist Einheit')
               else writeln('u ist keine Einheit');
end.
```

Anhang B

Beispiel für eine Modulbeschreibung

```
{Definitionen der elementaren transzendenten Funktionen}
{oeffentlich zugaenglich}
{Realisierung im Modul "carith"}

function csqrt(z: complex): complex; external carith;

function cexp(z: complex): complex; external carith;

function cln(z: complex): complex; external carith;

function csin(z: complex): complex; external carith;

function ccos(z: complex): complex; external carith;

function carctan(z: complex): complex; external carith;
```

Anhang C

Beispiel für einen externen Modul

```
module carith;

{elementare transzendente Funktionen im Komplexen}

type complex = ext_real;

#include "carith.def"
{Funktionsdefinitionen, extern sichtbar}

const i = primitive_root;
      pi4 = arctan(1);
      pi2 = pi4 * 2;
      pi = pi2 * 2;


function csqrt{(z: complex): complex};
{komplexe Quadratwurzel, Hauptwert}
var x, y, d, e: real;
begin x := coeff(z, 0); y := coeff(z, 1);
  d := sqrt(sqr(x) + sqr(y));
  if x > 0 then
  begin e := sqrt((d + x)/2); csqrt := e + y/(2 * e) * i;
  end else
  begin e := sqrt((d - x)/2);
    if y >= 0 then csqrt := y/(2 * e) + e * i
              else csqrt := -y/(2 * e) - e * i
  end;
end;
```

```
function cexp{(z: complex): complex};
{komplexe Exponentialfunktion}
var x, y, d: real;
begin x := coeff(z, 0); y := coeff(z, 1);
  d := exp(x);
  cexp := d * cos(y) + d * sin(y) * i;
end;

function cln{(z: complex): complex};
{komplexer Logarithmus, Hauptwert}
var x, y, d: real;
    e: complex;
begin x := coeff(z, 0); y := coeff(z, 1);
  d := sqrt(sqr(x) + sqr(y));
  if x > 0 then cln := ln(d) + arctan(y/(d + x)) * 2 * i
  else begin e := ln(d) - arctan(y/(d - x)) * 2 * i;
         if y > 0 then cln := e + i * pi else cln := e - i * pi;
       end;
end;

function carctan{(z: complex): complex};
{komplexer Arcustangens, Hauptwert}
var x, y, d, e, f: real;
    g: complex;
begin x := coeff(z, 0); y := coeff(z, 1);
  e := sqr(x); d := 1 - e - sqr(y);
  f := sqrt(4*e + sqr(d));
  if d > 0 then
  carctan := arctan(2 * x / (f + d))
             - ln((e + sqr(1 - y))/(e + sqr(1 + y)))/4 * i
  else begin g := -arctan(2 * x / (f - d))
                  - ln((e + sqr(1 - y))/(e + sqr(1 + y)))/4 * i;
         if x > 0 then carctan := g + pi2
         else carctan := g - pi2;
       end;
end;

function csin{(z: complex): complex};
{komplexer Sinus}
var x, y, d, e: real;
begin x := coeff(z, 0); y := coeff(z, 1);
  d := exp(y); e := 1/d;
  csin := (d + e) * sin(x)/2 + (d - e) * cos(x)/2 * i;
```

```
end;

function ccos{(z: complex): complex};
{komplexer Cosinus}
var x, y, d, e: real;
begin x := coeff(z, 0); y := coeff(z, 1);
  d := exp(y); e := 1/d;
  ccos := (d + e) * cos(x)/2 - (d - e) * sin(x)/2 * i;
end;
```

Anhang D

Beispiel für einen Haupt-Modul

```
program modtest(output);

{Test fuer komplexe transzendente Funktionen}

type complex = ext_real;

#include "carith.def"
{Definitionen der elementaren transzendenten Funktionen}

const i = primitive_root;
      pi4 = arctan(1);
      pi2 = pi4 * 2;
      pi = pi2 * 2;

var k: integer;
    a, b, c: complex;

procedure cwrite(z: complex; m, n: integer);
{formatierte Ausgabe komplexer Zahlen}
begin write(coeff(z, 0):m:n, ' + i * ', coeff(z, 1):m:n);
end;

function cinv(z: complex): complex;
{Variante von 1/z, direkt ausprogrammiert}
var x, y, d: real;
begin x := coeff(z, 0); y := coeff(z, 1);
  d := sqr(x) + sqr(y);
  cinv := x/d - y/d * i;
end;
```

```
const p = basic_prime;
      n = basic_exponent;
{Parameter der Koerpererweiterung}

begin writeln('Test komplexe Arithmetik');
  if (p <> 2) or (n <> 2)
  {pruefen, ob richtiger Zahlkoerper eingestellt}
  then writeln('falscher Zahlbereich, p =', p, ', n =', n)
  else
  begin a := 6 + i * 8;
    for k := 1 to 20 do
    begin cwrite(a, 8, 4); writeln(' = a');
          cwrite(1/a, 8, 4); writeln(' = 1/a');
          cwrite(csqrt(a), 8, 4); writeln(' = sqrt(a)');
          cwrite(cexp(a), 8, 4); writeln(' = exp(a)');
          cwrite(cln(a), 8, 4); writeln(' = ln(a)');
          cwrite(csin(a), 8, 4); writeln(' = sin(a)');
          cwrite(ccos(a), 8, 4); writeln(' = cos(a)');
          cwrite(carctan(a), 8, 4); writeln(' = arctan(a)');
          writeln;
          cwrite(1/a*a-1, 8, 4); writeln(' = 1/a * a - 1');
          cwrite(sqr(csqrt(a))-a, 8, 4); writeln(' = sqr(sqrt(a)) - a');
          cwrite(csqrt(sqr(a))/a, 8, 4); writeln(' = sqrt(sqr(a)) / a');
          cwrite(cexp(cln(a))-a, 8, 4); writeln(' = exp(ln(a)) - a');
          cwrite((cln(cexp(a))-a)/pi, 8, 4);
          writeln(' = (ln(exp(a)) - a) / pi');
          cwrite(sqr(csin(a))+sqr(ccos(a))-1, 8, 4);
          writeln(' = sqr(sin(a)) + sqr(cos(a)) - 1');
          cwrite((carctan(csin(a)/ccos(a))-a)/pi, 8, 4);
          writeln(' = (arctan(sin(a)/cos(a)) - a) / pi');

          writeln;
          a := a * 0.8 * (0.32 + i * 0.96);
    end;
    writeln('Ende Test komplexe Arithmetik');
  end;
end.
```