Studienarbeit

vorgelegt von
Jürgen Nickelsen
Matrikel-Nummer 89816

Thema:
Parsierung von Make- und Shapefiles

Aufgabensteller:
Prof. Dr. Stefan Jähnichen

1. Einleitung

Diese Arbeit beschreibt die Konzeption und die Implementierung eines Parsers für „Shape“, einem Bestandteil von ShapeTools.

ShapeTools ist ein Programmsystem zur Versions- und Konfigurationsverwaltung in Software-Projekten. Das Programm „Shape“ ist ein zentraler Bestandteil von ShapeTools. Es übernimmt die Konfigurierung und Herstellung von Softwaresystemen, das heißt die Identifizierung der zur Herstellung benötigten Software-Objekte und die eigentliche Herstellung des Systems aus diesen Komponenten.

Shape ist ähnlich zu „Make“, indem es ein Systemmodell interpretiert, in dem die Quellkomponenten und die daraus herzustellenden Objekte aufgeführt sind. Shape hat die zusätzlichen Funktionalitäten, von jeder Quellkomponente des Systemmodells die gewünschte Version für den Herstellungsprozeß auszuwählen und verschiedene Varianten des Systems herstellen zu können.

Ebenso wie Make liest Shape dieses Systemmodell aus einer Systembeschreibungsdatei (*Shapefile*). In dieser Datei ist die abstrakte Systemstruktur festgelegt. Produktionsregeln beschreiben, wie aus Quelldokumenten abgeleitete Objekte erzeugt werden. Zusätzlich enthält die Systembeschreibungsdatei von Shape Regeln für die Versionsauswahl von Quellkomponenten und Beschreibungen von Systemvarianten. Die Syntax der Shapefiles ist ähnlich der von Makefiles; Shape kann reguläre Makefiles interpretieren.

Der in dieser Arbeit beschriebene Parser ist der Teil von Shape, der die Syntax des Shapefiles überprüft und das darin beschriebene Systemmodell liest. Dieser Parser ist eine Reimplementierung des bisher (bis ShapeTools Version 1.3) in Shape enthaltenen Parsers.

In dieser Arbeit wird nach der Klärung einiger Begriffe auf die Syntax und Semantik von Make- und Shapefiles eingegangen. Der neue Parser wird mit den Parsern verschiedener Make-Implementierungen und dem alten Parser von Shape verglichen. Nach einer Beschreibung der Implementierung des Parsers gehe ich noch kurz auf die mit dem neuen Parser gemachten Erfahrungen ein. Im Anhang finden sich Literaturanga ben, die Shapefile-Grammatik, die Quelltexte des Parsers und eine kurze Beschreibung von ShapeTools.

2. Begriffe

- Abhängigkeitsregeln (*dependency rules*)

  Ein Makefile besteht aus *Makrodefinitionen* und Abhängigkeitsregeln. Die Abhängigkeitsregeln geben an, wie ein oder mehrere *Targets* von *Dependents* abhängen; wenn sich ein Dependent geändert hat, muß das Target neu erzeugt werden. Zusätzlich kann in den Abhängigkeitsregeln spezifiziert werden, wie die Targets aus den Dependents erzeugt werden können. Das geschieht durch Kommandos, die von der Shell ausgeführt werden (*Shell-Skript*). In diesem Fall ist eine Abhängigkeitsregel gleichzeitig eine Produktionsregel.

  Generische Abhängigkeitsregeln sind Regeln, die Abhängigkeiten zwischen bestimmten Typen von Objekten beschreiben. Diese Typen werden dabei durch ein Suffix oder ein Pattern im Namen angegeben. Generische Regeln werden dann verwendet, wenn keine explizite Regel existiert, um ein Target zu erzeugen.

  Implizite Regeln sind Regeln, die in Make bzw. Shape eingebaut sind. Implizite Regeln sind durchweg generische Regeln.

  Explizite Regeln sind Regeln, in denen Targets und Dependents explizit angeführt werden.
  [make(1)], [shape(1)], [Tutorial]

- Dependent

  Ein Dependent ist ein Objekt, von dem das Target einer Abhängigkeitsregel abhängt. Ein Dependent kann ein Quelldokument oder ein Objekt sein, das in einer anderen Abhängigkeitsregel als Target vorkommt. Das muß nicht immer eine Datei sein, sondern ist oft auch ein *Special Target*.

- Target

  Ein Target ist das Objekt in einer Abhängigkeitszeile, daß von den Dependents abhängig ist. Dabei ist das Target nicht immer eine Datei, sondern kann auch ein *Special Target* sein.

  Ein Target wird bei Make „aktuell“ oder „up to date“ genannt, wenn das zu erzeugende Objekt neuer ist als die Dependents, von denen es abhängt. Bei Shape ist das komplizierter, da die Aktualität eines Targets zusätzlich von den aktivierten *Varianten* und *Versionsauswahlregeln* abhängt.

- Special Target

  Ein Special Target ist ein Target, das keine Datei ist.

  Ein typischer Fall dafür ist das sehr verbreitete Target „`clean`“, das dazu dient, daß der Benutzer mit dem Aufruf „`make clean`“ nicht mehr benötigte Dateien löschen kann (typischerweise Objekt-Dateien und andere temporär erzeugte Dateien). Die Abhängigkeitsregel für „`clean`“ ist dabei im allgemeinen so gegeben, daß *keine* Datei „`clean`“ erzeugt wird.

  Weitere Special Targets dienen dazu, das Verhalten von Make oder Shape zu steuern.
  [Sun Make], [Dmake], [GNU Make], [shape(1)]

- Makros

  Make hat einen einfachen Makromechanismus für Ersetzungen in Abhängigkeits- und Kommandozeilen. Makros werden dabei mit Zeilen der Form „`NAME = WERT`“ definiert und mit „`$(NAME)`“ referenziert, wobei die Referenz durch `WERT` erzetzt wird. Diese Ersetzung passiert rekursiv, eine Makroreferenz kann also auch Teil einer Makrodefinition sein.

  Die Nachfolger von Make (s.u.) sind in ihren Möglichkeiten gerade im Makromechanismus teilweise stark erweitert.
  [Make], [Sun Make], [GNU Make], [shape(1)]

- Quoting

  In manchen Fällen ist es nötig, Zeichen, die eine Sonderbedeutung haben, literal zu verwenden. Dafür müssen sie auf bestimmte Weise „geschützt“ werden, um ihre Interpretation als Sonderzeichen zu verhindern. Das geschieht im allgemeinen durch einfache oder doppelte Anführungszeichen („`'`“, „`"`“).

- das „alte“ Shape

  Mit dem „alten“ Shape ist vor allem die alte Implementierung des Parsers gemeint, der sich in der Syntax und Fehlerbehandlung auch nach außen hin von dem hier vorgestellten Parser unterscheidet.
  Ich benutze das Wort „Unix“ in dieser Schreibweise nicht für das Produkt UNIX† der Firma USL, sondern als Oberbegriff für die UNIX-ähnlichen Betriebssysteme verschiedener Hersteller wie z.B. HP-UX (Hewlett-Packard), SunOS (SunSoft), Ultrix (Digital Equipment) oder das BSD Unix der University of California at Berkeley.

- Unix

- Varianten

  „Variants are versions that are not related through the predecessor/successor relationship, but represent equivalent instantiations of the same concept, the invariant of several variants. There is no commonly agreed concept of variant other than this in the software engineering community.“ [Tutorial]
  Varianten werden zum Beispiel benutzt, um Code für verschiedene Plattformen zu erzeugen, um das System mit verschiedenen Features erzeugen zu können, etc. Dabei können auch mehrere Varianten gleichzeitig aktiv sein, wenn sie sich nicht gegenseitig ausschließen.

---

† UNIX is a trademark of Unix System Laboratories

- Variant-Class

  „Variant classes are groups of mutually exclusive variant definitions.“ [Tutorial]

  Beispiele dafür sind Varianten für verschiedene Compiler oder verschiedene Zielplattformen, wobei es nicht sinnvoll ist, zwei Varianten für verschiedene Zielplattformen gleichzeitig zu aktivieren; diese würde man in einer Variant-Class zusammenfassen. Eine Variante für einen bestimmten Compiler und eine Variante für eine bestimmte Zielplattform können dagegen ohne weiteres gleichzeitig aktiviert werden.

  Falls zwei Varianten aus einer Variant-Class gleichzeitig aktiviert werden, erkennt Shape das als einen Fehler und bricht mit einer entsprechenden Meldung ab.

- Versionsauswahl (*version binding*)

  Der Prozeß, der den Übergang von einem Namen zu einer konkreten Version beinhaltet, wird Versionsauswahl genannt. [vbind]
  Diese Auswahl folgt bestimmten Regeln wie z.B. „nimm das neueste“ oder „nimm, falls vorhanden, das zuletzt veröffentlichte, oder sonst das zuletzt gesicherte“. Diese *Versionsauswahlregeln* werden im Shapefile definiert und können den Erfordernissen entsprechend gewählt werden.

3. Make

Das Konzept von Make ist beschrieben in [Feldman]:

> „The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.“

3.1. Makefiles

Makefiles haben eine recht einfache Grammatik:

```
makefile = {macrodef | deprule}*

deprule  = {NAME}+ {COLON | DOUBLECOLON} {NAME}* {SHELLINE}*

macrodef = NAME "=" {CHAR}*
```

Das Makefile besteht aus einer beliebigen Anzahl von Makrodefinitionen und Anhängigkeitsregeln. Eine Abhängigkeitsregel besteht aus mindestens einem Namen, dem Trennzeichen (das ein einfacher oder doppelter Doppelpunkt sein kann), einer weiteren Liste von Namen und einer Liste von Kommandozeilen (für die Shell). Dieses Shell-Skript muß mit einem Tabulator eingerückt sein. Makrodefinitionen bestehen aus einem Namen, einem Gleichheitszeichen und dahinter weiteren Zeichen.

Die lexikalische Syntax ist nicht genau spezifiziert. Daraus ergeben sich die größten Unterschiede bei den einzelnen Make-Derivaten.

3.1.1. Probleme mit der Makefile-Syntax

Das Design der Eingabesprache von Make gehört nicht zu den Stärken dieses Programms. Es gibt mehrere problematische Punkte:

- Die Einrückung der Regeln mit einem Tabulator führt häufig zu Problemen, weil der Tabulator in vielen Texteditoren optisch nicht von der entsprechenden Anzahl Leerzeichen zu unterscheiden ist. Genausowenig ist meistens ein Tabulator zu unterscheiden von ein paar Leerzeichen gefolgt von einem Tabulator.

  Daraus resultieren gerade für den weniger geübten Benutzer häufig Fehler, indem im Makefile Leerzeichen stehen, wo ein Tabulator vorgesehen war.

- Es gibt keine Regeln für das Quoting von Make-Meta-Zeichen („=“, „:“, etc.) außer dem „$“ für Makroreferenzen. Das hat zur Folge, daß sich die Make-Derivate in diesem Punkt sehr verschieden

verhalten.

- Makefiles sind zeilenorientiert. Fortsetzungszeilen werden durch einen Backslash („\“) am Ende gekennzeichnet. Stehen nun hinter diesem Backslash noch Leerzeichen oder Tabulatoren (die in vielen Texteditoren nicht zu sehen sind), ist die folgende Zeile für Make *keine* Fortsetzungszeile.

3.2. Make-Implementierungen

Mittlerweile gibt es eine Reihe von Make-Implementierungen. Die erste Make-Implementierung von Stu Feldman lag mir leider nicht vor — ich hätte sie als „Reinform“ gerne näher untersucht. BSD Make (aus 4.3 BSD der UCB) und System V Make (aus UNIX System V von USL) stammen beide direkt davon ab. Wesentliche Erweiterungen sind VPATH und Behandlung von Object Libraries (nur System V). [BSD Make]

Weitere, modernere Implementierungen sind Sun Make (Bestandteil von SunOS der Firma SunSoft Inc. bis Version 4), GNU Make (aus dem GNU Projekt der Free Software Foundation), Dmake, Reno-Make (aus 4.3-Reno BSD der UCB) und Pmake (aus dem Sprite-Projekt der UCB), sowie eine ganze Reihe weiterer, teilweise herstellerspezifischer Make-Derivate. Alle sind gegenüber den eher „traditionellen“ Versionen aus 4.3 BSD und System V stark erweitert.

Übliche Features sind:

- Erweiterte Funktionalität bei Makros, z.B. String-Substitutionen
- Fallunterscheidungen
- Zugriff auf SCCS- oder RCS-Archive
- Paralleles Bauen des Systems auf verschiedenen Rechnern/CPUs

etc.

Wenn ich im folgenden über verschiedene Make-Derivate spreche, meine ich Sun Make, GNU Make und BSD Make, die ich näher untersucht habe.

4. Shape

Die Konzeption von Shape verfolgt bei der Verwaltung von Software einen sehr viel ambitionierteren Ansatz als Make; trotzdem sollen weiterhin normale Makefiles verarbeitet werden können. Shape-Beschreibungsdateien enthalten daher sowohl dieselben Elemente wie Makefiles als auch eine Reihe von neuen Konstrukten. [MahlerLampen88], [shape(1)]

Makrodefinitionen und Abhängigkeitsregeln, die einen großen Teil der Shapefiles ausmachen, sind syntaktisch gleich zu Make. Dazu kommen die Shape-spezifischen Teile:

Variantendefinitionen

In einer Variantendefinition werden die für eine Variante gültigen Makrowerte definiert.

Der Kopf der Variantendefinition ist der Name der Variante und das für eine Variantendefinition signifikante Token „`:+`“. Im Rumpf der Variantendefinition stehen dann die Makrodefinitionen, um einen Tabulator eingerückt.

Diese Syntax wurde vom alten Shape noch nicht unterstützt.

Vclass-Definitionen

„Vclass“ steht für „Variant-Class“, d.h. eine Gruppe von Varianten, die sich gegenseitig ausschließen. z.B. verschiedene Zielplattformen, verschiedene Compiler, etc.

Eine Vclass-Definition beinhaltet den Namen der Vclass (der ansonsten nirgends mehr auftaucht; er hat lediglich dokumentierenden Wert) und die Namen der zu dieser Vclass gehörenden Varianten.

Das sieht zum Beispiel so aus:

```
vclass compiler ::= (gcc, pcc)
```

[MahlerLampen88]

Versionsauswahlregeln

Die Versionsauswahlregeln definieren die Auswahlstrategie von Versionen.

Der Kopf der Versionsauswahlregel ist der Name der Regel und das für eine Versionsauswahlregel signifikante Token „:-“. Der Rumpf der Regel besteht aus Zeilen, die um einen Tabulator eingerückt sind.

Diese Syntax wurde vom alten Shape noch nicht unterstützt.

RULE- und VARIANT-Sections

Rule- und Variant-Sections waren beim alten Shape die vorgesehene Möglichkeit, Versionsauswahlregeln und Variantendefinitionen aufzuschreiben. Die Versionsauswahlregeln standen in einer Rule-Section, die Variantendefinitionen in einer Variant-Section. Eine Section begann mit „`#%VARIANT-SECTION`“ bzw. „`#%RULE-SECTION`“ am Anfang einer Zeile und endete mit „`#%END-VARIANT-SECTION`“ bzw. „`#%END-RULE-SECTION`“. Hinter dem „`#%`“ konnten optional noch Leerzeichen und Tabulatoren stehen.

Diese Syntax wird vom neuen Shape-Parser zwar (aus Kompatibilitätsgründen) noch unterstützt, wird aber nicht mehr empfohlen und soll bei Gelegenheit aussterben: Es hat sich einfach als zu umständlich („clumsy“) erwiesen, immer eine Section zu eröffnen, um Variantendefinitionen oder eine Versionsauswahlregeln hinzuschreiben.

Ursprünglich war es so vorgesehen, daß Variantendefinitionen und Versionsauswahlregeln an jeweils einer Stelle zusammengefaßt werden. Es hat sich aber herausgestellt, daß es im praktischen Umgang mit Shapefiles wünschenswert ist, beides im lokalen Zusammenhang mit anderen Inhalten des Shapefiles zu schreiben. Dafür ist die neue Syntax mit den Token „:+“ und „:-“ erheblich eleganter. Das Token „:-“ ist bewußt an die Schreibweise von Regeln in Prolog [ClocksinMellish] angelehnt, da die Versionsauswahlregeln gewisse Ähnlichkeiten mit Prolog-Regeln haben. [vbind]

4.1. Grammatik

Bisher war die Syntax von Shapefiles nicht genau spezifiziert; es wußte auch niemand so genau, wie sie aussehen sollte. Bei der Implementierung des Parsers mußte ich deshalb teilweise mit „try and error“ vorgehen. Das sah häufig so aus, daß irgendwelche Probleme auftraten, anhand derer dann diskutiert wurde, wie die Syntax dann eigentlich festgeschrieben werden sollte. Ich bin jetzt in der Lage, eine Grammatik für Shapefiles anzugeben, wie sie dem Parser zugrunde liegt.

In dieser Grammatik benutze ich die folgende Notation:

a	:	a genau einmal
{a}	:	a genau einmal
[a]	:	a null- oder einmal
{a}*	:	a nullmal oder öfter
{a}+	:	a einmal oder öfter
alb	:	einmal a oder einmal b

Alle Wörter, die nicht in Anführungszeichen (") angegeben sind, sind Nichtterminalsymbole. Terminalsymbole sind in Anführungszeichen als reguläre Ausdrücke (egrep-Syntax) angegeben. [egrep]

```
Shapefile       ::=     { {     deprule
                          |     blank* macrodef
                          |     vclassdef
                          |     selrule
                          |     vardef
                          |     rulesection
                          |     varsection
                          |     include
                          |     blank*   }   [comment] newline }*
```

Ein Shapefile besteht aus folgenden Elementen: Abhängigkeitsregel, Makrodefinition, Vclass-Definition, Auswahlregel, Variantendefinition, Rule-Section, Variant-Section, Include-Direktive, oder eine Zeile mit null oder mehreren Leerzeichen. Vor dem Zeilenende kann ein Kommentar stehen.

```
deprule        ::=   dependency  [ whitespace* semicolon deprulebody ] newline
                       {tab deprulebody newline}*
```

Eine Abhängigkeitsregel enthält die Abhängigkeitsdefinition und in den folgenden Zeilen den Rumpf der Regel. Dieser (oder ein Teil davon) kann auch durch Semikolon getrennt hinter der Abhängigkeitsdefinition stehen.

```
dependency     ::=   blank* name {whitespace+ name}* whitespace*
                       ":" whitespace* [name] {whitespace+ name}*
                       [ ":" whitespace* [name] {whitespace+ name}* ]
```

Die Abhängigkeitsdefinition enthält vor dem ersten Semikolon die Targets, dahinter die Dependents. Hinter denen können hinter einem Doppelpunkt noch die Namen der Makros stehen, von denen die Targets abhängen.

```
deprulebody    ::=   "[^\n]*"
```

Der Rumpf der Regel enthält die Transformationsvorschrift, wie sie an die Shell weitergegeben wird. Dieser Teil kann beliebig sein, der Parser gibt ihn literal weiter.

```
macrodef       ::=        { name whitespace*  "=" [string] [comment] newline }
                    |     { name whitespace* "+=" [string] [comment] newline }
                    |     { name whitespace* "-=" [string] [comment] newline }
                    |     { name whitespace* ":=" [string] [comment] newline }
```

Eine Makrodefinition besteht aus dem Makronamen, dem Zuweisungsoperator, und einem beliebigen String dahinter.

```
vclassdef      ::=   blank* "vclass" whitespace+ name whitespace*
                       ":::=" whitespace* "(" whitespace* name
                       { "," whitespace* name whitespace* }* ")"
                       whitespace* newline
```

Vclass-Definitionen enthalten das Schlüsselwort „`vclass`“, dem Namen der Vclass, und hinter dem Token „`:::=`“ die Liste der Variantennamen in Klammern.

```
selrule        ::=   blank* name whitespace* ":-" whitespace* newline
                       {tab whitespace* rulebody [comment] newline}+
```

Eine Auswahlregel beginnt mit dem Namen der Regel und dem speziellen Token „`:-`“. Auf den folgenden Zeilen steht, um einen Tabulator eingerückt, der Rumpf der Regel.

```
rulebody       ::=   {bodystring | "(" rulebody ")"}+

bodystring     ::=   "[^#()\n]*"
```

Der Rumpf der Regel wird vom Parser fast ignoriert; er achtet nur darauf, daß öffnende runde Klammern wieder geschlossen werden. (Das Parsieren dieses Rumpfs passiert in der vbind-Komponente von Shape. [vbind])

```
vardef         ::=   blank* name whitespace* ":+" whitespace* newline
                       {tab macrodef newline}+
```

Eine Variantendefinition sieht ähnlich aus wie eine Auswahlregel, nur daß sich das spezielle Token unterscheidet und daß im Rumpf Makrodefinitionen stehen.

```
rulesection    ::=   blank* "#%" whitespace* "RULE-SECTION" whitespace* newline
                       { oldselrule | {blank* [comment] newline}}*
                       blank * "#%" whitespace* "END-RULE-SECTION" whitespace* newline
```

Eine Rule-Section wird geklammert durch ihre Anfangs- und Endzeilen und enthält Auswahlregeln nach der alten Syntax.

```
oldselruleordef  ::=  blank* name whitespace* ":" whitespace* newline
                        {whitespace* rulebody [comment] newline}+
```

Die Auswahlregeln nach der alten Syntax unterscheiden sich von den anderen Auswahlregeln dadurch, daß die Zeile mit dem Namen nur durch einen Doppelpunkt beendet wird und daß der Rumpf der Regel nicht mit einem Tabulator eingerückt werden muß.

```
varsection       ::=  blank* "#%" whitespace* "VARIANT-SECTION" whitespace* newline
                        { oldvardef | {blank* [comment] newline}}*
                        blank* "#%" whitespace* "END-VARIANT-SECTION" whitespace*
                        newline
```

Eine Variant-Section sieht entsprechend zur Rule-Section aus und enthält Variantendefinitionen nach der alten Syntax.

```
oldvardef        ::=  blank* name whitespace* ":" whitespace* newline
                        {whitespace* macrodef newline}+
```

Diese unterscheiden sich von ihren „modernen“ Pendants genauso wie die Auswahlregeln nach der alten Syntax.

```
include          ::=  blank* "include" {whitespace+ name}+
                        whitespace* newline
```

Eine Include-Direktive enthält das Schlüsselwort „`include`“ und dahinter die Namen der zu inkludierenden Dateien. Diese Namen können durch ein oder mehrere Makroreferenzen angegeben werden. Falls die Makros nicht zu einem oder mehreren Namen expandieren, gibt der Parser ein Warnung aus.

```
comment          ::=  "#[^\n]*"
```

Ein Kommentar beginnt mit „`#`“ und enthält kein Zeilenende (denn damit würde er beendet).

```
string           ::=  {"[^#\n\"']+" | quotedstring}+

quotedstring     ::=  "\"[^\"\n]*\"" | "'[^'\n]*'"
```

In Shapefiles haben einfache und doppelte Anführungszeichen („`'`“ und „`"`“) eine Sonderbedeutung, eben als Anführungszeichen. Damit diese Zeichen selbst auch literal benutzt werden können, haben einfache Anführungszeichen innerhalb von doppelten und doppelte innerhalb von einfachen Anführungszeichen keine Sonderbedeutung.

```
name             ::=  {"[^#:=; \t\n]" | quotedstring}+
```

Ein Name darf keine Zeichen enthalten, die für Shape eine Sonderbedeutung haben, es sei denn, innerhalb von Anführungszeichen.

Noch ein paar einfache Zeichendefinitionen:

```
blank            ::=  " "

newline          ::=  "\n"

semicolon        ::=  ";"

tab              ::=  "\t"

whitespace       ::=  "[ \t]"
```

4.2. Bedeutung syntaktischer Elemente in Shapefiles

‘ Im Shell-Skript wird das von Shape nicht gesehen, also direkt an die Shell weitergegeben.

In einem Kontext, der von Shape interpretiert wird, werden mit ‘ umgebene Strings wie mit " oder ’ gequotete String gelesen. Später werden sie dann (in `expandmacro()`) mit Shell-Substitution „behandelt“; der String wird als Shell-Kommando ausgeführt und durch die Ausgabe dieses Kommandos ersetzt. Konstrukte wie

```
OBJS = ‘echo $(CFILES) | sed s/.c/.o/g‘
```

sind damit möglich. Shape hat aber inzwischen auch String-Substitution in Makros wie Sun Make, womit das angegebene Beispiel mit

```
OBJS = $(CFILES:.c=.o)
```

einfacher zu behandeln ist.

= Makrodefinition der „herkömmlichen“ Art. Diese Definition erlaubt keine Referenz der Variablen auf sich selbst, d.h. Ausdrücke der Art

```
FOO = $(FOO) BAR
```

sind nicht möglich. Make und Shape überprüfen, ob eine Definition rekursiv ist und brechen in diesem Fall ab. Ein durch so einen Ausdruck definiertes Makro wird erst direkt bei seiner Verwendung in einem Skript expandiert.

:= Makrodefinition in der Semantik einer Variablenzuweisung. Hier sind Ausdrücke der Art

```
FOO := $(FOO) BAR
```

möglich (und „erwünscht“). Die Expandierung des Makros findet in dem Moment statt, wo die Zuweisung „gelesen wird“ (wobei noch nicht geklärt ist, an welcher Stelle das passieren soll), und nicht erst bei der Referenzierung des Makros.

Diese Semantik entspricht im Gegensatz zu „=“ der Semantik von Variablen in imperativen Programmiersprachen.

Diese Erweiterung der Syntax ist zur Zeit noch nicht realisiert, wird aber vielleicht in einer späteren Ausbaustufe folgen. Das entsprechende Konstrukt wird zur Zeit vom Parser zwar erkannt, aber als noch nicht implementiert abgelehnt.

+= Erweiterung einer Makrodefinition. Zur Zeit wird einfach der Text hinter dem „+=“ an das Makro angefügt. Es ist aber zu überlegen, ob hier eine Mengensemantik eingebaut werden sollte, bei der ein String oder Token, der/das schon im Wert des Makros enthalten ist, nicht mehr angefügt wird. (Der Unix-Linker meldet den Fehler „multiply defined symbols“, wenn die gleiche Objekt-Datei mehrfach angegeben wird.) In diesem Fall hätte auch die Form „−=“ einen Sinn.

Dagegen spricht, daß die Absicht des Benutzers nicht allzusehr vorweggenommen werden sollte, um Einengungen zu vermeiden. Ein „+=“ im Sinne der Konkatenierung von Strings hat auch ihren Sinn, ein dazu passendes „−=“ wird dann schwierig.

Zur Zeit wird das „+=“-Token als reine String-Konkatenation verstanden.

: Trenner in Abhängigkeitsregeln. Vor dem „:“ steht die Liste der Targets, dahinter die Liste der Dependents. Ein optionales zweites „:“ in einer Abhängigkeitsregel trennt die Dependents von den dahinter stehenden Makronamen, von deren Werten das Target ebenfalls abhängt.

In der „alten“ Syntax, die wir aus Kompatibilitätsgründen noch weiterhin unterstützen, trennt der Doppelpunkt auch in einer Rule-Section den Kopf einer Regel vom Rest (siehe „`:-`“) und in einer Variant-Section den Kopf einer Variantendefinition vom Rest, den Makrodefinitionen (siehe „`:+`“) — siehe auch „`#%`“

Jedes Target darf nur in einer Regel auftreten, die ein Shell-Skript enthält, dagegen in mehreren, die nur Abhängigkeiten beschreiben.

In den Zeilen nach der Regel kann die Aktion stehen, mit der das Target auf den aktuellen Stand gebracht wird. Diese Zeilen sind mit einem Tabulator eingerückt. Üblicherweise habe sie die Form eines Shell-Skripts (siehe auch „`#!`“).

`::` „`::`“ für mehrere Regeln zu einem Target wird von Shape (im Gegensatz zu Make) nicht unterstützt. Shape gibt hier eine Warnung aus und behandelt das Token wie einen einfachen Doppelpunkt. In den meisten Fällen (wo tatsächlich mehrere Regeln existieren) wird das zu einem fatalen Fehler führen, wo die zweite Regel zu diesem Target „entdeckt“ wird, da ein Target nur in einer Regel auftauchen darf. (Siehe „`:`“)

Das nimmt Shape aber keine Möglichkeiten: Dinge, die man mit „`::`“ ausdrücken kann, lassen sich mit Variantendefinitionen flexibler definieren.

`:-` Trenner in Versionsauswahl-Regeln. Vor dem „`:-`“ steht der Name der Auswahlregel, dahinter der Inhalt der Regel. Der Inhalt der Regel steht in einer oder mehreren neuen Zeilen und ist mit einem Tabulator eingerückt.

`:+` Trenner in Variantendefinitionen. Vor dem „`:+`“ steht der Name der Variante, dahinter die zur Variante gehörigen Makrodefinitionen. Die Makrodefinitionen stehen in den folgenden Zeilen und sind mit einem Tabulator eingerückt.

`$` Leitet eine Makroreferenz ein. Makroreferenzen sind von der Form „`$x`“ oder „`$(token)`“ oder „`${token}`“, wobei „`x`“ bzw. „`token`“ der Makroname ist. Ist der Makroname dabei nicht geklammert, wird nur ein Zeichen als Makroname angenommen. „`$token`“ ist also eine Referenz auf das Makro „`t`“ gefolgt von dem String „`oken`“.

Braucht man (z.B. in einem Shell-Skript) ein literales „`$`“-Zeichen, schreibt man „`$$`“. Das kann verstanden werden als eine Referenz auf das Builtin-Makro „`$`“, das den Inhalt „`$`“ hat.

`#` Leitet einen Kommentar ein, wenn es nicht am Anfang der Zeile steht und nicht von einem Zeichen mit Sonderbedeutung gefolgt wird (siehe „`#%`“ und „`#!`“). Der Kommentar endet am Zeilenende. In einem Shell-Skript wird das „`#`“ nicht interpretiert.

`#%` Am Anfang der Zeile: Leitet die alte Syntax von Variant-Section und Rule-Section ein. Hinter dem „`#%`“ steht dann „`VARIANT-SECTION`“, „`END-VARIANT-SECTION`“, „`RULE-SECTION`“ oder „`END-RULE-SECTION`“. Zwischen diesen „Klammern“ steht dann die eigentliche Variant-Section bzw. Rule-Section. Innerhalb einer Variant-Section bzw. Rule-Section gilt dann die alte Syntax für Auswahlregeln und Variantendefinitionen, bei denen der Kopf der Regel bzw. Variante nur durch einen „`:`“ vom Rest getrennt wird.

Wie bereits erwähnt wird diese Syntax nur noch aus Kompatibilitätsgründen unterstützt.

`#!` Am Anfang der ersten Zeile (hinter einem Tabulator) des Shell-Skripts: Kommandoprozessor für die folgenden Zeilen festlegen. Ähnlich wie in Unix-Shell-Skripts werden die folgenden Zeilen durch den hier angegebenen Kommandoprozessor interpretiert. Im Gegensatz zu Unix-Shell-Skripts braucht allerdings nicht der volle Pfadname des Kommandoprozessors angegeben zu werden, er wird im Kommando-Suchpfad gesucht. Der Kommandoprozessor erhält das angegebene Skript (die auf die mit `#!` beginnende Zeile folgenden Zeilen) als Standardeingabe.

Wie bei „normalen“ Aktionen sind alle Zeilen des Skripts (inklusive der ersten mit „`#!`“ beginnenden) um einen Tabulator eingerückt.

`vclass`, `::==`
Das Schlüsselwort `vclass` und das Token `::==` werden in Vclass-Definitionen benutzt.

`include`
Das Schlüsselwort `include` wird in einer Include-Direktive benutzt. Hinter dem Schlüsselwort

steht eine Liste von Dateinamen oder ein oder mehrere Makros, die zu einem oder mehreren Dateinamen expandieren. Diese Dateien werden so gelesen, als ob der Inhalt der Dateien in der Reihenfolge ihrer Nennung anstelle der Include-Direktive im Shapefile stünde.

4.3. Quoting

Die untersuchten Make-Derivate behandeln Quotes alle gleich: Leerzeichen innerhalb der Quotes sind signifikant, Makros werden darin aber trotzdem expandiert. Ist eine gequotete Regel in einem Makro enthalten, verhalten sich alle Makes etwas merkwürdig: wenn das Makro da referenziert wird, wo eine Regel stehen kann, wird es zwar als Regel erkannt, aber sie können das Target nicht erzeugen, obwohl es dafür eine Regel gibt. Läßt man die Quotes weg, ist alles in Ordnung.

Shape verhält sich da einfacher: Syntaktische Strukturen werden nicht erkannt, wenn sie als Inhalt eines Makros auftauchen. Andernfalls müßte der Parser sämtliche Makroreferenzen expandieren, um festzustellen, ob sich im Inhalt syntaktische Strukturen verbergen. Da diese Möglichkeit selten verwendet wird und kein wichtiges Ausdruckmittel darstellt, wurde diese Restriktion zur Vereinfachung des Parsers eingeführt.

Gequotete Strings werden wie Namen (Namen von Targets, Regel-, Vclass-, oder Variantennamen, Dateinamen) behandelt. Deshalb dürfen alle Arten von Zeichen in einem solchen String vorkommen — außer dem Zeichen, das Quoting eingeleitet hat. Es können aber zwei gequotete Strings direkt nebeneinander geschrieben werden; sie werden dann als Einheit betrachtet. Damit ist es möglich, Strings zu schreiben, die sowohl „"“ als auch „’“ enthalten:

```
"hier ist ’ enthalten "’und hier " ’
```

wird nach dem Entfernen der Quotes zu

```
hier ist ’ enthalten und hier "
```

Das ist vermutlich auch ein vergleichsweise unwichtiger Punkt, da syntaktische Elemente in Makros nur extrem selten verwendet werden.

4.4. Namenskonflikte, Geltungsbereiche

Bei Makrodefinitionen ergibt sich das Problem, daß andere Makros versehentlich umdefiniert werden können. Das kann dann leicht passieren, wenn ein Shapefile-System wie zum Beispiel das ShapeTools Release Management System viele Dateien inkludiert, die der Benutzer nicht selbst geschrieben hat (etwa aus Templates abgeleitet) oder noch nicht einmal direkt sieht (wie stdconf, stdvar, stdrules, stdtargets).

Wir haben überlegt, ob man lokale Variablen bzw. Makros mit einem begrenzten Geltungsbereich einführen sollte, um dieses Problem zu umgehen. Der Aufwand dafür wäre groß: Es müßten explizite Kontexte in Shapefiles eingerichtet werden, und jede Regel müßte einen Verweis auf ihr gültiges „Environment“ haben, ähnlich wie Environments in Scheme [R^3RS].

Wir haben uns dagegen entschieden, weil

- uns der Aufwand zu groß erschien und
- das Shapefile damit noch komplizierter würde.

Der Benutzer kann das Problem dadurch umgehen, indem er die Namenskonvention von GNU Make übernimmt, bei der „globale“ Namen GROSS geschrieben werden, „lokale“ Namen klein.

5. Makromechanismus

Makros in Make-(und Shape-)files stellen einen einfachen Textersetzungsmechanismus dar: Einem Makronamen wird ein Makrowert zugewiesen. Dieser Makrowert ist ein String. Ein Makro wird referenziert, indem sein Name mit einem vorgestellten Dollarzeichen geschrieben wird (bei Makronamen, die nur einen

Buchstaben lang sind), oder in geschweiften oder runden Klammern mit einem vorgestellten Dollarzeichen.

Die Auswertung (*Expansion*) einer Makroreferenz erfolgt zu dem Zeitpunkt, wo die Referenz benötigt wird, d.h. üblicherweise *nachdem* das ganze Makefile gelesen worden ist. Ein Makro kann deshalb auch innerhalb des Makefiles umdefiniert werden, es gilt dann die textuell letzte Definition. Das kann nützlich sein, wenn andere Dateien inkludiert werden — so ist es möglich, Makrodefinitionen aus der inkludierten Datei zu ändern. Es ist damit *nicht* möglich, in einer Makroreferenz auf einen „alten" Wert des Makros zu verweisen — es gibt keinen alten Wert. Eine Referenz auf das Makro selbst in seinem Wert führt zu einer Endlosrekursion in der Makroexpansion. Dieser Fall wird von Make und Shape abgefangen.

5.1. „+= - Makros"

Ein paar neuere Make-Derivate (Sun Make, Dmake, Shape) verwenden bei Makrodefinitionen zusätzlich zu „=" auch das Symbol „+=" (entlehnt der Sprache C [K&R]). In diesem Fall wird etwas zum Wert des Makros „addiert", bzw. der neue Wert wird an den alten Wert angehängt.

Dazu gab es für die Implementierung des neuen Shape die Überlegung, daß die Behandlung von Makros als Mengen von Namen sinnvoll sein könnte. Ein Beispiel: Die Makroform „+=" wird häufig dazu benutzt, eine bestimmte Objektdatei zusätzlich mit einzubinden, z.B. eine Library dazuzulinken, die bestimmte Debugging-Funktionen enthält („`electric_fence.o`", „`libdbmalloc.a`"). Genauso sinnvoll kann es aber auch sein, ein bestimmtes Modul auszutauschen gegen ein anderes: `bla.o` gegen `bla-debug.o` oder ähnliches. Dazu bräuchte man zusätzlich zum „+=" auch ein „-=", das ein anderes Element aus dem Wert des Makros entfernt.

Dieser Gedanke wurde wieder fallengelassen. Die bisherige Semantik des „+=" in Shape ist eine String-Konkatenation. Dies ist *auch* eine sinvolle Semantik — eine Änderung würde möglicherweise Dinge unmöglich machen, die darauf beruhen. Das obengenannte Austauschen eines Moduls läßt sich auch mit dem bisherigen Makromechanismus über Varianten erledigen.

5.2. „Variablen"-Makros in GNU-Make

In GNU-Make gibt es als Variation davon Makros mit einer Semantik, die Variablen in imperativen Programmiersprachen entspricht — der zugewiesene Wert wird schon bei der Makrodefinition expandiert. Damit ist es möglich, daß bei einer Redefinition des Makros der vorige Wert des Makros referenziert wird.

Diese Möglichkeit gibt es in Shape noch nicht. Wie schon beschrieben ist der Parser schon darauf eingerichtet, ein solches Konstrukt zu erkennen, lehnt es aber als noch nicht implementiert ab.

5.3. String-Substitution

Von Sun Make wurde die Möglichkeit von String-Substitution in Makros übernommen. Dabei können in einer Makro-Referenz Strings innerhalb des Makrowerts durch einen anderen String ersetzt werden. Eine übliche Form der Anwendung ist die Generierung einer Liste von Objektdateien aus einer Liste von Quelldateien:

```
OBJECTS = $(SOURCES:.c=.o)
```

Die String-Substitution wird allerdings nicht vom Parser vorgenommen, sondern von der Makroexpansion.

5.4. Tests und Vergleiche: Shape, Sun-Make, GNU-Make, BSD Make

Da einige Aspekte der Syntax am Anfang noch unklar waren, habe ich Sun Make, BSD Make, GNU Make und das alte Shape mit diversen Testfällen untersucht, um eine Vorstellung davon zu bekommen, wie sich der neue Parser verhalten soll. Für das neue Shape habe ich dann ebenfalls untersucht, wie es sich in diesen Fällen verhält.

```
 foo = bar bar = ronz foo = $(bar) $(foo)

target:
    echo $(foo)
```

Hier erkennen alle Programme, daß die Definition des Makros `foo` in eine Endlosrekursion führt.

```
FOO = blubb
FOO += bar

default:
    @echo $(FOO)
```

Sun Make, das alte und das neue Shape können mit der Makro-„Addition“ richtig umgehen. BSD Make und GNU Make verhalten sich merkwürdig: BSD Make gibt `bar` aus, GNU Make `blubb`.

```
FOO := blubb
FOO += bar

default:
    @echo $(FOO)
```

Sun Make, BSD Make und das alte Shape geraten durch die +=-Form durcheinander. GNU Make versteht das zwar, scheint aber das += nicht mehr zu sehen. Das neue Shape erkennt das :=, bricht aber ab mit der Meldung, daß diese Form noch nicht unterstützt wird.

```
blubber = ‘cat shapefiles‘

all: y x

x: $(blubber)
    @echo ’$(blubber)’

y:
    @echo ’$(blubber)’
    @echo $(blubber)
```

Alle Make-Programme können das „Backquote“-Konstrukt in der Abhängigkeitszeile nicht verarbeiten, nur das alte und das neue Shape arbeiten hier so, wie man es sich wünscht.

```
blubber = laber#fasel

target:
    echo $(blubber)

# alle: laber
```

Der Kommentar im Wort wird von allen Programmen richtig erkannt. Das hatte ich nicht unbedingt erwartet, da ein Kommentarzeichen von der Bourne-Shell nur dann als solches erkannt wird, wenn nichts

anderes direkt davor steht.

```
0123 = laber

target :
    @echo $(0123)

# alle ok
```

Mit dem Makronamen nur aus Ziffern hatte kein Programm Probleme.

```
default : foo

foo:: ronz
    @echo foo älter als ronz
    touch foo

foo:: blubb
    @echo foo älter als blubb
    touch foo

ronz: ; touch ronz

blubb: ; touch blubb

clean:
    -rm ronz blubb foo
```

Die Make-Programme verhalten sich hier alle korrekt. Das alte Shape erzeugt die Targets `ronz` und `blubb` immer neu, kommt aber offensichtlich nie zu der „`@echo ...`“-Anweisung. Das neue Shape warnt, daß die doppelten Doppelpunkte als einfache behandelt werden. Dann stellt es fest, daß für `foo` mehrere Regeln angegeben sind und bricht ab.

```
default : foo

foo : laber
    @echo foo älter als laber
    touch foo

foo:: ronz
    @echo foo älter als ronz
    touch foo

foo:: blubb
    @echo foo älter als blubb
    touch foo

ronz: ; touch ronz
blubb: ; touch blubb
laber: ; touch laber

clean:
    -rm ronz blubb foo laber
```

Alle Make-Programme stellen einen Konflikt bei den Regeln fest. BSD Make macht aber sonderbarerweise trotzdem `foo` aus `laber`. Das alte Shape macht zwar `laber`, `ronz` und `blubb`, hält es aber danach nicht für nötig, `foo` neu zu machen. Das neue Shape verhält sich genauso wie beim vorigen Beispiel.

```
include $(FOO)
```

BSD Make kennt die Include-Direktive nicht. Sun Make und GNU Make melden, daß ihnen ein Target fehlt. Das alte Shape findet das Include-File nicht. Das neue Shape warnt, daß kein Name hinter dem `include` steht, da das Makro `FOO` zu nichts expandiert.

```
blubber : ronz=doof
    touch blubber

ronz=doof :
    touch ronz=doof
```

Sun Make meldet hier eine Makrodefinition an ungültiger Stelle, BSD Make einen Syntaxfehler. GNU Make und das alte Shape wissen nicht, wie sie das Target `ronz=doof` herstellen sollen. Das neue Shape stellt in der ersten Regel einen Syntaxfehler fest, betrachtet die erste Zeile der zweiten als Makrodefinition und die darauf folgende Zeile als inkorrekt, da sie mit einem Tabulator beginnt, obwohl sie nicht zu einer Abhaengigkeitsregel gehoert..

```
blubber : ronz\=doof
    touch blubber

ronz\=doof :
    touch ronz\=doof
```

Auch mit einem Backslash kann das Gleichheitszeichen nicht gerettet werden, alle Programme verhalten sich entsprechend gleich.

```
INCS = fasel laber grunz

include $(INCS) blubber

target:
    @echo CFLAGS = $(CFLAGS)
```

BSD Make kennt keine ”`include`“-Anweisung und kommt daher nicht damit zurecht. Sun Make versteht die Makro-Referenz nicht. GNU Make versteht zwar die Makro-Referenz, betrachtet aber die ganze Zeile als *einen* Dateinamen und scheitert daran. Das alte Shape verhält sich ähnlich, nur daß es den Namen hinter der Makro-Referenz nicht mehr sieht. Das neue Shape löst die Makro-Referenz auf und includiert alle angegebenen Files.

```
d = '      $(blei)'
bla = a b c d
bli = $d e f
blei = eididei
# bli = $(bla) $(bli)

blo:
    @echo $(bli)
    @echo $d
```

Alle Programme verhalten sich hier wunschgemäß, die Leerzeichen im Makro `d` bleiben erhalten.

```
a = blubb blubb
b = a

target:
    @echo $($(b))

# sun: ok
# gnu: ok
# bsd: sh: syntax error at line 1: `)' unexpected
# shape: /bin/sh: syntax error at line 1: `)' unexpected
```

Sun Make und GNU Make sind die einzigen, die einen verschachtelten Makroaufruf auflösen können, alle anderen geben Unsinn an die Shell weiter.

```
'fa""""sel=laber' ': fasel': ; blubb
'=':ronz
```

Das neue Shape ist das einzige Programm, daß dieses Quoting versteht — was zu erwarten war.

```
laber = '    fasel'
blubber = '$(laber)'

target:
    @echo $(blubber)
    @echo $(laber)
```

Alle Programme expandieren ein Makro auch innerhalb von einfachen Anführungszeichen.

```
target:
    @echo command found
```

In der Shellskript-Zeile stehen hier zwei Leerzeichen vor dem Tabulator. Diese Zeile wird von fast allen Programmen abgelehnt — nur das alte Shape gibt sie auf die Standardausgabe aus und behauptet „`'target' is up to date`“, obwohl es `target` nicht gibt.

```
macro = target : dependent

$(macro)
    @echo bla

dependent:
    touch dependent
```

Alle Make-Programme expandieren das Makro `macro` und erkennen die Zeile dann als Abhängigkeitszeile. Das alte Shape gibt die ganze Regel auf die Standardausgabe aus und ignoriert sie. Das neue Shape meldet (korrekterweise), daß es diese Zeile nicht erkannt hat.

```
default : foo

foo: ronz
    @echo foo älter als ronz
    touch foo

foo: blubb
    @echo foo älter als blubb
    touch foo

ronz: ; touch ronz

blubb: ; touch blubb

clean:
    -rm ronz blubb foo
```

Da hier mehrere Regeln für ein Target vorhanden sind, brechen alle Programme mit einer entsprechenden Meldung ab.

```
FOO = $(BLUFOO) $(FASFOO)

#% VARIANT-SECTION

fasel:
FASFOO = fasel
vclass fasblu ::= (fasel, blubber)

blubber:
BLUFOO = bluber
#%END-VARIANT-SECTION

target: +fasel +blubber
    @echo $(FOO)
```

Die Make-Programme können hiermit natürlich nichts anfangen. Das alte Shape versteht erstaunlicherweise die Zeile „`@echo $(FOO)`“ nicht, gibt sie auf die Standardausgabe aus und bricht ohne weitere Meldung ab. Das neue Shape erkennt dagegen den Konflikt der Variantendefinitionen.

Interessant war auch, welche Zeichen in Namen akzepiert werden. Dafür habe ich Makefiles der folgenden Form benutzt:

```
&0123 = laber

target :
    @echo $(&0123)
```

Dies hier für das Beispiel Ampersand (Kaufmanns-Und). Voher hatte ich schon festgestellt, daß keins der Programme Schwierigkeiten mit einem Makronamen hat, der nur aus Ziffern besteht.

Die Programme verhalten sich sehr unterschiedlich:

Zeichen	Sun Make	BSD Make	GNU Make	altes Shape	neues Shape
&	–	–	+	*	+
\	*	–	+	*	+
!	*	–	+	*	+
|	–	–	+	*	+
{	–	–	+	*	+
,	+	–	+	*	+
>	–	–	+	*	+
<	+	–	+	*	+
(	–	–	+	*	+
)	–	–	*	*	*
%	+	–	+	*	+
+	+	–	+	*	+
?	+	–	+	*	+
;	–	–	+	*	+
*	+	–	+	*	+
~	+	–	+	*	+

In dieser Tabelle bedeutet „+“, daß das Zeichen als Bestandteil des Makronamens akzeptiert wurde und das Programm sich damit so verhält, wie man es erwartet. „−“ bedeutet, daß die Makrodefinition nicht akzeptiert wurde und das Programm mit einer Fehlermeldung abbrach. „*“ bedeutet, daß zwar keine Fehlermeldung ausgegeben wurde, aber das Programm sich nicht korrekt verhalten hat.

BSD Make ist das „kritischste“ Programm von allen, es akzeptiert keine Sonderzeichen in Makronamen.

Sun Make ist etwas weniger kritisch, verhält sich aber in zwei Fällen merkwürdig (das Echo ist leer).

Das alte Shape verhalt sich bei allen Sonderzeichen komisch: Es schreibt die Makrodefinition auf die Standardausgabe (der bekannte Fehler bei nicht erkannten Konstrukten), führt aber die Makrodefinition und die Aktion trotzdem korrekt aus!

GNU Make ist ausgesprochen „tolerant“ bezüglich der Sonderzeichen. Nur bei der schließenden Klammer wird offensichtlich ein Makro mit leerem Namen bei der Makro-Referenz im Shellskript erkannt, dessen Inhalt dann als leer angenommen wird.

Die große Toleranz von GNU Make habe ich dann als Maßstab für den Shape-Parser genommen und alle Zeichen, bei denen das irgend möglich war, in Makronamen (und anderen Namen) zugelassen. Das Phänomen bei der schließenden Klammer tritt beim neuen Shape auch auf; offensichtlich arbeitet die Makro-Expandierung ähnlich wie bei GNU Make.

5.5. Zeilen mit Kommandos bzw. rulebody allgemein

Sun Make, GNU Make, BSD Make und das alte Shape verhalten sich alle verschieden, wenn im Shell-Skript vor dem Tabulator noch Leerzeichen stehen. Shape erzwingt, daß der Rumpf einer Auswahlregel, Variantendefinition (in der neuen Syntax) oder Abhängigkeitsregel mit einem Tabulator anfängt.

Zeilen auf dem Top-Level dürfen dagegen *keinen* Tabulator vor dem ersten „Wort“ enthalten.

6. Der neue Parser für Shape

Bis zu dem Release ShapeTools 1.3 wurde Shape mit dem „alten“ Parser ausgeliefert. Der alte Parser war mit einer Lex-Grammatik implementiert. Der daran angehängte C-Code war betreffend Lesbarkeit und damit Wartbarkeit nicht zufriedenstellend. Die Meldung von Syntaxfehlern war schwach, und aufgrund eines Programmierfehlers wurden viele innerhalb des Parsers entdeckte Fehler gar nicht gemeldet.

Ebenfalls unzufriedenstellend war die Performance des alten Parsers. Ein Teil der verbrauchten Zeit war darauf zurückzuführen, daß auf temporäre Dateien geschrieben wurde, die später wieder eingelesen werden mußten, ein anderer Teil darauf, daß der Lex-generierte Scanner/Parser für diese Grammatik sehr ineffizient war.

Für die Implementierung des neuen Parsers wurden deshalb die folgenden Punkte als besonders wichtig eingeschätzt:

- erhöhte Effizienz,
- verbesserte Syntaxüberprüfung mit entsprechenden Fehlermeldungen,
- bessere Wartbarkeit.

6.1. Schnittstelle zu den anderen Teilen von Shape

Der neue Parser hat eine genau definierte Schnittstelle zu den anderen Teilen von Shape:

6.1.1. Zur Verfügung gestellte Aufrufe

Der Parser wird aufgerufen über die Hauptprozedur, `parse_file()`. Wenn diese Prozedur terminiert, ist die Datei (das kann auch eine AtFS-Version sein), deren Name als Parameter übergeben wurde, komplett bearbeitet.

Die Abhängigkeitsregeln und Versionsauswahlregeln müssen zusätzlich über `commit_deprule_definitions()` und `commit_selrule_definitions()` in die Datenstrukturen von Shape eingetragen werden. (Siehe „queues“)

6.1.2. Benutzte Aufrufe

Der Parser selbst ruft von den anderen Teilen von Shape Prozeduren auf, um

1. Makros zu expandieren (für Namen von Include-Dateien) (`expandmacro()`)
2. die erkannten Daten in die Datenstrukturen von Shape einzutragen. (`ruledef()`, `rulecont()`, `ruleend()`, `macrodef()`, `atBindAddRule()`, `vclassdef()`).

Für das Eintragen der Daten benutzt der neue Parser zur Zeit noch die Schnittstelle des alten Parsers, bei der komplette syntaktische Einheiten als Strings übergeben werden (z.B. eine komplette Makrodefinition). Da der neue Parser die Eingabedaten selbst schon sehr weitgehend zerlegt, ist das eine Redundanz, die in späteren Versionen eliminiert werden soll.

Zum Öffnen von AtFS-Versionen als Dateien benutzt der Parser Funktionen des AtFS-Toolkits und von AtFS (`atBindVersion()`, `af_open()`).

Für String-Manipulationen werden die String-Routinen aus `strbuf.c` benutzt. `strbuf.c` implementiert einen Satz von Funktionen zum Arbeiten mit dynamischen Strings — der Benutzer braucht sich nicht um das Allozieren von Speicherplatz etc. zu kümmern. Allerdings muß bei Operationen, die einen String neu anlegen oder vergrößern (`sbnew`, `sbcat`), der Rückgabewert überprüft werden. Ich benutze diese Prozeduren deswegen über eine „Hülle“, die den Rückgabewert prüft und im Fehlerfall entsprechend reagiert (siehe `check_sbnew()` und `check_sbaddc()`). Im einzelnen benutze ich `sbnew()` (Anlegen eines String-Buffers), `sbcat()` (Anhängen an einen String-Buffer), `sblen()` (gibt die Länge des Inhalts eines String-Buffers zurück), `sbstr()` (gibt den String-Inhalt eines String-Buffers zurück) und `sbfree` (gibt einen String-Buffer wieder frei).

`queue_variant_definition()` (siehe dort) ruft `vardef_end()` auf, um dem „Rest von Shape“ den Abschluß einer Variantendefinition mitzuteilen, damit diese Variante gegebenenfalls sofort aktiviert werden kann.

6.1.3. Globale Variablen

Die Daten von Variantendefinitionen trägt der Parser selbst in die Shape-Datenstrukturen ein, dafür greift er direkt darauf zu.

Der Parser definiert die globale Variable `parse_errors`. Hier wird die Anzahl der gefundenen Parsierungsfehler zurückgegeben; genaugenommen ist das die Anzahl der Aufrufe von `parsing_error()`.

`queue_variant_definition()` benutzt die globale Variable `lastVariantDef`. Hier wird die Nummer der zuletzt definierten Variante hineingeschrieben, die von `vardef_end()` benutzt wird.

Eine ungefähre Gliederung von Shape in Subsysteme mit ihren benutzt-Beziehungen zeigt Abbildung 1. Shape benutzt dabei folgende Libraries von ShapeTools:

- AtFS

  Attributed File System — die Datenhaltungsschicht von ShapeTools, die Versionen und Attribute von Dateien verwaltet.

- AtFStk

  AtFS Toolkit Library — auf dem AtFS aufsetzende häufiger gebrauchte Funtionen

- Sttk

  ShapeTools Toolkit Library — innerhalb von ShapeTools häufiger gebrauchte Funtionen, die nicht auf dem AtFS aufsetzen

Shape selbst gliedert sich in folgende Hauptbestandteile:

- Ablaufsteuerung

  Parsieren der Kommandozeile, Initialisieren globaler Datanstrukturen, Aufruf des Parsers, Aufruf der Produktion

- Parser

  Parsieren der Eingabedateien, Eintragen der erkannten Daten in die Datenstrukturen

- Daten

  Globale Datenstrukturen für Makros, Regeln, Varianten etc.
- Produktion

  Auswerten der Regeln, Überprüfen der Abhängigkeiten, Ausführen der Kommandos

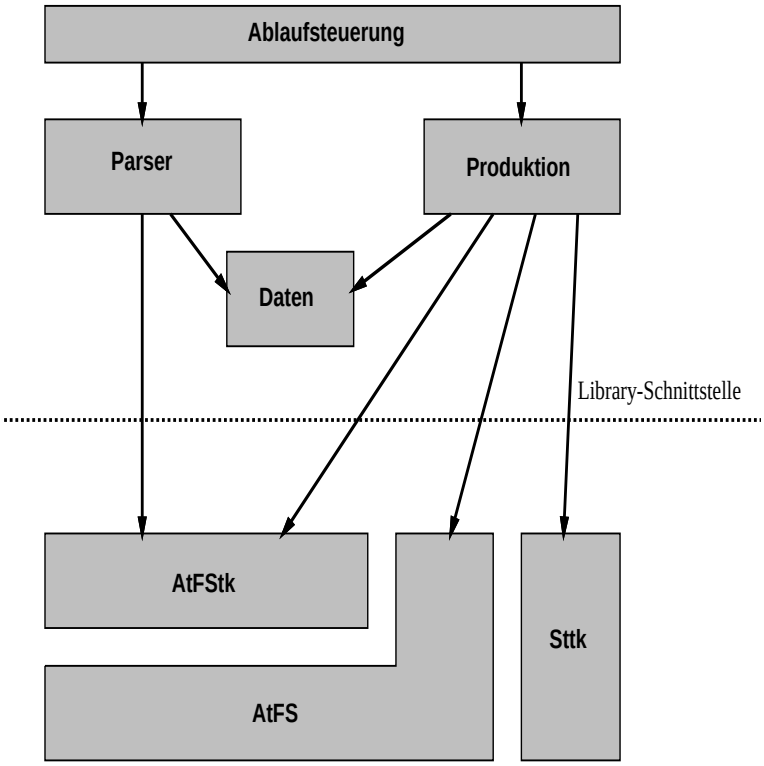

Abbildung 1: Subsysteme von Shape

6.2. Implementierungsdetails

6.2.1. File-IO (`mfiles.c`, `mfiles.h`)

Shape benutzt mehr Informationen als Make, deshalb liest Shape im allgemeinen mehr Dateien als Make. Im ShapeTools Release Management System sind das im Moment sieben Dateien:

- `Shapefile`
- `Makefile`
- `Dependencies`
- `stdconf`
- `stdrules`
- `stdtargets`
- `stdvar`

Die alte Shape-Implementierung arbeitet außerdem noch mit temporären Dateien, in denen unter anderem die Fortsetzungszeilen miteinander verbunden sind. Insgesamt ergibt sich ein rechter hoher IO-Aufwand. An dieser Stelle versuche ich, beim neuen Parser effizienter zu arbeiten.

Da bei modernen Maschinen der Speicher billiger ist als die Platten schnell sind, ist es günstiger, Dateien grundsätzlich komplett in den Speicher zu lesen. Das ist aus zwei Gründen schneller als die alte Methode:

1. Das Lesen an sich ist effizienter, weil weniger Systemaufrufe dafür gebraucht werden. Normale Dateien (im Gegensatz zu z.B. stdin) können mit einem `read`(2) gelesen werden.
2. Da der Inhalt einer Datei komplett im Speicher liegt, braucht die Datei nicht noch mehrfach gelesen zu werden, wenn sie mehrfach durchgegangen wird.

Der Speicher muß dadurch nicht unbedingt knapp werden: Die sieben Dateien des shape_RMS haben zusammen einen Umfang von etwas über 40 KB. Das ist eine Mindermenge in Bezug auf das Maß an Hauptspeicher, den Shape im weiteren Programmlauf benötigt, und damit unerheblich.

Zusammenfügen von Fortsetzungszeilen kann auch komplett im Speicher erfolgen: die Sequenz „Backslash-Newline“ wird einfach durch zwei Leerzeichen ersetzt. Dafür gibt es zwei Möglichkeiten:

- Es kann gleich nach dem Einlesen geschehen. Das ist sehr einfach zu implementieren, hat aber den Nachteil, daß die Datei schon vor dem Parsieren einmal komplett durchgegangen werden muß.
- Bei der Parsierung wird, wenn das Ende der Zeile erreicht ist, die Fortsetzungszeile angefügt. Das ist schneller, da ein Durchgang durch die Datei gespart wird, der Parser wird dadurch aber komplizierter.

Im Laufe der Test habe ich festgestellt, daß der Overhead für den ersten Durchgang durch die Datei zum Zusammenfügen der Fortsetzungszeilen nicht erheblich ist.

Es ergibt sich dabei aber noch ein weiteres Problem, das auch die Interpretation der Shapefiles beeinflußt: Während dieses ersten Durchgangs durch das Shapefile ist noch nicht (mit vertretbarem Aufwand) festzustellen, ob das Ende der Zeile Teil eines Kommentars ist. Dazu müßten während des Durchgangs durch die Zeile Quoting-Zeichen verfolgt werden — aber das ist genau der Aufwand, der an dieser Stelle nicht betrieben werden soll. Folglich müssen Fortsetzungszeilen innerhalb von Kommentaren genauso behandelt werden wie an anderer Stelle. Das heißt: Steht am Ende einer Kommentarzeile ein Backslash, ist die folgende Zeile auch eine Kommentarzeile. Das ist in gewisser Weise kontraintuitiv, aber bei dieser Methode unvermeidbar.

Folgende Überlegung spielte auch eine Rolle: Man könnte sich eventuell das häufige Kopieren von Strings ersparen, wenn alle Tokens, die nicht Syntax sind, d.h. Dateinamen, Namen von Regeln, Makronamen etc., an dieser Speicherstelle bleiben können und im Weiteren über Pointer auf diese Stelle referenziert werden können. Bei der weiteren Implementierung des Parsers stellte sich das zunächst als unpraktikabel heraus — es war in den meisten Fällen einfacher, einen String zu kopieren, als genau zu wissen, ob auf diese Stelle vielleicht noch einmal zugegriffen werden muß. In einer Überarbeitung dieses Parsers könnten allerdings einige Kopiervorgänge eingespart werden, weil jetzt genau zu sehen ist, wo welche Stellen noch referenziert werden müssen.

Die obengenannte Funktionalität ist in dem Modul MFILES realisiert. MFILES liest eine Datei in den Hauptspeicher und verwaltet für den Benutzer sichtbar die Vorstellung einer „aktuellen Zeile“. Eine Zeile in diesem Sinne ist schon mit ihren Fortsetzungzeilen zusammengefaßt, im Gegensatz dazu stehen die „Originalzeilen“, die mit den Zeilennummern in der eingelesenen Datei korrespondieren.

Das Modul MFILES bietet dem Benutzer folgende Funktionen:

- `mf_readin()`, `mf_readin_stdin()`: eine Datei oder die Standardeingabe in den Speicher lesen (die aktuelle Zeile ist dann vor der ersten Zeile gedacht)
- `mf_nextline()`: einen Zeiger auf die der aktuellen folgende Zeile bekommen, aktuelle Zeile wird diese (als Zeile gilt hier die Zeile inklusive der Fortsetzungszeilen)
- `mf_setline()`: aktuelle Zeile setzen auf die, in die ein Zeiger zeigt
- `mf_thisline()`: aktuelle Zeile anhand einer Zeilennummer setzen
- `mf_locate()`: zu einem Zeiger in einen Datei-Puffer den Datei-Namen und die Nummer der Originalzeile zurückgeben (vor allem für die Meldung von Syntaxfehlern nützlich)
- `mf_kilbuf()`: einen Datei-Puffer löschen
- `mf_killall()`: alle Datei-Puffer löschen

Das Module MFILES verwaltet für jede eingelesene Datei eine Struktur der folgenden Form:

```
typedef struct MFILE__ {
    char           *mf_name ;   /* filename */
    char           *mf_buffer ; /* pointer to mfile buffer */
    char          **mf_lbeg ;   /* array of original line beginnings */
    int             mf_lines ;  /* number of lines */
    int             mf_curline ;/* current line */
    char           *mf_end ;    /* end of buffer */
    struct MFILE__ *mf_next ;   /* pointer to next, don't use! */
} MFILE ;
```

Die einzelnen Felder haben folgende Bedeutung:

`mf_name`
: der komplette Names der eingelesenen Datei, so wie er bei mf_readin() angegeben wird.

`mf_buffer`
: Zeiger auf den Speicherbereich, in den der Inhalt der Datei eingelesen wird.

`mf_lbeg`
: Zeiger auf die Anfänge der Originalzeilen. Anhand dieser Zeiger wird später die Nummer der Originalzeile einer Textstelle rekonstruiert.

`mf_lines`
: Anzahl der eingelesenen Originalzeilen.

`mf_curline`
: Nummer der aktuellen Originalzeile.

`mf_end`
: Zeiger auf das erste Zeichen *hinter* dem eingelesenen Inhalt der Datei.

`mf_next`
: Zeiger auf die nächste Struktur.

Die MFILE-Strukturen werden in eine zu diesem Modul lokale Liste eingehängt (`mf_structlist`, über `mf_next`), in der sie bei `mf_locate()` und `mf_kilbuf()` gesucht werden. Da die Anzahl der eingelesenen Dateien im allgemeinen gering sein wird und diese Operationen relativ selten erfolgen (`mf_kilbuf()` wird zur Zeit überhaupt nicht benutzt), kann an dieser Stelle darauf verzichtet werden, einen effizienteren Algorithmus als das lineare Suchen in dieser Liste zu implementieren.

```
void mf_kilbuf(MFILE *mf) ;
```

Die angegebene MFILE-Struktur wird in `mf_structlist` gesucht (s.o.) und aus dieser Liste ausgehängt. Die belegten Resourcen werden freigegeben.

```
void mf_killall(void) ;
```

mf_structlist wird linear durchgegangen, dabei werden die belegten Resourcen aller MFILE-Strukturen freigegeben.

```
MFILE *mf_readin(char *fname) ;
```

Zunächst wird die Datei mit dem angegebenen Namen zum Lesen geöffnet (`mf_open()`). Da das auch eine Version aus einem AtFS-Archiv sein kann, wird erst mit `atBindVersion()` [vbind] ein Key für die Version gesucht. Über diesen Key wird dann mit af_open() die Datei geöffnet. Da `af_open()` einen `FILE` Pointer zurückgibt, aber `mf_readin_nosplit()` einen Unix File Handle braucht, wird der File Handle zurückgegeben. Der `FILE` Pointer wird aber aufbewahrt, um beim Schließen der Datei die allozierten Resourcen der `stdio`-Library wieder freigeben zu können.

Anschließend wird die Datei in den Speicher gelesen (`mf_do_read()`). Um das möglichst effizient zu tun, wird in möglichst großen Blöcken gelesen — so, daß bei regulären Dateien der gesamte Inhalt mit einem `read`(3)-Systemaufruf gelesen wird. Der dafür benötigte Speicherbereich wird von vornherein in der Größe der Datei alloziert.

Falls die zu lesende Datei keine reguläre Datei ist, ist das nicht möglich, da die Menge der zu lesenden Daten nicht bekannt ist (das betrifft vor allem Pipes). Hier wird zunächst ein Puffer einer bestimmten Größe alloziert (z.Z. festgelegt auf 8 KBytes) und in diesen Puffer gelesen. Ist der Puffer bis zu einem bestimmten Maß voll, wird er vergrößert (`mf_readin_special()`).

In `mf_split_lines()` wird der Inhalt der Datei in die einzelnen Zeilen aufgebrochen: Zeilenvorschübe (newline characters) werden durch Null-Zeichen ersetzt (ASCII 0), es sei denn, ihnen geht ein Backslash (dt. „Rückwärtsschrägstrich“) voraus. In diesem Fall werden beide Zeichen durch Leerzeichen ersetzt — damit sind dann die Fortsetzungszeilen mit der vorausgehenden Zeile verbunden. Zeiger auf die Zeilenanfänge der Originalzeilen werden in `mf_lbeg` geschrieben.

Die Adresse der für diese Datei allozierten MFILE-Struktur wird als Ergebnnis zurückgegeben. Ist in `mf_readin()` eine Fehlerbedingung aufgetreten, wird hier ein `NULL`-Pointer zurückgegeben — das heißt im allgemeinen, daß die Datei bzw. die Version mit dem angegebenen Namen nicht gefunden werden konnte. Eine genauere Diagnose läßt sich in diesem Fall den Variablen `errno` und `af_errno` entnehmen.

```
MFILE *mf_readin_stdin(void) ;
```

`mf_readin_stdin()` liest die Standard-Eingabe in eine MFILE-Struktur. Das hier im allgemeinen Fall nicht bekannt ist, wieviel Daten gelesen werden sollen, wird über `mf_readin_special()` gelesen.

```
char *mf_nextline(MFILE *mf) ;
```

`mf_nextline()` gibt einen Zeiger auf die nächste Zeile zurück. Dabei kann nicht der nächste Zeiger in `mf->mf_lbeg` benutzt werden, sondern es muß der Anfang einer „richtigen“ Zeile sein. Dieser wird daran erkannt, daß das Zeichen davor ein Nullbyte ist, das Ende der vorigen Zeile. Ist es das nicht, handelt es sich um eine Fortsetzungszeile.

```
char *mf_thisline(MFILE *mf, int lineno) ;
```

Zu einer bestimmten Zeilennummer wird ein Zeiger auf den Anfang der entsprechenden Originalzeile zurückgegeben. Diese Funktion ist trivial, es wird lediglich der entsprechende Wert aus dem Array `mf->mf_lbeg` zurückgegeben. Ich habe diese Funktion trotzdem implementiert, um dem Benutzer des MFILES-Moduls den direktem Zugriff auf die Elemente der Struktur `MFILE` zu ersparen.

`mf_thisline()` wurde nur für Debugging des Parsers gebraucht. Mit dem zurückgegebenen Zeiger wurde die Spalte in der Originalzeile berechnet, in der der Parser einen Fehler bemerkt hat.

```
int mf_locate(char *ptr, char **pfilename, int *plineno) ;
```

`mf_locate()` nimmt als Argument einen Zeiger in einen MFILE-Puffer und gibt den Namen der Datei, die in diesen Puffer gelesen wurde, und die Nummer der Originalzeile zurück, in die der Zeiger zeigt. Ein Zeiger auf den Dateinamen wird in dahin geschrieben, wohin `pfilename` zeigt, die Zeilennummer wird an die durch `plineno` bezeichnete Adresse geschrieben.

Die Liste der MFILE-Puffer wird dazu linear durchsucht; in jedem einzelnen MFILE-Puffer wird dann mit `mf_locate_in()` festgestellt, ob der Zeiger `ptr` in diesen Puffer zeigt.

Der Rückgabewert von `mf_locate()` ist `0`, wenn `ptr` nicht in einen der MFILE-Puffer zeigt, ansonsten `1`.

```
int mf_locate_in(MFILE *mf, char *ptr) ;
```

`mf_locate_in()` gibt die Zeilennummer der Zeile zurück, auf die der Zeiger `ptr` im MFILE-Puffer `mf` zeigt. Zeigt der Zeiger nicht in den Puffer von `mf`, gibt `mf_locate_in()` `-1` zurück.

```
void mf_setline(MFILE *mf, char *ptr) ;
```

Mit `mf_setline()` wird die Zeile zur aktuellen Zeile gemacht, in die der Zeiger `ptr` zeigt. Das war ursprünglich dazu gedacht, um an einer vorher schon gelesenen Zeile wieder aufsetzen zu können. Inzwischen wird dazu aber die Funktion `mf_prevline()` benutzt.

```
char *mf_prevline(MFILE *mf) ;
```

`mf_prevline()` gibt einen Zeiger auf die vorige Zeile in `mf` zurück. Diese Funktion wird jetzt statt `mf_setline()` benutzt, um im Parser nach einem Look-Ahead wieder zurückzugehen.

6.2.2. Utilities (utils.c)

Eine Reihe Utility-Funktionen sind ausgelagert in das Modul utils.c. Kandidaten dafür waren

- Funktionen, die einen gewissen Grad von Allgemeinheit aufweisen (Speicherverwaltung, Fehlerbehandlung, etc.), und
- Funktionen, die in verschiedenen Modulen benutzt werden.

6.2.2.1. Fehlerbehandlung

Im Parser kommen außer Parsierungsfehlern (dafür gibt es parsing_warning() und parsing_error() in pa.c.) nur fatale Fehler vor, bei denen das Programm komplett beendet werden muß. Bedingungen dafür sind:

- fehlgeschlagene Betriebssystemaufrufe (mfiles.c)
- fehlgeschlagene AtFS-Aufrufe (mfiles.c)
- Nullbytes im Eingabefile (mfiles.c)
- Fehlgeschlagene Speicherallozierung (utils.c)
- Tabellenüberlauf (queues.c)
- interne Inkonsistenzen (mfiles.c, pa.c)

Dafür gibt es die Funktion

```
void fatal (char *message, char *message2) ;
```

fatal() gibt den String message aus, und falls message2 nicht NULL ist, auch diesen, zusammen mit dem Hinweis, daß ein fataler Fehler aufgetreten ist. Da Shape Zustände persistenter Daten verändert, kann das Programm in so einem Fall nicht einfach abgebrochen werden, sondern muß wohlgeordnet beendet werden, um temporär umbenannte Dateien wieder herzustellen und temporär erzeugte Dateien wieder zu löschen. Dies wird von der Funktion cleanup() erledigt, die von fatal() aufgerufen wird. (cleanup() nicht Bestandteil des Parsers.)

6.2.2.2. Speicherverwaltung

Für eine sorglose Speicherverwaltung sind Allozierungsroutinen wichtig, die *garantiert* den verlangten Speicherbereich zurückgeben — *wenn* sie etwas zurückgeben. Kann der Speicherbereich nicht alloziert werden, wird das Programm über fatal() mit einer entsprechenden Meldung beendet.

Die Allozierungsroutinen habe ich wie die entsprechenden „normalen“ Funktionen benannt mit dem Präfix „check_“ davor, also check_malloc(), check_realloc(), check_strdup(). Zusätzlich gibt es noch check_strndup(), das wie check_strdup() arbeitet, aber noch ein zusätzliches Argument nimmt für die Zahl der zu duplizierenden Zeichen.

Außerdem war es noch sinnvoll, die Funktion „sbnew()“ aus dem Modul strbuf.c entsprechend zu umhüllen (check_sbnew()), sowie ein check_sbaddc() auf der Basis von sbcat() zu bauen, das ein einzelnes Zeichen an einen Stringbuffer anhängt und den Rückgabewert überprüft.

6.2.2.3. String-Listen

Der Parser benutzt recht häufig einfache Listen von Strings. Die Elemente dieser Listen sind vom Typ „struct stringlist“:

```
struct stringlist {
    char *string ;
    int freeable ;
    struct stringlist *next ;
} ;
```

string ist dabei der Zeiger auf den String, der Null-terminiert ist. freeable ist ein Flag, das angibt, ob

der Speicherbereich, auf den `string` zeigt, einzeln mit `free`(3) wieder freigegebenwerden kann. (Diese Unterscheidung ist nötig, da auch Strings in solchen Listen vorkommen, bei denen das Element `string` nicht auf einen einzeln allozierten Bereich zeigt, sondern in einen MFILE-Puffer.) `next` ist der Zeiger auf das nächste Element der Liste und hat beim letzten Listenelement den Wert `NULL`.

Um mit diesen Listen umzugehen, gibt es in `utils.c` ein paar Funktionen:

```
struct stringlist *push_stringlist(char *str, struct stringlist *list,
int freeable)
```

Alloziert eine Struktur und initialisiert die Felder mit den übergebenen Argumenten. Der Rückgabewert ist ein Zeiger auf die allozierte Struktur.

```
struct stringlist *revert_stringlist(struct stringlist *strlp)
```

Dreht eine Stringliste um. Eine Zeiger auf die umgedrehte Liste wird zurückgegeben.

```
void free_stringlist(struct stringlist *strlist)
```

Gibt den von `strlist` belegten Speicherplatz frei. Dabei wird bei den einzelnen Elementen das Feld `freeable` beachtet.

6.2.2.4. Spezialfunktionen für den Parser

In `utils.c` befinden sich einige Funktionen, die für den Parser spezifisch sind. Ein Teil davon sind Funktionen, die über bestimmte Textteile in einer Zeile hinwegspringen. Sie bekommen einen Zeiger in eine Zeile als Argument und liefern einen Zeiger auf das nächste Zeichen hinter dem übersprungenen Teil der Zeile (oder auf das Ende der Zeile) zurück.

- `char *skip_over_blanks(char *p)` springt über Leerzeichen.
- `char *skip_over_whitespace(char *p)` springt über Leerzeichen und Tabulatoren.
- `char *skip_over_name(char *p)` springt über das, was der Parser für einen Namen hält. Siehe `skip_over_name_with_stop()`.
- `char *skip_over_name_with_stop(char *p, char *stop)` tut das ebenfalls, hält aber zusätzlich noch an Zeichen an, die in dem String `stop` enthalten sind.

Ein Name wird dabei folgendermaßen verstanden:

Zu einem Namen gehören alle Zeichen, für die die Funktion „`is_no_namecomponent()`“ Null zurückgibt. Diese Funktion gibt wahr zurück für „`#`“, „`:`“, „`;`“, „`=`“ und „ “ (Leerzeichen), die als Namensbegrenzer in Shapefiles auftreten können.

Außerdem gehören auch andere Zeichen zu einem Namen, wenn sie in einfache oder doppelte Anführungszeichen eingeschlossen sind. Dabei sind einfache Anführungszeichen nicht signifikant, wenn sie in doppelte Anführungszeichen eingeschlossen sind und umgekehrt.

Falls im String `stop` eine runde Klammer enthalten ist, werden runde Klammern gesondert behandelt: Zeichen innerhalb des äußersten Klammernpaares werden auf jeden Fall als zum Namen gehörig betrachtet, Anführungszeichen werden hier ignoriert.

Vom Parser recht häufig gebraucht wird `get_name()`:

```
char *get_name(char **linep)
```

`get_name()` liefert den nächsten Namen auf der Zeile zurück. Dabei wird der Zeiger auf die Zeile, dessen Adresse in `linep` übergeben wird, hinter den Namen gesetzt. `get_name()` setzt zunächst mit `skip_over_whitespace()` einen Zeiger hinter führende Leerzeichen oder Tabulatoren. Dann wird mit `skip_over_name()` ein anderer Zeiger hinter einen (potentiellen) Namen gesetzt. Sind die beiden Zeiger gleich, gab es also nichts, was wie ein Name aussieht, und es wird ein Fehler gemeldet (`parsing_error()`). Andernfalls wird der dazwischenliegende Text kopiert (`check_strndup()`), und nach Entfernung von Quotes (`unquote()`) zurückgegeben.

```
char *unquote(char *string)
```

entfernt einfache und doppelte Anführungszeichen von einem String; das wird vor allem für die Namen benutzt. Dabei gehe ich von der Idee aus, daß es keine Escape-Zeichen o. ä. gibt, sondern daß einfache

Anführungszeichen mit doppelten Anführungszeichen gequotet werden und umgekehrt. Die Anführungszeichen werden „in Place“ entfernt, d.h. der String bleibt an seinem Platz (er wird nur potentiell kürzer).

6.2.3. Der Parser (`parser.h`, `pa.c`)

In dem Modul `pa.c` befindet sich der eigentliche Shapefile-Parser. Die Struktur des Parsers ist der zu parsierenden Grammatik entsprechend hierarchisch:

Die Prozedur `parse_file()` bearbeitet eine komplette Datei. Auf dem Top-Level der Datei werden die einzelnen syntaktischen Elemente erkannt; die Kontrolle wird dann an die entsprechende Prozedur übergeben, die dieses Element bearbeitet.

6.2.3.1. Top-Level

```
int parse_file(char *filename)
```

bekommt den Namen der zu parsierenden Datei (das kann auch eine AtFS-Version sein) als Argument übergeben. Der Rückgabewert ist `FALSE`, wenn die Datei nicht gelesen werden konnte, also `mf_readin()` fehlschlug, andernfalls `TRUE`. Zusätzlich wird die globale Variable `parse_errors` auf die Anzahl der gefundenen Syntaxfehler gesetzt.

`parse_file()` liest zunächst die Eingabedatei ein mit `mf_readin()`. Der von `mf_readin()` zurückgegebene `MFILE` Pointer wird in einer zu dem Modul `pa.c` globalen Variablen gehalten, damit die anderen Parsierungsfunktionen darauf zugreifen können (`currentmf`).

Mit `setjmp()` wird ein Einsprungspunkt für `longjmp()` vor den Anfang der Hauptschleife gesetzt. Dieser Einsprungspunkt (`jmp_toplevel`) dient dazu, um von `parsing_error()` im Toplevel wieder aufsetzen zu können.

Die Hauptschleife liest nun eine Zeile ein und versucht sie auszuwerten:

- Führende Leerzeichen werden übersprungen.
- Das erste Zeichen der verbleibenden Zeile wird angesehen. Hier kann schon festgestellt werden, ob es sich um eine leere Zeile handelt oder eine Zeile, die nur einen Kommentar enthält. Diese Zeilen werden ignoriert. Dabei kann eine Zeile, die mit einem Kommentarzeichen anfängt, auch noch den Anfang einer RULE- oder VARIANT-Section enthalten. Beginnt die Zeile mit einem Tabulator, ist sie ungültig, dafür wird ein Fehler gemeldet. Für eine RULE- oder VARIANT-Section wird `parse_rule_or_var_section()` aufgerufen.
- `recognize_line()` wird aufgerufen, um den Typ der Zeile festzustellen. `recognize_line()` gibt außerdem alle Namen, die gelesen wurden, bevor der Typ der Zeile erkannt wurde, in einer Liste zurück (`name_list`). Konnte `recognize_line()` den Typ der Zeile nicht feststellen, wird ein Fehler gemeldet.
- Kommentare werden mit `clean_from_comment()` aus der Zeile entfernt.
- Abhängig vom Typ der Zeile wird nun die entsprechende Parsierungsfunktion aufgerufen. Diese liest (abhängig vom Typ der Zeile) eventuell weitere Zeilen.

Stellt eine der aufgerufenen Parsierungsfunktionen einen Fehler fest, wird dafür `parsing_error()` aufgerufen. `parsing_error()` meldet den Fehler und springt mit `longjmp()` wieder zur Hauptschleife zurück (`jmp_toplevel`).

```
void parse_rule_or_var_section(char *line) ;
```

Wie erwähnt, wird `parse_rule_or_var_section()` aufgerufen, wenn der Top-Level eine mit „`#%`“ beginnende Zeile gefunden hat. Der Parameter line enthält den Rest der Zeile (hinter dem „`#%`“).

Zunächst wird festgestellt, ob es sich um eine RULE- oder eine VARIANT-Section handelt oder um keins von beiden. Ist es keins von beiden, handelt es sich bei der Zeile um eine Kommentarzeile, die ignoriert wird.

Diese Funktion ist insofern besonders, als bei Syntaxfehlern in den enthaltenen Konstrukten nicht zum Top-Level zurückgesprungen wird, sondern in diese Funktion zurück. Dazu wird der Einsprungspunkt für

`parsing_error()` temporär umgesetzt.

Abhängig von der Art der Section („RULE“ oder „VARIANT“) wird die entsprechende Parsierungsfunktion für die enthaltenen Konstrukte aufgesetzt. Diese wird für jede nichtleere Zeile, die nicht mit einem „`#`“ anfängt (das ist dann entweder ein Kommentar oder das Ende der Section), aufgerufen. Diese Parsierungsfunktion ist entweder `rule_section_body()` oder `variant_section_body()`. Diese Funktionen sind sich recht ähnlich, da der Anfang von Versionsauswahlregeln und Variantendefinitionen in der entsprechenden Section gleich ist. Der Unterschied: `variant_section_body()` erkennt auch Vclass-Definitionen und ruft dafür `parse_vclass_definition()` auf. Ansonsten lesen `rule_section_body()` und `variant_section_body()` beide den Namen der Regel bzw. Definition und rufen damit `parse_selection_rule()` bzw. `parse_variant_definition()` auf. Damit können `parse_selection_rule()` und `parse_variant_definition()` sowohl vom Top-Level als auch von `parse_rule_or_var_section()` (hier indirekt) aufgerufen werden. Sie bekommen aber einen Parameter übergeben, der mitteilt, woher sie aufgerufen wurden, da sich Auswahlregeln und Variantendefinitionen im Top-Level von denen in einer Section syntaktisch unterscheiden.

```
static int recognize_line(char **linep, struct stringlist **name_list_p)
```

Eine zentrale Bedeutung für den ganzen Parser hat die Funktion `recognize_line()`, da in ihr „hart codiert“ der größte Teil des Wissens steckt, welche Zeilen in einem Shapefile welche Bedeutung haben. `recognize_line()` muß alle Zeilen auf dem Toplevel eines Shapefiles erkennen außer

- ungültigen Zeilen,
- leeren Zeilen,
- Kommentarzeilen und
- Anfangszeilen von RULE- oder VARIANT-Sections.

`recognize_line()` versucht zuerst, den Anfang einer Zeile als „`vclass`“ oder „`include`“ zu erkennen. Das schließt die Verwendung dieser Namen als Targets oder als Namen von Makros, Versionsauswahlregeln oder Varianten aus, aber das ist zu verschmerzen.

Schlägt das fehl, wird der Inhalt der Zeile zeichenweise untersucht. Hier kann nicht mit `skip_over_name()` oder `get_name()` gearbeitet werden, weil diese Funktionen die Zeichen „`-`“ und „`+`“ als „Namecomponents“ behandeln. Das darf hier aber nicht passieren, weil diese Zeichen eine Makrodefinition mit „`-=`“ bzw. „`+=`“ einleiten können.

Zeichen, die zu einem Namen gehören können, werden an den String-Buffer `sb` angehängt (`check_sbaddc()`). Kommt ein Zeichen vor, das nicht zu einem Namen gehören kann, wird dieser String-Buffer in die Liste `name_list` gepackt (`push_stringlist()`).

Leerzeichen und Tabulatoren werden übersprungen. Kommt ein Quoting-Zeichen vor („`"`“ oder „`'`“), werden bis zum nächsten Auftreten dieses Zeichens *alle* Zeichen an den String-Buffer `sb` angehängt.

Beim Auftreten von „`-`“ oder „`+`“ wird das darauf folgende Zeichen angesehen. Ist es ein „`=`“, handelt es sich bei der Zeile um eine Makrodefinition.

Wird ein „`:`“ gelesen, kann es sich um eine Makrodefinition handeln („`:=`“), eine Varianten-Definition („`:+`“), eine Versionsauswahlregel („`:-`“), eine VCLASS-Definition („`::=`“), oder um eine Abhängigkeitsregel, die eventuell auch mit zwei Doppelpunkten angegeben ist. Um diese Fälle zu unterscheiden, werden hier ebenfalls die folgenden Zeichen untersucht.

Ist das erste „interessante“ Zeichen ein „`=`“, handelt es sich um eine normale Makrodefinition.

Ist nun der Typ der Zeile erkannt worden, wird der String-Buffer in die Liste gepackt, falls Zeichen darin enthalten sind. Diese Liste wird dann umgedreht, damit sie die gelesenen Namen in der richtigen Reihenfolge enthält. Die Adresse dieser Liste wird dem Parameter `name_list_p` zugewiesen und damit der aufrufenden Stelle verfügbar gemacht.

Falls Quotes in der Zeile nicht geschlossen worden sind, wird ein entsprechender Fehler gemeldet.

Schließlich wird der erkannte Zeilentyp (`line_type`) als Wert zurückgegeben.

Abhängig vom so erkannten Typ werden dann die Parsierungsfunktionen für die einzelnen Top-Level-Elemente aufgerufen.

Eine Sonderrolle bei den Parsierungsfunktionen spielt `parse_include_statement()`: Da hier eine weitere Datei gelesen wird, wird `parse_file()` rekursiv aufgerufen. Vor diesem Aufruf werden `currentmf` und `jmp_toplevel` in lokalen Variablen gesichert, da von dem rekursiven Aufruf von `parse_file()` diesen (modulglobalen!) Variablen neue Werte zugewiesen werden, die nur für den rekursiven Aufruf gültig sind.

6.2.3.2. Unterhalb des Top-Levels

Vom Top-Level werden nach dem Aufruf von recognize_line() die Funktionen aufgerufen, die einzelne syntaktische Elemente des Shapefiles parsieren (mit Ausnahme der Sections). Das sind (um's nochmal zusammenzufassen):

- Vclass-Definitionen
- Abhängigkeitsregeln
- Variantendefinitionen
- Makrodefinitionen
- Versionsauswahlregeln
- Include-Statements

Für jedes dieser Elemente gibt es eine eigene Parsierungsfunktion. Mit der Ausnahme von `parse_vclass_definition()` und `parse_include_statement()` bekommen alle die Liste der schon in `recognize_line()` gelesenen Namen als Parameter `name_list` übergeben. Bei `parse_include_statement()` und `parse_vclass_definition()` braucht das nicht zu passieren, da der schon gelesene Name „`vclass`“ bzw. „`include`“ ist.

Außerdem bekommt jede Parsierungsfunktion den noch nicht gelesenen Rest der Zeile als Parameter `line` übergeben.

```
void parse_vclass_definition(char *line) ;
```

Hier steckt nicht viel besonderes drin. Etwas ärgerlich an dieser Stelle ist die ziemlich barocke Syntax der Vclass-Definition mit den Klammern und Kommas. Ohne diese könnten die Varianten-Namen mit `collect_names()` gelesen werden wie in `parse_dependency_rule()`, aber da im Gegensatz zu den meisten anderen Stellen Kommas und Klammern hier nicht in Namen erlaubt sind, mußte ich eine eigene Prozedur `get_variant_name()` schreiben, die genauso wie `get_name()` arbeitet, nur mit Komma und Klammern als zusätzliche Trennzeichen. (Alternativ wäre auch möglich gewesen, `get_name()` mit einem zusätzlichen Parameter für die Trennzeichen zu versehen, ähnlich wie bei `skip_over_name_with_stop()`.)

```
void parse_dependency_rule(char *line, struct stringlist *name_list) ;
```

Hier tauchen Aufzählungen von Namen in der „normalen“ Form an drei verschiedenen Stellen auf: Targets, Dependents und Makronamen (rechts vom zweiten Doppelpunkt. Das bot an, eine Prozedur `collect_names()` zu implementieren, die mit Hilfe von `get_name()` Namen aus einer Zeile „aufsammelt“ und in einer Liste zurückgibt.

Das zur Abhängigkeitsregel gehörende Shell-Skript ist dadurch gekennzeichnet, daß die dazu gehörenden Zeilen mit einem Tabulator anfangen, was das Einsammeln sehr einfach macht. Ansonsten werden diese Zeilen nicht weiter angesehen. Da eine Zeile zu weit gelesen wird, nämlich bis zu der, die *nicht* mit einem Tabulator anfängt, muß wieder um eine Zeile zurückgesetzt werden (`mf_prevline()`).

```
void parse_variant_definition(char *line,
                              struct stringlist *name_list,
                              int in_section) ;
```

Wie oben schon besprochen, bekommt `parse_variant_definition()` noch in einem Parameter übergeben, ob die Variantendefinition in einer Section oder auf dem Top-Level stand. Das ist deshalb wichtig, weil im Top-Level die Zeilen im Rumpf der Regel mit einem Tabulator beginnen müssen, in einer

Section aber nicht.

Bei einer Variantendefinition im Top-Level kann deshalb das Ende der Definition leicht daran erkannt werden, daß die Zeile nicht mit einem Tabulator beginnt. Bei einer Variantendefinition in einer Section ist das schwieriger — hier muß man das Ende einer Variantendefinition etwas umständlicher feststellen. Das passiert mit der Funktion `rule_or_variant_body()` (siehe dort).

Der Rest von `parse_variant_definition()` ist ziemlich geradeaus: Die Zeilen, die zum Rumpf einer Variantendefinition gehören, werden in eine Liste gehängt und zum Schluß mit `queue_variant_definition()` weitergereicht.

```
void parse_macro_definition(char *line, int type,
                            struct stringlist *name_list) ;
```

In Shape sind vier Typen von Makros vorgesehen, die mit „=“, „+=“, „-=“ und „:=“ definiert werden. Das Zeichen dieses Tokens, das nicht „=“ ist, wird als der Parameter `type` übergeben; für ein „=“-Makro ist das ein Blank. Die Makrotypen mit „-=“ und „:=“ sind zur Zeit noch nicht realisiert und werden deswegen abgelehnt.

Da der Wert des Makros einfach der (in `line` übergebene) Rest der Zeile ist, bleibt in `parse_macro_definition()` nicht mehr zu tun, als die Anzahl der Namen in `name_list` zu überprüfen (muß 1 sein) und Namen, Typ und Wert des Makros mit `define_macro()` weiterzureichen.

```
void parse_selection_rule(char *line,
                          struct stringlist *name_list,
                          int in_section) ;
```

`parse_selection_rule()` ist sehr ähnlich zu `parse_variant_definition()`. Der einzige Unterschied besteht darin, daß die „eingesammelten“ Zeilen nicht in eine Liste eingehängt werden, sondern zu einem String zusammengehängt werden (mit `check_sbcat()`), da `atBindAddRule()` (siehe `commit_selrule_definitions()`) den Inhalt einer Auswahlregel als einen einzigen String erwartet.

```
void parse_include_statement(char *line) ;
```

Eine Include-Direktive in einem Shapefile kann mehrere Parameter haben; jeder ist der Name einer zu lesenden Include-Datei. Die Zeile hinter dem „`include`“, die in `line` übergeben wird, wird in eine Liste dieser Namen umgewandelt (`collect_names()`). Da an dieser Stelle auch Makros erlaubt sind, werden vorher eventuell in der Zeile vorkommende Makros expandiert. Nachdem der globale Status gesichert wurde (`jmp_toplevel`, `currentmf`), wird jede der Dateien mit `parse_file()` gelesen. Danach wird der globale Status, der vom rekursiven Aufruf von `parse_file()` geändert worden ist, wiederhergestellt.

6.2.4. Schnittstelle zu den Datenstrukturen von Shape (`queues.c`)

Der Parser gibt fünf Arten von Daten an Shape weiter:

- Makrodefinitionen
- Versionsauswahlregeln
- Variantendefinitionen
- Vclass-Definitionen
- Abhängigkeitsregeln

Um für bestimmte Fälle Zeit zu sparen, gab es den Gedanken, gelesene Konstrukte nicht sofort in die Datenstrukturen von Shape einzutragen, sondern erst, wenn sie gebraucht werden. Werden bestimmte Daten nicht gebraucht, ist damit Zeit gewonnen. Gerade die spezielle Anwendung „`shape -echo`“ [shape(1)], mit der Makrowerte aus Shapefiles wiedergegeben werden, wird oft in Shellskripts verwendet, so daß man das gerne besonders schnell ausgeführt haben mochte. Abhängigkeits- und Versionsauswahlregeln werden dafür nicht gebraucht.

Der Parser muß Makros expandieren können, da als Namen von Include-Dateien auch Makroreferenzen zugelassen sind. Dazu müssen die auch Varianten- und Vclass-Definitionen bekannt sein. „Verzögerte“

Einträge sind also nur für Versionsauswahlregeln und Abhängigkeitsregeln möglich.

Trotzdem heißt das entsprechende Modul des Parsers, das die Listen von bereits gefundenen Daten verwaltet, „queues“ und die Funktionen darin „queue_...“, aber diese Namen mögen sich in zukünftigen Versionen noch ändern.

Bis auf Makrodefinitionen werden alle Daten über eine entsprechende „queue_...“-Funktion an Shape weitergegeben. queue_selection_rule() und queue_dependency_rule() tragen dabei die übergebenen Daten in eine Liste ein (selrulelist und deprulelist), deren Inhalt später mit commit_selrule_definitions() bzw. commit_deprule_definitions() an Shape weitergereicht wird.

```
void queue_dependency_rule(struct stringlist *targets,
                           struct stringlist *deps,
                           struct stringlist *macros,
                           struct stringlist *cmds) ;
```

Aus den übergebenen Listen von Targets, Dependents, Makros und Kommandos wird eine Struktur aufgebaut, die in deprulelist eingetragen wird.

```
void queue_selection_rule(char *name, char *body) ;
```

Entsprechend zu queue_dependency_rule().

```
void queue_variant_definition(char *name,
                              struct stringlist *macros) ;
```

queue_variant_definition() trägt die übergebenen Daten selbst in die Datenstrukturen von Shape ein. Auf diese Weise sollen später einmal auch queue_vclass_definition(), commit_deprule_definitions() und commit_selrule_definitions() arbeiten, weil damit die Redundanz, die mit der zur Zeit benutzten alten Schnittstelle auftritt, vermieden wird.

```
void queue_vclass_definition(char *name, struct stringlist *elements) ;
```

Die übergebenen Daten werden über vclassdef() in die Shape-Datenstrukturen eingetragen.

```
void commit_selrule_definitions(void) ;
```

Die in selrulelist abgelegten Daten werden über ruledef(), rulecont() und ruleend() in die Shape-Datenstrukturen eingetragen.

```
void commit_deprule_definitions(void) ;
```

Entsprechend zu commit_selrule_definitions().

```
void commit_definitions(void) ;
```

Ruft commit_selrule_definitions() und commit_deprule_definitions() auf. commit_definitions() ist der vorgesehene Weg, Abhängigkeits- und Versionsauswahlregeln einzutragen. Die beiden anderen „commit_...“-Funktionen wurden trotzdem exportiert, um eventuellen zukünftigen Bedürfnissen eine größere Flexibilität zu bieten.

7. Rückblick

Der neue Parser von Shape sollte vor allem drei Kriterien erfüllen:

- gesteigerte Effizienz
- klarere Fehlermeldungen
- bessere Wartbarkeit.

7.1. Effizienz

Die Effizienz läßt sich leider nicht direkt prüfen, da Shape gleichzeitig mit dem neuen Parser auch mit einer neuen Versionsbindungskomponente verheiratet worden ist, die schneller arbeitet als die alte. Auf jeden Fall ist Shape mit diesen beiden neuen Komponenten deutlich schneller geworden; vor allem ist die „Atempause“, die früher beim Start von Shape vor dem Erzeugen der Targets zu beobachten war, verschwunden.

7.2. Fehlermeldungen

Das neue Shape gibt Fehlermeldungen in einem ähnlichen Format wie der GNU C Compiler gcc aus:

<Dateiname>:<Zeile>: <Meldung>

Das sieht zum Beispiel so aus:

```
Shapefile:4: shape parse error: dependency rule must begin with target
```

Dieses Format ist ein informeller Standard bei einer ganzen Reihe von Unix-Dienstprogrammen. Von Emacs wird dieses Format so erkannt, daß der Benutzer mit der Emacs-Funktion `next-error` sich sofort mit einem Kommando die monierte Stelle in der entsprechenden Datei ansehen kann, wenn er Shape über das „`compile`“-Kommando von Emacs gestartet hat.

Das alte Shape gibt an der gleichen Stelle meistens ohne weitere Meldung die fehlerhafte Zeile auf die Standardausgabe aus.

7.3. Wartbarkeit

Ich habe mich bemüht, durch gute Strukturierung des Programmcodes und ausreichende Kommentierung eine gute Lesbarkeit und damit Wartbarkeit des Shape-Parsers zu erreichen. Ein problematischer Punkt ist eventuell die etwas umständliche Prozedur `recognize_line()`, in der viel Wissen über die Syntax von Shapefiles verborgen ist. Hier Änderungen oder Erweiterungen anzubringen könnte unter Umständen schwierig sein.

Die Arbeit an zwei Erweiterungen, die Axel Mahler kürzlich am Parser vorgenommen hat, zeigte, daß der neue Parser tatsächlich leichter wartbar ist als der alte.

Einen zusätzlichen Vorzug hat der neue Parser: In dieser Arbeit sind die wesentlichen Gedanken und Konzepte dokumentiert, auf denen er beruht. Das sollte die Wartbarkeit auf jeden Fall fördern.

8. Ausblick

8.1. Datenstrukturen

Shape ist immer noch nicht „fertig“. Was bisher noch zu wünschen übrig läßt, ist die Schnittstelle zu den globalen Datenstrukturen, die die Bemühungen bei der Implementierung des Parsers um brauchbare und konsistente Quotingregeln bisher noch zunichte macht und unnötigen Aufwand treibt, indem bereits in Tokens zerlegte Zeilen erneut parsiert werden. Prozeduren wie `vclassdef()` erwarten ein einziges String-Argument, in dem die Vclass-Definition in praktisch der gleichen Form steht wie im Shapefile. Da innerhalb von `vclassdef()` keine Quoting-Zeichen mehr berücksichtigt werden, werden Daten eventuell falsch interpretiert. (Natürlich ist es nicht besonders sinnvoll, Leerzeichen in Variantennamen unterzubringen, aber meines Erachtens sollten solche Möglichkeiten nicht stärker als nötig eingeschränkt werden.)

Das gleiche gilt für Abhängigkeitsregeln. Hier würde eine Renovierung der Schnittstelle zu den Datenstrukturen auch gleich dazu einladen, die Datenstrukturen selbst neueren Bedürfnissen anzupassen.

8.2. Syntax

Zur Zeit schleppt Shape immer noch die „Altlast“ der RULE- und VARIANT-SECTIONS mit sich herum. Das hat den Nachteil, daß der Code des Parsers umfangreicher ist, als er sein müßte; außerdem wird (in der Prozedur `rule_or_variant_body()`) mit nicht sehr schönen Mitteln gearbeitet. Diese Sections sollen in Zukunft wegfallen.

Zur Zeit ist es noch so, daß Auswahlregeln und Variantenaktivierungen wie normale Dependents in der Abhängigkeitszeile stehen, wobei eine Auswahlregel als erstes Dependent stehen muß, Varianten dahinter, aber auch noch *vor* den eigentlichen Dependents. Hier ist schon angedacht, daß sie in Zukunft eindeutig markiert werden. Im Gespräch war eine Markierung „`@+` für Varianten und „`@-`“ für Auswahlregeln, korrespondierend mit der Syntax der Varianten- bzw. Regeldefinition. Dadurch ist es einerseits möglich, daß Auswahlregeln und Varianten beliebig in der Abhängigkeitszeile verteilt werden können, andererseits kann dann auch schon der Parser diese Tokens als solche erkennen und gleich entsprechend in die Datenstrukturen eintragen. Das würde zusammen mit einer entsprechenden Überarbeitung der Datenstrukturen eine Vereinfachung dieses Mechanismus bewirken.

Diese Änderungen in der Syntax sind inkompatibel zum alten Shape. Deshalb sollen sie erst im nächsten Major Release von ShapeTools (2.0) eingebaut werden, zusammen mit einer Reihe weiterer Änderungen. Hier ist zum Beispiel eine Anbindung an einen Tcl-Interpreter im Gespräch, so daß die Shellskripts in Tcl geschrieben werden können und im gleichen Prozeß abgearbeitet werden können. Abgesehen von einer deutlichen Performance-Verbesserung hätte man vom Tcl-Skript auch direkten Zugriff auf AtFS-Funktionen — die sich hieraus ergebenden Möglichkeiten sind noch kaum im Ansatz abzusehen.

8.3. Fehlermeldungen

In einer frühen Version des Parsers habe ich zu Testzwecken nicht nur die Zeile, sondern auch die Spalte eines erkannten Syntaxfehlers mit ausgegeben. Vielleicht sollte das (eventuell über eine Option einschaltbar) auch in einer ausgelieferten Version passieren — es wäre jedenfalls leicht, das noch mit einzubauen.

Anhang A: Literatur

[Clocksin, Mellish]
W. F. Clocksin, C. S. Mellish: Programming in Prolog, Cambridge (England) 1984

[egrep]
Unix Manual Page zu egrep(1), 4.3 BSD, University of California at Berkeley 1986

[Feldman]
Stuart I. Feldman: Make — a program for maintaining computer programs, in *Software — Practice and Experience* , Vol. 9, No.4, pp. 225−265, Apr. 1979

[K&R]
Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language, 1978

[Sun Make]
Unix Manual Page zu make(1), SunOS 4.1.2, Sun Mincrosystems Inc. 1990

[GNU Make]
Richard M. Stallman, Roland McGrath: GNU Make — A Program for Directing Recompilation, Edition 0.27 Beta, Free Software Foundation Inc. 1990

[BSD Make]
Unix Manual Page zu make(1), 4.3 BSD, University of California at Berkeley 1986

[Dmake]
Dennis Vadura: Unix Manual Page zu dmake(1), University of Waterloo

[vbind]
Andreas Lampen: Version Binding in ShapeTools, STONE Technical Report TUB 0023.01, Technische Universität Berlin 1992

[Tutorial]
A. Mahler: Using the Shape Toolkit for Cooperative Software Development, in

[Mahler, Lampen 88]
Axel Mahler, Andreas Lampen: An Integrated Toolset for Engineering Software Configurations, in *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments* pp. 191−200, SIGSOFT SE Notes Vol. 13-No.5, SIGPLAN Notices Vol. 24-No.2, Boston MA, 28-30 November 1988

[shape(1)]
Axel Mahler: Unix Manual Page zu shape(1), Technische Universität Berlin 1992

[R^3RS]
Jonathan Ree, William Clinger (Hrsg.): The Revised^3 Report On The Algorithmic Language Scheme, in *ACM SIGPLAN Notices* 21(12):37−39

Anhang B: Die Grammatik im Zusammenhang

```
Shapefile       ::=   { {       deprule
                        |       blank* macrodef
                        |       vclassdef
                        |       selrule
                        |       vardef
                        |       rulesection
                        |       varsection
                        |       include
                        |       blank*    }    [comment] newline }*

deprule         ::=   dependency  [ whitespace* semicolon deprulebody ] newline
                        {tab deprulebody newline}*

dependency      ::=   blank* name {whitespace+ name}* whitespace*
                        ":" whitespace* [name] {whitespace+ name}*
                        [ ":" whitespace* [name] {whitespace+ name}* ]

deprulebody     ::=   "[^\n]*"

macrodef        ::=           { name whitespace*  "=" [string] [comment] newline }
                        |     { name whitespace* "+=" [string] [comment] newline }
                        |     { name whitespace* "-=" [string] [comment] newline }
                        |     { name whitespace* ":=" [string] [comment] newline }

vclassdef       ::=   blank* "vclass" whitespace+ name whitespace*
                        ":=" whitespace* "(" whitespace* name
                        { "," whitespace* name whitespace* }* ")"
                        whitespace* newline

selrule         ::=   blank* name whitespace* ":-" whitespace* newline
                        {tab whitespace* rulebody [comment] newline}+

rulebody        ::=   {bodystring | "(" rulebody ")"}+

bodystring      ::=   "[^#()\n]*"

vardef          ::=   blank* name whitespace* ":+" whitespace* newline
                        {tab macrodef newline}+

rulesection     ::=   blank* "#%" whitespace* "RULE-SECTION" whitespace* newline
                        { oldselrule | {blank* [comment] newline}}*
                        blank * "#%" whitespace* "END-RULE-SECTION" whitespace* newline
```

```
oldselruleordef ::=  blank* name whitespace* ":" whitespace* newline
                       {whitespace* rulebody [comment] newline}+

varsection      ::=  blank* "#%" whitespace* "VARIANT-SECTION" whitespace* newline
                       { oldvardef | {blank* [comment] newline}}*
                       blank* "#%" whitespace* "END-VARIANT-SECTION" whitespace*
                       newline

oldvardef       ::=  blank* name whitespace* ":" whitespace* newline
                       {whitespace* macrodef newline}+

include         ::=  blank* "include" {whitespace+ name}+
                       whitespace* newline

blank           ::=  " "

comment         ::=  "#[^\n]*"

newline         ::=  "\n"

semicolon       ::=  ";"

tab             ::=  "\t"

string          ::=  {"[^#\n\"']+" | quotedstring}+

quotedstring    ::=  "\"[^\"\n]*\"" | "'[^'\n]*'"

whitespace      ::=  "[ \t]"

name            ::=  {"[^#:=; \t\n]" | quotedstring}+
```

Anhang C: Quelltext

mfiles.h

```
/* Copyright (C) 1993,1994 by the author(s).

 This software is published in the hope that it will be useful, but
 WITHOUT ANY WARRANTY for any part of this software to work correctly
 or as described in the manuals. See the ShapeTools Public License
 for details.

 Permission is granted to use, copy, modify, or distribute any part of
 this software but only under the conditions described in the ShapeTools
 Public License. A copy of this license is supposed to have been given
 to you along with ShapeTools in a file named LICENSE. Among other
 things, this copyright notice and the Public License must be
 preserved on all copies.
*/
/*
 * ShapeTools/shape program - mfiles.h
 *
 * by Juergen.Nickelsen@cs.tu-berlin.de
 *
 * $Header: mfiles.h[9.0] Tue Mar  2 15:02:19 1993 shape@cs.tu-berlin.de accessed $
 */
#ifndef __MFILES_H
#define __MFILES_H

#define STDIN_NAME "Standard Input" /* Name used for standard input */
#define INITIAL_BUFFER_SIZE 8192    /* initial buffer size for reading
                                       from special file or stdin */

typedef struct MFILE__ {
    char           *mf_name ;        /* filename */
    char           *mf_buffer ;        /* pointer to mfile buffer */
    char          **mf_lbeg ;        /* array of original line beginnings */
    int             mf_lines ;        /* number of lines */
    int             mf_curline ;/* current line */
    char           *mf_end ;        /* end of buffer */
    struct MFILE__ *mf_next ;        /* pointer to next, don't use! */
} MFILE ;

#include "mffuncs.h"
#endif /* __MFILES_H */
```

mfiles.c

```
/* Copyright (C) 1993,1994 by the author(s).

 This software is published in the hope that it will be useful, but
 WITHOUT ANY WARRANTY for any part of this software to work correctly
 or as described in the manuals. See the ShapeTools Public License
 for details.

 Permission is granted to use, copy, modify, or distribute any part of
 this software but only under the conditions described in the ShapeTools
 Public License. A copy of this license is supposed to have been given
 to you along with ShapeTools in a file named LICENSE. Among other
 things, this copyright notice and the Public License must be
 preserved on all copies.
*/
/*
 * ShapeTools/shape program - mfiles.c
 *   implements memory file. A memory file is completely
 *   read into a buffer. All operations work on that buffer.
 *
 * by Juergen.Nickelsen@cs.tu-berlin.de
 *
 * $Header: mfiles.c[9.0] Tue Jul  6 15:46:00 1993 axel@cs.tu-berlin.de accessed $
 */
#ifndef lint
static char *AtFSid = "$Header: mfiles.c[9.0] Tue Jul  6 15:46:00 1993 axel@cs.tu-berlin.de accessed $
#endif


#include "shape.h"
#include "mfiles.h"
#include "parser.h"

/* list of all MFILE structures */
static MFILE *mf_structlist = NULL ;

/* internal functions */
static void mf_do_read A((MFILE *mf, int fd)) ;
static void mf_split_lines A((MFILE *mf)) ;
static MFILE *mf_struct_alloc A((char *fname)) ;
static int mf_open A((char *fname, FILE **fpp)) ;
static void mf_readin_special A((MFILE *mf, int handle)) ;
static MFILE *mf_readin_nosplit A((char *fname)) ;

static MFILE *mf_struct_alloc(fname)
char *fname;
{
    MFILE *mf ;

    mf = (MFILE *) check_malloc(sizeof(MFILE)) ;
    mf->mf_name = check_strdup(fname) ;
    mf->mf_buffer = NULL ;
    mf->mf_lbeg = NULL ;
```

```
    mf->mf_lines = 0 ;
    mf->mf_curline = -1 ;
    mf->mf_end = NULL ;
    mf->mf_next = mf_structlist ;
    mf_structlist = mf ;

    return mf ;
}

/* Kill an MFILE buffer pointed to by mf. Releases all memory
   associated with this buffer. */
void mf_kilbuf(mf)
MFILE *mf;
{
    MFILE *tmp ;

    if (mf == mf_structlist) {
        mf_structlist = mf->mf_next ;
    } else {
        for (tmp = mf_structlist; tmp->mf_next != NULL; tmp = tmp->mf_next) {
            if (tmp->mf_next == mf) {
                tmp->mf_next = mf->mf_next ;
            }
        }
        if (tmp->mf_next == NULL) {
#ifdef DEBUG
            fatal("freeing spurious buffer", mf->mf_name) ;
#endif
            return ;                /* not found in our list */
        }
    }
    free(mf->mf_name) ;
    free(mf->mf_buffer) ;
    free(mf->mf_lbeg) ;
    free(mf) ;
}

/* Kill all MFILE buffers. */
void mf_killall()
{
    MFILE *tmp, *next ;

    for (tmp = mf_structlist, next = mf_structlist->mf_next;
         tmp != NULL;
         tmp = tmp->mf_next, next = next->mf_next) {

        free(tmp->mf_name) ;
        free(tmp->mf_buffer) ;
        free(tmp->mf_lbeg) ;
        free(tmp) ;
    }
}

/* Read standard input into MFILE buffer */
```

```
MFILE *mf_readin_stdin()
{
    MFILE *mf ;

    mf = mf_struct_alloc("Standard Input") ;
    mf_readin_special(mf, 0) ;
    mf_split_lines(mf) ;

    return mf ;
}

static void mf_readin_special(mf, handle)
MFILE *mf ;
int handle ;
{
    unsigned long buffer_size ;        /* current size of buffer */
    unsigned long increment ;        /*  */
    unsigned long read_ret ;        /* return value of read(2) */
    unsigned long total ;        /* no of bytes read in, next read()
                                    goes to this offset */

    buffer_size = INITIAL_BUFFER_SIZE ;
    increment = INITIAL_BUFFER_SIZE ;
    mf->mf_buffer = check_malloc(INITIAL_BUFFER_SIZE) ;

    total = 0 ;
    do {
        if ((read_ret = read(handle, mf->mf_buffer + total,
                            buffer_size - total)) == -1) {
            fatal_perror(mf->mf_name) ;
        }
        total += read_ret ;
        if (total > 3 * buffer_size / 4) {
            increment = increment * 4 / 3 ;
            mf->mf_buffer = check_realloc(mf->mf_buffer,
                                        buffer_size + increment) ;
            buffer_size += increment ;
        }
    }  while (read_ret > 0);
    mf->mf_buffer = check_realloc(mf->mf_buffer, total + 1) ;

    mf->mf_end = mf->mf_buffer + total ;
    *(mf->mf_end) = '\0' ;
}



/* Read file fname into an MFILE buffer */
MFILE *mf_readin(fname)
char *fname;
{
    MFILE *mf ;                        /* struct holding buffer information */

    mf = mf_readin_nosplit(fname) ;
```

```
    if (mf != NULL) {
        mf_split_lines(mf) ;
    }

    return mf ;
}

/* Read file fname into an MFILE buffer without splitting into lines */
static MFILE *mf_readin_nosplit(fname)
char *fname;
{
    MFILE *mf ;                          /* struct holding buffer information */
    unsigned long fsize ;        /* size of the file */
    int fd ;                          /* file descriptor to read from */
    struct stat statbuf ;
    FILE *in ;                          /* *this* must be closed instead of fd */

    if ((fd = mf_open(fname, &in)) == -1) {
        return NULL ;
    } else {
        if (fstat(fd, &statbuf) == -1) {
            fatal_perror(fname) ;
        }
        mf = mf_struct_alloc(fname) ;
        if (S_ISREG(statbuf.st_mode)) {
          fsize = statbuf.st_size ;
          mf->mf_buffer = check_malloc(fsize + 1) ;
          mf->mf_end = mf->mf_buffer + fsize ;
          *(mf->mf_end) = '\0' ;
          mf_do_read(mf, fd) ;
        } else {
          /* if it is not a regular file, we don't know the size */
          mf_readin_special(mf, fd) ;
        }
        fclose(in) ;
    }

    return mf ;
}

/* Open file fname, return file descriptor. Pointer to FILE structure
   is written into place filepp points to. */
static int mf_open(fname, filepp)
FILE **filepp ;
char *fname ;
{
    Af_key *key ;                /* key for af_open() */

    if (!(key = atBindVersion (fname, "")))
      return -1;

    if ((*filepp = af_open(key, "r")) == NULL) {
        /* This must be an error because AtFS promised it could
         * open the file by giving us a key */
```

```
        af_perror(fname) ;
        fatal("AtFS error", NULL) ;
    }

    return fileno(*filepp) ;
}

/* Read the contents of the file fd is a descriptor to into buffer mf.
   Space for the file contents is already allocated. */
static void mf_do_read(mf, fd)
MFILE *mf;
int fd ;
{
    unsigned long left ;        /* bytes left to read */
    unsigned long read_ret ;        /* return value of read(2) */
    char *next_here ;               /* next read(2) goes here */

    left = mf->mf_end - mf->mf_buffer ;
    next_here = mf->mf_buffer ;
    while (left > 0) {
        if ((read_ret = read(fd, next_here, left)) == -1) {
            fatal_perror(mf->mf_name) ;
        }
        left -= read_ret ;
        next_here += read_ret ;
    }
}

/* Split the file buffer mf into lines. Newlines are replaced by null bytes,
   backslash/newline sequences are replaced by two blanks. Pointers to the
   original line beginnings are written into mf->mf_lbeg[].
   mf->mf_lbeg[mf->mf_lines] points to the first character past the last
   line, i.e. mf->mf_end.
   The size of this array is written into mf->mf_lines. */

static void mf_split_lines(mf)
MFILE *mf;
{
    /* for the sake of efficiency backslash-newline pairs make
     * continuation lines even inside comments, i.e. *always* */

    unsigned long enoli ;        /* estimated number of lines */
    unsigned long lines        ;        /* actual number of lines */
    char *bufptr ;               /* pointer into buffer */
    char *oldbufptr ;               /* copy of bufptr */

#define CHARS_PER_LINE 15

    enoli = (mf->mf_end - mf->mf_buffer) / CHARS_PER_LINE + 1 ;
                                /* perhaps... */

    mf->mf_lbeg = (char **) check_malloc(enoli * sizeof(char *)) ;

    bufptr = mf->mf_buffer ;
```

```
    lines = 0 ;

    while (bufptr < mf->mf_end) { /* bufptr gets incremented in the switch */
        mf->mf_lbeg[lines++] = bufptr ;        /* get beg. of current line */
        if (lines >= enoli) {
            /* extend array of pointers */
            enoli *= 2 ;
            mf->mf_lbeg =
                (char **) check_realloc((char *) mf->mf_lbeg,
                                        enoli * sizeof(char *)) ;
        }
        oldbufptr = bufptr ;
        if ((bufptr = strchr(bufptr, '\n')) == NULL) {
            bufptr = oldbufptr ;
            break ;                /* no newline to end of buffer */
        }
        if (*(bufptr - 1) == '\\') {
                                /* continuation line */
            *(bufptr - 1) = ' ' ;
            *bufptr++ = ' ' ;
        } else {
            *bufptr++ = '\0' ;
        }
    }
    if (bufptr + strlen(bufptr) != mf->mf_end) {
        fatal("Nullbyte in file", mf->mf_name) ;
    }
    mf->mf_lbeg = (char **) check_realloc((char *) mf->mf_lbeg,
                                          (lines + 1) * sizeof(char *)) ;
    mf->mf_lbeg[lines] = mf->mf_end ;
    mf->mf_lines = lines ;
}


/* The current line is set to the next line. A pointer to the beginning
   of this line is returned. */
char *mf_nextline(mf)
MFILE *mf;
{
    int line = mf->mf_curline ;

    if (line < 0) {
        /* if current position is before the beginning of the file,
         * go to the first line */
        line = 0 ;
    } else {
        do {
            if (++line >= mf->mf_lines) {
                /* end of file reached */
                return NULL ;
            }
            /* beginning of a real line (not original)
             * is only after a null byte */
        } while (*(mf->mf_lbeg[line] - 1) != '\0') ;
```

```
    }
    mf->mf_curline = line ;
    return mf->mf_lbeg[line] ;
}


/* The current line is set to the previous line. A pointer to the beginning
   of this line is returned. */
char *mf_prevline(mf)
MFILE *mf;
{
    int line = mf->mf_curline ;

    if (--line < 0) {
        /* kind of "virtual" position before the beginning */
        mf->mf_curline = -1 ;
    } else {
        /* look for the beginning of a "real" line. A real line
         * beginning if found if it is the beginning of the buffer
         * or otherwise has a null byte in the previous position. */
        while (line > 0 &&
               *(mf->mf_lbeg[line] - 1) != '\0') {
            line-- ;
        }
        mf->mf_curline = line ;
    }
    return line >= 0 ? mf->mf_lbeg[line] : NULL ;
}


/* Return a pointer to the specified line in mf. */
char *mf_thisline(mf, lineno)
MFILE *mf;
int lineno ;
{
    return mf->mf_lbeg[lineno - 1] ;
}

/* The file name and line number ptr points into are written into
   *pfilename and *plineno. If ptr does not point into an MFILE
   buffer, 0 is returned, 1 on success. */
int mf_locate(ptr, pfilename, plineno)
char *ptr ;
char **pfilename ;
int *plineno ;
{
    MFILE *mf ;
    int lineno ;

    /* go through mf_structlist and try to locate ptr */
    for (mf = mf_structlist; mf != NULL; mf = mf->mf_next) {
        if ((lineno = mf_locate_in(mf, ptr)) > 0) {
            *plineno = lineno ;
            if (pfilename != NULL) {
```

```
                *pfilename = mf->mf_name ;
            }
             return 1 ;
        }
    }
    return 0 ;
}

/* The line number ptr points into in the buffer of mf is returned.
   If ptr does not point into the buffer of mf, -1 is returned. */
int mf_locate_in(mf, ptr)
MFILE *mf ;
char *ptr ;
{
    unsigned long here ;
    unsigned long upper ;
    unsigned long lower ;

    /* check bounds first */
    if (ptr >= mf->mf_buffer && ptr <= mf->mf_end) {
        upper = mf->mf_lines - 1 ;
        lower = 0 ;

        /* kind of binary search */
        while (lower <= upper) {
            here = (lower + upper) / 2 ;
            if (ptr > mf->mf_lbeg[here]) {
                if (ptr < mf->mf_lbeg[here + 1]) { /* in THIS line */
                    return here + 1 ;
                } else {
                    lower = here + 1 ;
                }
            } else if (ptr < mf->mf_lbeg[here]) {
                upper = here ;
            } else {
                return here + 1 ;
            }
        }
    }
    return -1 ;
}

/* set the current line of mf to the line ptr points to. If ptr does
   not point into the buffer of mf, a fatal error is raised. */
void mf_setline(mf, ptr)
MFILE *mf ;
char *ptr ;
{
    int lineno ;                /* number of line ptr points to */

    lineno = mf_locate_in(mf, ptr) ;
    if (lineno < 0) {
        fatal("ptr not in mf", mf->mf_name) ;
    }
```

```
    mf->mf_curline = lineno - 1 ;
                                /* mf_locate_in() returns user level
                                   line numbers, not an index in mf_lbeg */
}

/*EOF*/
```

utils.c

```
/* Copyright (C) 1993,1994 by the author(s).

 This software is published in the hope that it will be useful, but
 WITHOUT ANY WARRANTY for any part of this software to work correctly
 or as described in the manuals. See the ShapeTools Public License
 for details.

 Permission is granted to use, copy, modify, or distribute any part of
 this software but only under the conditions described in the ShapeTools
 Public License. A copy of this license is supposed to have been given
 to you along with ShapeTools in a file named LICENSE. Among other
 things, this copyright notice and the Public License must be
 preserved on all copies.
*/
/*
 * ShapeTools/shape program - utils.c
 *
 * by Juergen.Nickelsen@cs.tu-berlin.de
 *
 * $Header: utils.c[9.0] Thu Jun 10 18:30:02 1993 axel@cs.tu-berlin.de accessed $
 */
#ifndef lint
static char *AtFSid = "$Header: utils.c[9.0] Thu Jun 10 18:30:02 1993 axel@cs.tu-berlin.de accessed $"
#endif

#include <errno.h>
#include <ctype.h>
#include "shape.h"
#include "parser.h"

/*
 * The memory management functions do the following: allocate, check
 * if result is valid, initialize fields if necessary
 */
```

```
/*** raw allocators ***/

/* allocate memory and check */
char *check_malloc(size)
unsigned size ;
{
    char *tmp ;

    tmp = malloc(size) ;
    if (tmp == NULL) {
        fatal(NOMORECORE, NULL) ;
    }
    return tmp ;
}

/* reallocate memory and check */
char *check_realloc(ptr, size)
char *ptr ;
unsigned size ;
{
    char *tmp ;

    tmp = realloc(ptr, size) ;
    if (tmp == NULL) {
        fatal(NOMORECORE, NULL) ;
    }
    return tmp ;
}
```

```
/*** string functions ***/


/* like strdup, but only n characters
 * the resulting string is null padded, freeable
 */
char *check_strdup(s)
char *s ;
{
    char *tmp = check_malloc(strlen(s) + 1) ;

    strcpy(tmp, s) ;
    return tmp ;
}


/* like strdup, but only n characters
 * the resulting string is null padded, freeable
 */
char *check_strndup(s, n)
char *s ;
int n ;
{
    char *tmp = check_malloc(n + 1) ;

    strncpy0(tmp, s, n) ;
    return tmp ;
}


/* safe strncpy with null padding but no return value */
void strncpy0(s1, s2, n)
char *s1, *s2 ;
int n ;
{
    strncpy(s1, s2, n) ;
    s1[n] = '\0' ;
}

sb_ptr check_sbcat(sb, string)
sb_ptr sb ;
char *string ;
{
    if ((sb = sbcat(sb, string, strlen(string))) == NULL) {
        fatal(NOMORECORE, NULL) ;
    }

    return sb ;
}

/* add one character to a string buffer */
sb_ptr check_sbaddc(sb, c)
sb_ptr sb ;
char c ;
```

```
{
    if ((sb = sbcat(sb, &c, 1)) == NULL) {
        fatal(NOMORECORE, NULL) ;
    }

    return sb ;
}

sb_ptr check_sbnew(len)
int len ;
{
    sb_ptr sb ;

    if ((sb = sbnew(len)) == NULL) {
        fatal(NOMORECORE, NULL) ;
    }

    return sb ;
}

/* allocate a struct stringlist. This struct is pushed on list
 * with contents str. */

struct stringlist *push_stringlist(str, list, freeable)
char *str ;                         /* string value */
struct stringlist *list ;        /* list to push struct on */
int freeable ;                        /* if non-zero, string can be freed */
{
    struct stringlist *s ;        /* pointer to allocated list */

    s = (struct stringlist *) check_malloc(sizeof(struct stringlist)) ;
    s->string = str ;                /* set string value */
    s->freeable = freeable ;        /* for free_stringlist() */
    s->next = list ;                /* put in front of list */

    return s ;                          /* return head of new list */
}


/* revert a stringlist. This is necessary for rule.cmds,
 * since the commands are added with push_stringlist().
 */
struct stringlist *revert_stringlist(strlp)
struct stringlist *strlp ;
{
    struct stringlist *tmp, *new ;

    new = NULL ;                /* initialize new list */

    while (strlp != NULL) {
                                /* put head of old list to
                                 * head of new list */
        tmp = strlp->next ;
        strlp->next = new ;
```

```
        new = strlp ;
        strlp = tmp ;
    }

    return new ;
}


/* free all elements of a stringlist */
void free_stringlist(strlist)
struct stringlist *strlist ;
{
    struct stringlist *tmp ;

    while (strlist != NULL) {
        tmp = strlist ;
        strlist = strlist->next ;
        if (tmp->freeable) {
            free(tmp->string) ;
        }
        free(tmp) ;
    }
}
```

```
/*** errors and warnings ***/

/* print fatal error message and exit */
void fatal(message, message2)
char *message ;
char *message2 ;
{
    fprintf(stderr, "%s fatal error: %s", stProgramName, message) ;
    if (message2 != NULL) fprintf(stderr, " (%s)", message2) ;
    fputc('\n', stderr) ;

#ifdef DEBUG
    abort() ;
#else
    cleanup() ;
    exit(3) ;
#endif
}
```

```
/* specials for the parser */

/* char *skip_over_XXX(char *p): skip over item XXX */

char *skip_over_blanks(p)
char *p ;
{
    while (*p == ' ') p++ ;
    return p ;
}


/* stops at NUL byte */
char *skip_over_whitespace(p)
char *p ;
{
    while (*p && isspace(*p)) p++ ;
    return p ;
}

/* skip over a name or macro reference */

char *skip_over_name(p)
char *p ;
{
    return skip_over_name_with_stop(p, "") ;
}


/* skip over a name or macro reference. Stops at notnamecomponents
 * and at characters in stop */

char *skip_over_name_with_stop(p, stop)
char *p ;
char *stop ;                        /* additional characters to stop at */
{
    char in_quote = 0 ;                /* initially outside of quoting */
    int ignore_parens ;                /* if non-zero, don't do paren matching */
    int parenth_level = 0 ;        /* initially no parantheses */

    ignore_parens = strchr(stop, ')') || strchr(stop, '(') ;
    /* loop over string */
    while (*p) {
        switch (*p) {
          case '\"':                /* quoting character encountered */
          case '\'':
          case '`':                /* backquote counts as quoting here */
            if (in_quote) {
                if (in_quote == *p) { /* end of quoting */
                    in_quote = 0 ;
                }
            } else {                /* beginning of quoting */
                in_quote = *p ;
            }
```

```
            break ;
          case '(':
            /* if parentheses are to be ignored, we have to fall
             * through to the "default:" label. This looks pretty ugly,
             * I know!
             */
            if (!ignore_parens) {
                parenth_level++ ;
                break ;
            }
          case ')':
            if (!ignore_parens) {
                if (parenth_level <= 0) {
                    parsing_error("unmatched closing parenthesis", p, NULL) ;
                }
                parenth_level-- ;
                break ;
            }
          default:
            if (!in_quote &&
                (is_no_namecomponent(*p) || strchr(stop, *p)) &&
                (ignore_parens || parenth_level == 0)) {
                /* return if:
                 *  - outside of quoting
                 *  - all parentheses closed (or ignored)
                 *  - character is
                 *      - no namecomponent
                 *      - or is stop character
                 */

                return p ;        /* points at first character after name */
            }
        }
        p++ ;
    }

    /* quoting must be terminated at end of line */
    if (in_quote) {
        parsing_error("non-terminated quoting at end of line", p - 1, NULL) ;
    }

    /* all parentheses must be matched */
    if (!ignore_parens && parenth_level != 0) {
        parsing_error("mismatched parenthesis", p, NULL) ;
    }

    return p ;
}

/* copies a name after point *linep points to a string
 * and returns a pointer to the allocated string. *linep is incremented
 * to point to the first character after the name.
 * Any quoting is removed from the name.
 * macro references are treated as names. */
```

```
char *get_name(linep)
char **linep ;                          /* pointer to pointer to string */
{
    char *name_beg ;                /* pointer to beginning of name */
    char *name_end ;                /* pointer to end of name */

    name_beg = skip_over_whitespace(*linep) ;
    name_end = skip_over_name(name_beg) ;

    if (name_beg == name_end) {
        parsing_error("name or macro expected", *linep, NULL) ;
    }

    *linep = name_end ;
    return unquote(check_strndup(name_beg, name_end - name_beg)) ;
}


/* remove pairs of quoting characters from string. The quoting characters
 * are already guaranteed to be matching by skip_over_name().
 * The unquoting is done in place, but the address of the string
 * is also returned.
 */

char *unquote(string)
char *string ;
{
    int distance = 0 ;                /* distance to move folling characters */
    char in_quote = 0 ;                /* character of current quoting */
    char *from ;                /* place to copy characters from... */
    char *to ;                        /* ... and to. */

    to = from = string ;
    while (*from) {
        if (in_quote) {
            if (*from == in_quote) {
                from++ ;
                in_quote = 0 ;
                distance++ ;
            } else {
                *to++ = *from++ ;
            }
        } else {
            if (*from == '\'' || *from == '\"') {
                in_quote = *from ;
                from++ ;
                distance++ ;
            } else {
                *to++ = *from++ ;
            }
        }
    }
    *to = '\0' ;
#ifdef DEBUG
```

```
    if (distance != from - to) {
        fatal("wrong distance in unquote()", "Hmm...") ;
    }
#endif

    return string ;
}


/* int is_no_namecomponent(char c) */

char notnamecomponents[] = "#:;= " ;

/* may become a macro eventually */
/* is true for NUL character */
int is_no_namecomponent(c)
long c ;
{
    return (int) strchr(notnamecomponents, c) ;
}



/*EOF*/
```

parser.h

```
/* Copyright (C) 1993,1994 by the author(s).

 This software is published in the hope that it will be useful, but
 WITHOUT ANY WARRANTY for any part of this software to work correctly
 or as described in the manuals. See the ShapeTools Public License
 for details.

 Permission is granted to use, copy, modify, or distribute any part of
 this software but only under the conditions described in the ShapeTools
 Public License. A copy of this license is supposed to have been given
 to you along with ShapeTools in a file named LICENSE. Among other
 things, this copyright notice and the Public License must be
 preserved on all copies.
*/
/*
 * ShapeTools/shape program - parser.h
 *
 * by Juergen.Nickelsen@cs.tu-berlin.de
 *
 * $Header: parser.h[9.0] Thu Jun 10 18:30:06 1993 axel@cs.tu-berlin.de accessed $
 */
#ifndef NULL
#define NULL 0
#endif
#define NOMORECORE        "no more core"



#define SB_LENGTH                128 /* initial length of string buffer
                                      * for dependency rules etc. */


#ifdef DEBUG
extern int dump_parsed_definitions ;
#endif
extern int errno ;                /* I want to handle ... */
extern char *sys_errlist[] ;        /* ... these strings by myself */
#define fatal_perror(string)    fatal(sys_errlist[errno], string)
```

pa.c

```
/* Copyright (C) 1993,1994 by the author(s).

 This software is published in the hope that it will be useful, but
 WITHOUT ANY WARRANTY for any part of this software to work correctly
 or as described in the manuals. See the ShapeTools Public License
 for details.

 Permission is granted to use, copy, modify, or distribute any part of
 this software but only under the conditions described in the ShapeTools
 Public License. A copy of this license is supposed to have been given
 to you along with ShapeTools in a file named LICENSE. Among other
 things, this copyright notice and the Public License must be
 preserved on all copies.
*/
/*
 * ShapeTools/shape program - pa.c
 *
 * by Juergen.Nickelsen@cs.tu-berlin.de
 *
 * $Header: parser.c[9.0] Fri Aug  6 20:12:26 1993 nickel@cs.tu-berlin.de accessed $
 */
#ifndef lint
static char *AtFSid = "$Header: parser.c[9.0] Fri Aug  6 20:12:26 1993 nickel@cs.tu-berlin.de accessed
#endif

#include <ctype.h>
#include <setjmp.h>
#include "shape.h"
#include "parser.h"
#include "mfiles.h"

/* This variable is exported: */
int parse_errors = 0 ;                /* number of parse_errors */
#ifdef DEBUG
int dump_parsed_definitions ;
#endif

/* local declarations and definitions */

/* several string literals and their length: */
#define STRING_LENGTH(s)        (sizeof(s) - 1)        /* ONLY for literals! */
#define INCLUDE_STRING                "include"
#define INCLUDE_LENGTH                STRING_LENGTH(INCLUDE_STRING)
#define VCLASS_STRING                "vclass"
#define VCLASS_LENGTH                STRING_LENGTH(VCLASS_STRING)
#define VCLASS_SEPARATOR        "::="
#define VCLSEP_LENGTH                STRING_LENGTH(VCLASS_SEPARATOR)
#define RULESEC_STRING                "RULE-SECTION"
#define RULESEC_LENGTH                STRING_LENGTH(RULESEC_STRING)
#define VARSEC_STRING                "VARIANT-SECTION"
#define VARSEC_LENGTH                STRING_LENGTH(VARSEC_STRING)
#define ENDSEC_STRING                "END-"
```

```
#define ENDSEC_LENGTH                STRING_LENGTH(ENDSEC_STRING)


/* constants for the type of a top-level line */
#define UNKNOWN_LINE                  0 /* not recognized */
#define VCLASS_LINE                  1 /* vclass definition */
#define DEPRULE_LINE                  2 /* dependency rule */
#define VARDEF_LINE                  3 /* variant definition */
#define MACRO_LINE                  4 /* macro definition with = */
#define ADDMAC_LINE                  5 /* macro definition with += */
#define SUBMAC_LINE                  6 /* macro definition with -=  */
#define VARMAC_LINE                  7 /* macro definition with := */
#define SELRULE_LINE                  8 /* selection rule */
#define RULESEC_LINE                  9 /* beginning of rule section */
#define VARSEC_LINE                 10 /* beginning of variant section */
#define INCLUDE_LINE                 11 /* "include" line */
#define DOUBLEC_LINE                 12 /* line with double colon */

/* Constants for parse_selection_rule() and parse_variant_definition():
 * Has definition been found in a SECTION (old style) or not. */
#define IN_SECTION                  1
#define NO_SECTION                  0

/* possible components of selection rule or variant names, this is
 * essentially [0-9_A-Za-z] */
#define RULE_OR_VAR_NAME_COMPONENTS \
"0123456789_ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

/* local Functions */

static int recognize_line A((char **linep, struct stringlist **name_list_p)) ;
static void skip_over_rulebody A((void)) ;
static char *get_variant_name A((char **linep)) ;
static struct stringlist *collect_names A((char **linep)) ;
static void parse_rule_or_var_section A((char *line)) ;
static void parse_macro_definition A((char *line, int type,
                                struct stringlist *name_list)) ;
static void parse_vclass_definition A((char *line)) ;
static void parse_dependency_rule A((char *line, struct stringlist *name_list)) ;
static int end_of_section A((char *line, char *section)) ;
static void parse_selection_rule A((char *line,
                                    struct stringlist *name_list,
                                    int in_section)) ;
static void variant_section_body A((char *line)) ;
static void rule_section_body A((char *line)) ;
static void parse_variant_definition A((char *line,
                                        struct stringlist *name_list,
                                        int in_section)) ;
static void parse_include_statement A((char *line)) ;
static int rule_or_variant_body A((char *line)) ;
static void clean_from_comment A((char *line)) ;
static void clean_selrule_from_comment A((char *line)) ;

/* local Variables */
```

```
static MFILE *currentmf ;
static jmp_buf jmp_toplevel ;


/* int parse_file(filename): toplevel parse funtion */

int parse_file(filename)
char *filename ;
{
    /* It does:
       - read in the specified file with mf_readin()
       - for each line at the top-level do the following:
         - skip over leading blanks
         - look at the first character of the line. It may indicate a
           comment line, an empty line, or a line beginning with TAB,
           which is not allowed at top-level.
         - call recognize_line() to determine the type of a non-empty
           line. recognize_line() also reads the "names" (target,
           rule, variant, or macro names) it encounters before seeing
           a "magic token" which identifies the line.
         - call the appropriate parse function for this line. This
           function may read more lines.
         - if an error is found, a parse function may jump back to the
           top-level with longjmp() (usually called by parsing_error()).
           The entry point is saved in jmp_toplevel.

       The current MFILE * and jmp_toplevel are global to this module
       in order to be accessible by the other parse functions. In case
       of an recursive invokation of parse_file() by
       parse_include_statement() they are saved in local variables of
       parse_include_statement().
     */

    char *line ;                /* pointer to current line */
    int line_type ;                /* type of line */
    char *line_begin ;                /* save ptr to beginning of line */
    struct stringlist *name_list ; /* list of "names" collected
                                    * before type of line is
                                    * determined */
#ifdef DEBUG
    dump_parsed_definitions = getenv("SHAPE_PARSER_DUMP") ;
#endif

    /* check if standard input is to be parsed */
    if (strcmp(filename, "-")) {
        currentmf = mf_readin(filename) ;
    } else {
        currentmf = mf_readin_stdin() ;
    }

    if (currentmf == NULL) {        /* mf_readin() failed */
        return FALSE ;
    }
```

```
/* jump target for parsing_error() */
setjmp(jmp_toplevel) ;

/* top-level loop */
while ((line = mf_nextline(currentmf))) {
    line_begin = line ;
    line = skip_over_blanks(line) ;
    switch (*line) {        /* look at first character */
      case '#':                /* comment or RULE- or VARIANT-SECTION */
        if (*(line + 1) == '%') {
            parse_rule_or_var_section(line) ;
        }                        /* else comment: ignore */
        break ;

      case '\t':                /* tab at beginning only in rule or
                                 * variant bodies allowed (or line
                                 * must be empty) */
        line = skip_over_whitespace(line) ;
        if (*line != '\0' && *line != '#') {
            parsing_error("line with tab character at top-level",
                          line, NULL) ;
        }
        break ;

      case '\0':                /* empty line */
        break ;

      default:
        line_type = recognize_line(&line, &name_list) ;

        /* line now points to first character after magic token,
         * name_list contains names found up to that point in
         * correct order.
         */

        clean_from_comment(line) ;
        switch (line_type) {
          case UNKNOWN_LINE:
            parsing_error("line not recognized", line_begin, NULL) ;
            break ;
          case VCLASS_LINE:
            parse_vclass_definition(line) ;
            break ;
          case DEPRULE_LINE:
            parse_dependency_rule(line, name_list) ;
            break ;
          case VARDEF_LINE:
            parse_variant_definition(line, name_list, NO_SECTION) ;
            break ;
          case MACRO_LINE:
            parse_macro_definition(line, ' ', name_list) ;
            break ;
          case ADDMAC_LINE:
            parse_macro_definition(line, '+', name_list) ;
```

```
                break ;
              case SUBMAC_LINE:
                parse_macro_definition(line, '-', name_list) ;
                break ;
              case VARMAC_LINE:
                parse_macro_definition(line, ':', name_list) ;
                break ;
              case SELRULE_LINE:
                parse_selection_rule(line, name_list, NO_SECTION) ;
                break ;
              case INCLUDE_LINE:
                parse_include_statement(line) ;
                break ;
              case DOUBLEC_LINE:
                parsing_warning("double colon treated as single colon",
                              line, NULL) ;
                parse_dependency_rule(line, name_list) ;
                break ;
              default:
                fatal("This can't happen", "invalid line type at top-level") ;
            }
        }
    }

    return TRUE ;
}


/* parse functions for top-level items */

/* parse an include file */
static void parse_include_statement(line)
char *line ;
{
    char *includefilenames ;        /* rest of line after "include" */
    MFILE *save_mf ;                /* to save currentmf in  */
    jmp_buf save_jmpbuf ;        /* to save jmp_toplevel in */
    int retval ;                /* return value of parse_file() */
    struct stringlist *name_list ; /* names of include files to parse */
    struct stringlist *curname ; /* current include file to parse */
    char incpath[PATH_MAX];        /* construct include file name here */
    char *st_env ;                /* ST_ENV environment */

    line = skip_over_whitespace(line) ;
    includefilenames = expandmacro(line) ;
    name_list = collect_names(&includefilenames) ;

    if (name_list == NULL) {
        parsing_warning("name of include file(s) missing", line, line) ;
    } else {

        /* save global state */
        memcpy (save_jmpbuf, jmp_toplevel, sizeof(jmp_buf)) ;
        save_mf = currentmf ;
```

```
        /* iterate through name_list */
        for (curname = name_list; curname != NULL; curname = curname->next) {
            retval = parse_file(curname->string) ;
            if (retval == FALSE) {
                /* try to look for file in standard shape include path */
                if ((st_env = getenv(ST_ENV)) != 0) {
                    strcpy(incpath, st_env) ;
                    strcat(incpath, ST_ENV_SHAPELIB) ;
                    strcat(incpath, "/") ;
                    strcat(incpath, curname->string) ;

                    /* try again */
                    retval = parse_file(incpath) ;
                }
            }
            if (retval == FALSE) {
                currentmf = save_mf ;
                parsing_warning("could not open include file",
                                line, curname->string) ;
            }
        }
        free_stringlist(name_list) ;

        /* restore global state */
        currentmf = save_mf ;
        memcpy (jmp_toplevel, save_jmpbuf, sizeof(jmp_buf)) ;
    }
}


/* parse dependency rule. Target names are in name_list. */
static void parse_dependency_rule(line, name_list)
char *line ;
struct stringlist *name_list ;
{
    struct stringlist *targets = NULL ;
                                /* targets of rule */
    struct stringlist *deps = NULL ;
                                /* dependents of rule */
    struct stringlist *macros = NULL ;
                                /* macros targets depend on */
    struct stringlist *cmds = NULL ;
                                /* shell commands to build */

    /* names of targets are in name_list, *if* */
    if (name_list == NULL) {
        parsing_error("dependency rule must begin with target", line, NULL) ;
    }
    targets = name_list ;

    /* dependents are following */
    deps = collect_names(&line) ;

    /* line may now point to second ':', then collect macros
```

```
     * the target depends on */
    if (*line == ':') {
        line++ ;
        macros = collect_names(&line) ;
    }

    /* line must point to command, comment, or end of line now */
    switch (*line) {
      case ';':
        /* first command line */
        if (skip_over_whitespace(line + 1) != '\0') {
            cmds = push_stringlist(line + 1, cmds, 0) ;
        }
        break ;
      case '#':                         /* "virtual" end of line... */
      case '\0':                /* and the real one */
        break ;
      default:
        parsing_error("Syntax error in dependency rule", line, NULL) ;
    }

    /* collect command lines for shell script. The shell script ends
     * on a non-comment line that does not begin with a TAB. */
    while ((line = mf_nextline(currentmf)) != NULL) {
        char *first_nonblank = skip_over_whitespace(line) ;

        if (*line != '\t' && *first_nonblank != '#') {
            break ;                /* end of shell script */
        }
        /* give only lines with real commands to the shell */
        if (*first_nonblank != '\0' && *first_nonblank != '#') {
            cmds = push_stringlist(line, cmds, 0) ;
        }
    }

    /* make first non-command line next to be read if ! EOF */
    if (line != NULL) {
        mf_prevline(currentmf) ;
    }

    /* cmds are in still in reverse order! */
    cmds = revert_stringlist(cmds) ;

    queue_dependency_rule(targets, deps, macros, cmds) ;
}



/* collect the names (usually of targets or dependants) from a string.
 * Returned is a stringlist with the names in correct order.
 * Stops at the first character which is neither namecomponent nor
 * whitespace. The pointer linep points to is set to the position
 * where collect_names() stopped collecting.
 */
```

```
static struct stringlist *collect_names(linep)
char **linep ;                          /* pointer to pointer to string */
{
    struct stringlist *names = NULL ; /* list of names */
    struct stringlist *last = NULL ; /* last member */
    struct stringlist *tmp ;

    while (strlen(*linep)) {        /* something is left */
        *linep = skip_over_whitespace(*linep) ;

        /* stop here if no name is found */
        if (is_no_namecomponent(**linep)) {
            break ;
        }

        /* append name to list */
        tmp = (struct stringlist *) check_malloc(sizeof(struct stringlist)) ;
        if (last == NULL) {
            last = names = tmp ;
        } else {
            last->next = tmp ;
            last = last->next ;
        }
        last->next = NULL ;
        last->string = get_name(linep) ;
        last->freeable = 1 ;
    }
    return names ;
}


/* parse selection rule */

static void parse_selection_rule(line, name_list, in_section)
char *line ;                          /* line with head of rule */
struct stringlist *name_list ;        /* list with name of selection rule */
int in_section ;
{
    sb_ptr sb ;                        /* buffer for selection predicates */
    int src_line_no;            /* pos. of sel.rule head in input file */
    char *src_filename_ptr;     /* pointer to name of cur. input file */
    char src_filename_buf[PATH_MAX];
    char *pretty_header;

    /* "pretty-print" selection rule header */
    {
      sb_ptr head_buf;
      struct stringlist *next_str;

      head_buf = check_sbnew (SB_LENGTH);
      for (next_str = name_list; next_str; next_str = next_str->next) {
        head_buf = check_sbcat (head_buf, next_str->string);
      }
      pretty_header = check_strdup (sbstr (head_buf));
```

```
  sbfree (head_buf);
}

/*
 * consider header syntactically correct, if it either consists
 * of a single name, or a single name, followed by a parenthesized
 * parameter list. In the latter case, it will suffice, if the first
 * non-white-space character after the name is a '(', and the last
 * character is a ')'.
 */
{
  register char *open_paren = pretty_header, *close_paren;

  while (open_paren && *open_paren && (*open_paren != '(') &&
         !isspace (*open_paren)) open_paren++;
  while (open_paren && *open_paren && isspace (*open_paren)) open_paren++;
  close_paren = open_paren;
  while (close_paren && *close_paren) close_paren++;
  if (close_paren && (*close_paren == '\0')
      && (close_paren > pretty_header)) {
    close_paren--;
    while (*close_paren && isspace(*close_paren)) close_paren--;
  }
  if (open_paren && *open_paren && (*open_paren != '(')) {
    parsing_error ("rule name with optional argument list required",
                   line, NULL);
  }
  if (open_paren && *open_paren && close_paren && *close_paren &&
      (*close_paren != ')')) {
    parsing_error ("rule name with optional argument list required",
                   line, NULL);
  }
}
mf_locate (line, &src_filename_ptr, &src_line_no);
(void) strcpy (src_filename_buf, src_filename_ptr);

/* check if text follows */
line = skip_over_whitespace(line) ;
if (*line != '\0') {
    parsing_error("newline expected after selection rule header",
                  line, NULL) ;
}

sb = check_sbnew(SB_LENGTH) ; /* prepare string buffer for rule body */

/* collect rule body lines */
while ((line = mf_nextline(currentmf)) != NULL) {
    if (in_section ?
            rule_or_variant_body(line)
          : *line == '\t') {
        clean_selrule_from_comment(line) ;
        sb = check_sbcat(sb, line) ;
    } else {
        break ;
```

```
        }
    }

    /* ... and feed them to vbind. */
    queue_selection_rule(pretty_header, sbstr(sb), src_filename_buf,
                         src_line_no);

    free(name_list) ;
    free(sb) ;

    /* recover from look-ahead */
    if (line != NULL) {
        mf_prevline(currentmf) ;
    }
}


static int rule_or_variant_body(line)
char *line ;
{
    /* In a section there are three ytpes of lines:
         1. empty or comment-only line (possibly end of section)
                --> return FALSE
         2. body of rule or variant
                --> return TRUE
         3. header of rule or variant, this is something like
             "^[ \t]+[_0-9a-zA-Z]+[ \t]+:[ \t]+\(#.*\)*$"
                --> return FALSE
         4. vclass definition, beginning with "[ \t]vclass[ \t]"
                 --> return FALSE
       The choice between 2. and 3. is difficult. I decide to return
       TRUE if the header pattern is not matched.
     */

    char *word_beginning ;        /* beginning of first word in line */

    line = skip_over_whitespace(line) ;
    if (*line == '\0' || *line == '#') {
                                  /* empty or comment-only */
        return FALSE ;
    }

    word_beginning = line ;
    while (*line != '\0' && strchr(RULE_OR_VAR_NAME_COMPONENTS, *line)) {
                                  /* skip over possible header */
        line++ ;
    }
    if (line - word_beginning == VCLASS_LENGTH &&
        (*line == '\t' || *line == ' ') &&
        !strncmp(word_beginning, VCLASS_STRING, VCLASS_LENGTH)) {
        /* is a vclass definition */
        return FALSE ;
    }
```

```
    line = skip_over_whitespace(line) ;
    if (*line != ':') {
        return TRUE ;                /* does not match header pattern */
    }

    /* if nothing but comment follows, header pattern is matched */
    line = skip_over_whitespace(line + 1) ;
    if (*line == '\0' || *line == '#') {
        return FALSE ;               /* matched */
    }

    return TRUE ;                /* not matched */
}




/* parse variant definition */

static void parse_variant_definition(line, name_list, in_section)
char *line ;                         /* line with variant name */
struct stringlist *name_list ;       /* must contain only variant name */
int in_section ;
{
    char *variant_name ;
    struct stringlist *macro_list = NULL ;
                                /* list of macros for variant */

    if (name_list == NULL || name_list->next != NULL) {
        parsing_error("exactly one variant name required", line, NULL) ;
    }

    /* check if text follows */
    line = skip_over_whitespace(line) ;
    if (*line != '\0') {
        parsing_error("newline expected after variant definition header",
                    line, NULL) ;
    }

    variant_name = name_list->string ;
    free(name_list) ;

    /* feed macro definitions to shape */
    while ((line = mf_nextline(currentmf)) != NULL) {
        if (in_section ?
                rule_or_variant_body(line)
              : *line == '\t') {
            clean_from_comment(line) ;
            macro_list = push_stringlist(skip_over_whitespace(line),
                                         macro_list, 0) ;
        } else {
            break ;
        }
    }
```

```
    queue_variant_definition(variant_name, macro_list) ;

    /* recover from look-ahead */
    if (line != NULL) {
        mf_prevline(currentmf) ;
    }
}


/* parse rule- or variant-section */

static void parse_rule_or_var_section(line)
char *line ;
{
    char *section_name ;        /* pointer to name of section */

                                /* functions to parse the body of a
                                 * rule resp. variant section */
    void (*body_parser)A((char *line)) ; /* pointer to hold the address
                                          * of the appropriate routine */
    char *backup = line ;        /* line before current. Must have an initial
                                 * value because it is given as argument to
                                 * parsing_error() even if no line has been
                                 * read here (in case of EOF) */
    jmp_buf save_jmpbuf ;        /* save state of top-level loop */



    line = skip_over_whitespace(line + 2) ;
    section_name = get_name(&line) ; /* get name of section */

    if (!strcmp(section_name, RULESEC_STRING)) {
        /* this is a rule section */
        body_parser = rule_section_body ;
    } else if (!strcmp(section_name, VARSEC_STRING)) {
        /* this is a variant section */
        body_parser = variant_section_body ;
    } else {
        return ;                /* must be comment */
    }

    /* a section has its own top-level, so it needs
     * an own recovery point for parsing_error() */
    memcpy (save_jmpbuf, jmp_toplevel, sizeof(jmp_buf)) ;
    setjmp(jmp_toplevel) ;

    /* section top-level loop */
    do {
        line = mf_nextline(currentmf) ;
        if (line == NULL) {        /* end of file reached */

            /* restore global state */
            memcpy (jmp_toplevel, save_jmpbuf, sizeof(jmp_buf)) ;
            parsing_error("end of file in section", backup, section_name) ;
```

```
        }
        backup = line ;
        line = skip_over_whitespace(line) ;

        /* skip empty and comment lines */

        switch (*line) {
          case '\0':
          case '#':
            break ;
          case '\t':
            parsing_error("line beginning with TAB at top-level", line,
                          section_name) ;
            break ;
          default:
            (*body_parser)(line) ;
        }
    } while (!end_of_section(line, section_name)) ;
    free(section_name) ;

    /* restore global state */
    memcpy (jmp_toplevel, save_jmpbuf, sizeof(jmp_buf)) ;
}


/* parse a body part of a rule section */

static void rule_section_body(line)
char *line ;
{
    struct stringlist *name_list = NULL ;

    /* get name of rule */
    name_list = push_stringlist(get_name(&line), name_list, 1) ;

    line = skip_over_whitespace(line) ;
    if (*line != ':' || *skip_over_whitespace(++line) != '\0') {
        parsing_error("syntax error in selection rule header", line, NULL) ;
    }
    parse_selection_rule(line, name_list, IN_SECTION) ;
}


/* parse a body part of a variant section */

static void variant_section_body(line)
char *line ;
{
    char *name ;                /* variant name or "vclass" */
    struct stringlist *name_list ; /* to feed it to
                                    * parse_variant_definition() */

    name = get_name(&line) ;        /* variant name or "vclass" */
```

```
    if (!strcmp(name, VCLASS_STRING)) {
        parse_vclass_definition(line) ;
    } else {
        name_list = push_stringlist(name, NULL, 1) ;
        line = skip_over_whitespace(line) ;
        if (*line != ':' || *skip_over_whitespace(++line) != '\0') {
            parsing_error("syntax error in variant definition header",
                        line, NULL) ;
        }

        parse_variant_definition(line, name_list, IN_SECTION) ;
    }
}


/* determine if line is the end line of section */

static int end_of_section(line, section)
char *line ;                        /* line with possible "#% END-..." */
char *section ;                        /* name of section as string */
{
    if (strncmp(line, "#%", 2)) {
        return FALSE ;                /* no "#%" at beginning */
    }

    line = skip_over_whitespace(line + 2) ;
    if (strncmp(line, ENDSEC_STRING, ENDSEC_LENGTH)) {
        return FALSE ;                /* no "END-" */
    }

    if (strncmp(line + 4, section, strlen(section))) {
        return FALSE ;                /* name of appropriate section missing */
    }

    line = skip_over_name(line) ;
    if (*line != '\0' && !isspace(*line)) {
        return FALSE ;                /* no whitespace after name of section */
    }

    line = skip_over_whitespace(line) ;
    if (*line != '\0') {
        parsing_warning("trailing text on end-section line ignored",
                    line, section) ;
    }

    return TRUE ;                /* it IS the end of the section (at last) */
}


/* parse_macro_definition(char *line, char type): parse macro definition */

static void parse_macro_definition(line, type, name_list)
char *line ;
char type ;
```

```
struct stringlist *name_list ;
{
  int mac_def_type = DYNAMIC;

  if (name_list == NULL || name_list->next != NULL) {
    parsing_error("exactly one macro name required", line, NULL) ;
  }

  switch (type) {
  case '-':
    parsing_error("macros with \"-=\" not yet supported", line, NULL) ;
    break ;
  case ':':
    mac_def_type = ONCE;
    break ;
  case '+':
    mac_def_type = APPEND;
    break ;
  case ' ':
    break ;
  }

  define_macro(name_list->string, line, mac_def_type) ;
  free_stringlist(name_list) ;
}


/* parse vclass definition. "vclass" has been eaten by recognize_line() */

static void parse_vclass_definition(line)
char *line ;
{
    char *name ;                /* name of vclass */
    struct stringlist *elems = NULL ; /* list of elements */

    name = get_name(&line) ;        /* get name of vclass */
    line = skip_over_whitespace(line) ;

    if (strncmp(line, VCLASS_SEPARATOR, VCLSEP_LENGTH)) { /* "::=" */
        parsing_error("Syntax error in vclass definition", line, NULL) ;
    }

    line += VCLSEP_LENGTH ;
    line = skip_over_whitespace(line) ;

    if (*line != '(') {
        parsing_error("\"(\" expected in vclass definition", line, NULL) ;
    }

    line++ ;
    /* get first element (at least one must be present) */
    elems = push_stringlist(get_variant_name(&line), elems, 1) ;
    line = skip_over_whitespace(line) ;
```

```
    /* continue until ')' is seen */
    while (*line != ')') {
        if (*line == ',') {
            line = skip_over_whitespace(++line) ;
            elems = push_stringlist(get_variant_name(&line), elems, 1) ;
            line = skip_over_whitespace(line) ;
        } else {
            parsing_error("syntax error in vclass definition", line,
                        "\")\" or \",\" expected") ;
        }
    }
    /* feed definition to shape */
    queue_vclass_definition(name, elems) ;
    if (*skip_over_whitespace(++line)) {
        parsing_warning("trailing text after vclass definition ignored",
                       line, NULL) ;
    }
}



/* The following function is different from the other "skip_over_"s
 * because it is intended to throw away information. */
static void skip_over_rulebody()
{
    char *line ;                /* line to be read */

    /* This is only a bad guess. Since the bodies of selection
     * rule definitions in rule sections and of variant definitions
     * in variant sections need not begin with a tab, we may lose here.
     * This will result in lots of complaints about syntactically wrong
     * ruledef and vardef headers (which are no headers).
     * But to make this of somewhat general purpose, I decide
     * to make this tradeoff.
     * For all things outside sections this works ok.
     */
    while ((line = mf_nextline(currentmf)) != NULL
           && *line == '\t') ;

    if (line != NULL) {
        mf_prevline(currentmf) ;
    }
}



/* parsing_error(char *message, char *ptr, char *option): give
 * error message, skip any lines beginning with tabs and
 * jump to recovery point. extern int parse_errors is incremented. */

void parsing_error(message, ptr, option)
char *message ;                         /* error message */
char *ptr ;                          /* pointer to error position */
char *option ;                          /* additional message */
```

```
{
    char *fname ;                /* name of file error occured in */
    int lineno ;                /* number of line */

    parse_errors++ ;                /* increment number of detected errors */

    /* locate position of error */
    if (mf_locate(ptr, &fname, &lineno)) {
        fprintf(stderr, "%s:%d: ", fname, lineno) ;
    }

    /* give message */
    fprintf(stderr, "%s parse error: %s", stProgramName, message) ;
    if (option) {
        fprintf(stderr, " (%s)", option) ;
    }
    fprintf(stderr, "\n") ;

#ifdef DEBUG
    /* determine column where the error was detected */
    line = mf_thisline(currentmf, lineno) ;
    fprintf(stderr, "  Column %d\n", ptr - line) ; /* counts from 0 */
#endif

    /* throw away anything that is not top-level */
    skip_over_rulebody() ;

    /* jump to top-level loop */
    longjmp(jmp_toplevel, 1) ;
}


/* issue a warning */

void parsing_warning(message, ptr, option)
char *message ;                        /* warning message */
char *ptr ;                        /* pointer to place */
char *option ;                        /* additional message */
{
    char *fname ;                /* name of file place is in */
    int lineno ;                /* number of line */

    /* locate place */
    if (mf_locate(ptr, &fname, &lineno)) {
        fprintf(stderr, "%s:%d: ", fname, lineno) ;
    }

    /* issue warning */
    fprintf(stderr, "%s parse warning: %s", stProgramName, message) ;
    if (option) {
        fprintf(stderr, " (%s)", option) ;
    }
    fprintf(stderr, "\n") ;
}
```

```
/* tries to recognize the type of a top-level line.
 * *name_list_p contains a list of names found before the token that
 * determines the line type.
 * *linep points to the first character *after* this token.
 * We don't need to recognize RULESEC_LINE or VARSEC_LINE, because
 * these would have been detected earlier in the top-level loop.
 */

static int recognize_line(linep, name_list_p)
char **linep ;
struct stringlist **name_list_p ;
{
    char *line = *linep ;        /* pointer into line */
    struct stringlist *name_list = NULL ; /* list of names read */
    sb_ptr sb ;                          /* buffer to collect a name */
    char in_quote = 0 ;                  /* if non-zero: current wuoting character */
    int paren_level = 0 ;        /* level of nested parentheses we're in */
    int line_type = UNKNOWN_LINE ; /* don't know yet... */

    /* "include" or "vclass" at beginning of line
     * overrides all other line types,
     * so we can return immediately */
    if (!strncmp(line, INCLUDE_STRING, INCLUDE_LENGTH)
        && isspace(*(line + INCLUDE_LENGTH))) {

        *linep = line + INCLUDE_LENGTH ;
        line_type = INCLUDE_LINE ;
        *name_list_p = NULL ;
        return line_type ;
    } else if (!strncmp(line, VCLASS_STRING, VCLASS_LENGTH)
               && isspace(*(line + VCLASS_LENGTH))) {
        *linep = line + VCLASS_LENGTH ;
        line_type = VCLASS_LINE ;
        *name_list_p = NULL ;
        return line_type ;
    }

    /* In the other case we have to look at each character.
     * Each character that is not whitespace or part of the
     * magic token is saved in a string buffer. If whitespace is
     * found, the string buffer is pushed to the name list.
     *
     * This can *not* be done with get_name(), since some of the
     * characters in the magic tokens are valid name components.
     */

    sb = check_sbnew(SB_LENGTH) ; /* prepare string buffer */

    /* iterate over line while we don't know the type */
    while (*line && line_type == 0) {

        /* handle quoted strings */
        if (in_quote) {
            if (*line == in_quote) {
```

```
            in_quote = 0 ;
        } else {
            sb = check_sbaddc(sb, *line) ;
        }
    } else {
        switch (*line) {
          case '(':
            paren_level++ ;
            sb = check_sbaddc(sb, *line) ;
            break ;
          case ')':
            if (paren_level <= 0) {
                parsing_error("unexpected close parentheses", line, NULL) ;
            }
            paren_level-- ;
            sb = check_sbaddc(sb, *line) ;
            break ;
          case '\'':
          case '\"':
          case '`':
            in_quote = *line ;
            break ;
          default:
            if (paren_level > 0) {
                sb = check_sbaddc(sb, *line) ;
            } else {
                switch (*line) {
                  case '#':
                    *line = '\0' ;
                    break ;
                  case '-':        /* perhaps -= macro */
                    if (*(line + 1) == '=') {
                        line_type = SUBMAC_LINE ;
                        line++ ;
                    } else {
                        sb = check_sbaddc(sb, *line) ;
                    }
                    break ;
                  case '+':        /* perhaps += macro */
                    if (*(line + 1) == '=') {
                        line_type = ADDMAC_LINE ;
                        line++ ;
                    } else {
                        sb = check_sbaddc(sb, *line) ;
                    }
                    break ;
                  case ':':        /* dependency rule, vclass definition,
                                    * := macro, selection rule, or variant
                                    * definition */
                    switch (*(line + 1)) {
                      case ':': /* vclass or :: rule */
                        if (*(line + 2) == '=') {
                            line_type = VCLASS_LINE ;
                            line += 2 ;
```

```
                        } else {
                            line_type = DOUBLEC_LINE ;
                            line++ ;
                        }
                        break ;
                      case '=': /* := macro */
                        line_type = VARMAC_LINE ;
                        line++ ;
                        break ;
                      case '+': /* variant definition */
                        line_type = VARDEF_LINE ;
                        line ++ ;
                        break ;
                      case '-': /* selection rule */
                        line_type = SELRULE_LINE ;
                        line++ ;
                        break ;
                      default: /* dependency rule */
                        line_type = DEPRULE_LINE ;
                    }
                    break ;
                  case '=':        /* normal macro */
                    line_type = MACRO_LINE ;
                    break ;
                  case ' ':        /* ends a name */
                  case '\t':
                    /* if a name is in the string buffer,
                     * save it to the name list */
                    if (sblen(sb) > 0) {
                        /* the string is not unquote()ed since the
                         * quotes are already removed */
                        name_list = push_stringlist(sbstr(sb),
                                                    name_list, 1) ;
                        free(sb) ;
                        /* prepare new string buffer */
                        sb = check_sbnew(SB_LENGTH) ;
                    }
                    break ;
                  default:
                    sb = check_sbaddc(sb, *line) ;
                }
            }
        }
    }
    line++ ;
}

/* save last name to name list */
if (sblen(sb) > 0) {
    name_list = push_stringlist(sbstr(sb), name_list, 1) ;
    free(sb) ;
} else {
    sbfree(sb) ;
}
```

```
    /* name list is backwards yet */
    *name_list_p = revert_stringlist(name_list) ;
    *linep = line ;

    /* any open quotes? */
    if (in_quote) {
        char quote_char[2] ;

        quote_char[0] = in_quote ;
        quote_char[1] = '\0' ;
        free_stringlist(name_list) ;
        parsing_error("non-terminated quoting at end of line",
                      line, quote_char) ;
    }

    /* any open parentheses? */
    if (paren_level > 0) {
        parsing_error("missing close parentheses at end of line", line, NULL) ;
    }

    return line_type ;
}



/* This function differs from get_name in that commas and parentheses
 * are not allowed in variant names. */
static char *get_variant_name(linep)
char **linep ;
{
    char *name_end, *name_beg ;

    name_beg = skip_over_whitespace(*linep) ;
    name_end = skip_over_name_with_stop(name_beg, ",()") ;
    if (name_beg == name_end) {
        parsing_error("name or macro expected", *linep, NULL) ;
    }

    *linep = name_end ;
    return check_strndup(name_beg, name_end - name_beg) ;
}


/* clean trailing comments from a selection rule line
 * quting doesn't matter here, only parentheses
 */
static void clean_selrule_from_comment(line)
char *line ;
{
    int paren_level = 0 ;        /* number of nested () pairs */

    while (*line != '\0') {
        switch (*line) {
          case '(':
```

```
            paren_level++ ;
            break ;
          case ')':
            paren_level-- ;
            break ;
          case '#':
            if (paren_level <= 0) {
                *line = '\0' ;
                return ;
            }
            break ;
          default: ;
        }
        line++ ;
    }
}

/* clean trailing comments from a line */
static void clean_from_comment(line)
char *line ;
{
    while (*line != '\0') {
        if (*line == '#') {
            *line = '\0' ;
            return ;
        }
        if (isspace(*line)) {
            line = skip_over_whitespace(line) ;
        } else if (is_no_namecomponent(*line)) {
            line++ ;
        } else {
            line = skip_over_name(line) ;
        }
    }
}


/* EOF */
```

queues.c

```
/* Copyright (C) 1993,1994 by the author(s).

 This software is published in the hope that it will be useful, but
 WITHOUT ANY WARRANTY for any part of this software to work correctly
 or as described in the manuals. See the ShapeTools Public License
 for details.

 Permission is granted to use, copy, modify, or distribute any part of
 this software but only under the conditions described in the ShapeTools
 Public License. A copy of this license is supposed to have been given
 to you along with ShapeTools in a file named LICENSE. Among other
 things, this copyright notice and the Public License must be
 preserved on all copies.
*/
/*
 * ShapeTools/shape program - queues.c
 *
 * by Juergen.Nickelsen@cs.tu-berlin.de
 *
 * $Header: queues.c[9.0] Tue Jun 29 19:57:47 1993 axel@cs.tu-berlin.de accessed $
 */
#ifndef lint
static char *AtFSid = "$Header: queues.c[9.0] Tue Jun 29 19:57:47 1993 axel@cs.tu-berlin.de accessed $
#endif

#include <ctype.h>
#include "shape.h"
#include "parser.h"

struct deprule
{
    struct stringlist *macros ;        /* list of macros target depends on */
    struct stringlist *deps ;        /* list of dependants */
    struct stringlist *targets ;/* list of targets */
    struct stringlist *cmds ;        /* list of commands */
    struct deprule *next ;        /* pointer to next rule */
} ;

static struct deprule *deprulelist = NULL ; /* list of all dependency rules */
static struct deprule *lastdeprule = NULL ; /* pointer to last entry */

struct selrule {                /* selection rule */
    char *name ;                /* name of rule */
    char *body ;                /* body of rule */
    char *src_file;             /* name of src file containing rule */
    int  src_lineno;           /* lineno of rule-head */
    struct selrule *next ;        /* pointer to next rule */
} ;

static struct selrule *selrulelist = NULL ;
static struct selrule *lastselrule = NULL ;
```

```
struct vclassdef {
    char *name ;
    struct stringlist *elements ;
    struct vclassdef *next ;
} ;

void queue_dependency_rule (targets, deps, macros, cmds)
struct stringlist *targets ;
struct stringlist *deps ;
struct stringlist *macros ;
struct stringlist *cmds ;
{
    struct deprule *rule ;

    rule = (struct deprule *) check_malloc(sizeof(struct deprule)) ;
    rule->targets = targets ;
    rule->deps = deps ;
    rule->macros = macros ;
    rule->cmds = cmds ;

    if (lastdeprule == NULL) {
        lastdeprule = deprulelist = rule ;
    } else {
        lastdeprule->next = rule ;
        lastdeprule = lastdeprule->next ;
    }
    lastdeprule->next = NULL ;
}


void queue_selection_rule(name, body, src_filename, src_lineno)
char *name ;
char *body ;
char *src_filename;
int src_lineno;
{
    struct selrule *rule ;

    rule = (struct selrule *) check_malloc(sizeof(struct selrule)) ;
    rule->name = name ;
    rule->body = body ;
    rule->src_lineno = src_lineno;
    rule->src_file = check_strdup (src_filename);

    if (lastselrule == NULL) {
        lastselrule = selrulelist = rule ;
    } else {
        lastselrule->next = rule ;
        lastselrule = lastselrule->next ;
    }
    lastselrule->next = NULL ;
}
```

```
/* Insert a variant definition into Shape's data structures. */
void queue_variant_definition(name, macros)
char *name ;
struct stringlist *macros ;
{
    /* variant definitions are not really queued, since we need them
     * as early as macro definitions. They are directly written
     * into Shape's data structures instead. */

    struct stringlist *mac ;        /* loops over the "macros" list */
    struct stringlist *pr_mac ; /* loops over the "macros" list, but
                                 * one behind mac. Used for freeing the
                                 * stringlist struct. */
    int curvdef ;                /* loops over variantDefs[] */
    int curmdef ;                /* loops over variantDefs[curvdef].vmacros[] */


    /* Look for empty slot in variantDefs[].
     * If a variant definition with the same name is found,
     * this is considered an error.
     */

    for (curvdef = 0;
         variantDefs[curvdef].name != NULL && curvdef < MAXVARDEFS;
         curvdef++) {

        if (!strcmp (variantDefs[curvdef].name, name)) {
            /* variant name already defined */
            errexit (33, name) ; /* I'll kill him later for 33... */
        }
    }
    if (curvdef == MAXVARDEFS) {
        /* variant table overflow */
        errexit (27, "variant definitions") ;
    }

    /* vardef_end() uses this variable */
    lastVariantDef = curvdef ;

    /* Now we have a free slot in variantDefs[curvdef],
     * so initialize name, vpath, vflags, and vmacros fields
     */
    variantDefs[curvdef].name = name ;
    variantDefs[curvdef].vpath = NULL ;
    variantDefs[curvdef].vflags = NULL ;
    for(curmdef = 0; curmdef < MAXVMACROS; curmdef++) {
        variantDefs[curvdef].vmacros[curmdef] = NULL ;
    }

    /* mark next slot as last+1 with NULL in .name */
    variantDefs[curvdef + 1].name = NULL ;

    /* now insert macro definitions for this variant
     * macros "vpath" and "vflags" get special treatment */
```

```
curmdef = 0 ;                /* number of current macro definition */
for (mac = macros, pr_mac = NULL;
     mac != NULL;
     pr_mac = mac, mac = mac->next) {

    char *macname ;               /* name of macro */
    char *macptr ;               /* pointer into macro definition */
    char *macval ;               /* pointer to macro value */

    if (pr_mac != NULL) {
        free(pr_mac) ;        /* free previous struct stringlist */
    }

    /* get name of macro */
    macptr = mac->string ;
    macname = get_name(&macptr) ;
    /* macptr points to position after name now */
    macptr = skip_over_whitespace(macptr) ;

    /* this shodul be more general some time: check for
     * "=", "-=", "+=", ":=" */
    if (*macptr != '=') {
        parsing_error("\"=\" expected in variant definition",
                      macptr, name) ;
    }
    macval = macptr = skip_over_whitespace(macptr + 1) ;
    /* macptr and macval now point to macro value */

    /* check for special macros vpath and vflags if there is a value */
    if (*macval && !strcmp(macname, "vpath")) {
        char *cp ;               /* keep value to free it later */

        /* This is a very simple heuristic:
         * if vpath has only one component and this is "..",
         * use prev_dir instead */

        if (!strcmp((cp = get_name(&macval)), "..") &&
            /* macval now points behind the first component */
            *skip_over_whitespace(macval) == '\0') {
            variantDefs[curvdef].vpath = check_strdup(prev_dir) ;
        } else {
            /* macval's value is useless now, so use macptr */
            variantDefs[curvdef].vpath = macptr ;
        }
        free(cp) ;
    } else if (*macval && !strcmp(macname, "vflags")) {
        variantDefs[curvdef].vflags = macval ;
    } else {
        /* So this is a "real" variant macro --
         * do we still have a free macro slot? */
        if (curmdef >= MAXVMACROS) {
            fatal("too many variant macro definitions", name) ;
        }
```

```
            /* now insert definition */
            variantDefs[curvdef].vmacros[curmdef++] = mac->string ;
        }
        free(macname) ;
    }
    free(pr_mac) ;

    vardef_end() ;                /* This is a hook from which this variant
                                  * may already be activated */
}


void queue_vclass_definition(name, elements)
char *name ;
struct stringlist *elements ;
{
    /* vclass definitions are not really queued, since we need them
     * as early as macro definitions */

    vclassdef(name, elements) ;
    free_stringlist(elements) ;
}


void commit_selrule_definitions()
{
    struct selrule *srule, *prev_srule = NULL ;
    int backup = atBindDisplayErrors;

    /* selection rules */
    atBindDisplayErrors = TRUE;
    for (srule = selrulelist; srule != NULL; srule = srule->next) {
        atBindAddRule(srule->name, srule->body, srule->src_file,
                      srule->src_lineno) ;
        free(srule->name) ;
        free(srule->body) ;
        if (prev_srule != NULL) free(prev_srule) ;
        prev_srule = srule ;
    }
    atBindDisplayErrors = backup;
    if (prev_srule != NULL) free(prev_srule) ;
    lastselrule = selrulelist = NULL ;
}


void commit_deprule_definitions()
{
    struct deprule *drule, *prev_drule = NULL ;

    /* dependency rules */
    for (drule = deprulelist; drule != NULL; drule = drule->next) {
        sb_ptr sb ;                /* string buffer to accumulate rule */
        struct stringlist *tmp ; /* pointer into rulep->targets,
                                  * rulep->deps, and rulep->cmds */
```

```
        sb = check_sbnew(SB_LENGTH) ;

        /* copy targets to string buffer */
        for (tmp = drule->targets; tmp != NULL; tmp = tmp->next) {
            char *xtmp = expandmacro (tmp->string);
            while (xtmp && *xtmp && isspace(*xtmp)) xtmp++;
            sb = check_sbcat(sb, xtmp) ;
            sb = check_sbcat(sb, " ") ;
        }
        /* separator */
        sb = check_sbcat(sb, ": ") ;
        /* copy dependants to string buffer */
        for (tmp = drule->deps; tmp != NULL; tmp = tmp->next) {
            sb = check_sbcat(sb, tmp->string) ;
            sb = check_sbcat(sb, " ") ;
        }
        if (drule->macros) {
            sb = check_sbcat(sb, ": ") ;
            for (tmp = drule->macros; tmp != NULL; tmp = tmp->next) {
                sb = check_sbcat(sb, tmp->string) ;
                sb = check_sbcat(sb, " ") ;
            }
        }

        /* give rule to shape */
        ruledef(sbstr(sb)) ;
        sbfree(sb) ;

        /* add commands to rule */
        for (tmp = drule->cmds; tmp != NULL; tmp = tmp->next) {
            rulecont(tmp->string) ;
        }

        ruleend() ;
        free_stringlist(drule->targets) ;
        free_stringlist(drule->deps) ;
        free_stringlist(drule->macros) ;
        free_stringlist(drule->cmds) ;
        if (prev_drule != NULL) free(prev_drule) ;
        prev_drule = drule ;
    }
    if (prev_drule != NULL) free(prev_drule) ;
    lastdeprule = deprulelist = NULL ;
}


void commit_definitions()
{
    commit_selrule_definitions() ;
    commit_deprule_definitions() ;
}
```

Anhang D: ShapeTools

Die nachfolgende Beschreibung von ShapeTools ist entnommen aus der Ankündigung von ShapeTools 1.3 (Mai 1992) und wurde verfaßt von Axel Mahler.

What is ShapeTools ?

The shape toolkit — we call it ShapeTools — is an integrated set of UNIX commands to establish effective control over the intricacies of the change process during software development. It consists of six subsystems:

- Source Repository — the Attributed File System (AtFS) allows transparent access to regular UNIX files as well as versions of files. Files and versions can be associated with attributes of any length and any type. The repository can be accessed from the Shell command level as well as via a C-library interface.
- Version Control — a set of programs to save, restore, list, find, view, compare, and administer versions of files in various ways. File- and version attributes can be set, altered, and queried. Versions and files can be retrieved and accessed by name or by arbitrary attribute patterns.
- Build Driver — the shape program is a Makefile compatible build driver. Programs (and other aggregates) can be build from regular files and/or file versions that reside in the AtFS repository. The component versions that are made part of a configuration are dynamically selected according to version selection rules. Shape also maintains a derived object cache that accounts for a significant build speedup in cooperative software development projects.
- Emacs Interface — A "shapetools-mode" has been made part of the ShapeTools-1.3 distribution. It allows to browse the object repository in "dired"-style, and to perform various actions on files and versions in pseudo direct manipulation. One may find this interface useful or not, but it demonstrates how ShapeTools functions can be transparently integrated into different development environments.
- Release Management System — ShapeTools' very own release management system has been documented, and included into the ShapeTools-1.3 distribution. It supports the management of multi-person, workstation network based development projects, with a tree-structured source organization. A library of predefined version selection rules, variant definitions, standard targets, and Makefile- and Shapefile templates that is shared among the members of a project team, makes writing of control files trivial, and unobtrusively enforces certain standard procedures and project conventions (such as locking policies, release procedures on component and system level, or product naming conventions).

  Although probably useful for many small and medium software projects, the ShapeTools release management system should not be seen as a product per se. The RMS demonstrates how higher order, project oriented functionality is implemented from lower level SCM toolkit functionality. Other projects may want to develop their own RMS or alter the supplied one.
- Tutorial — A somewhat extensive tutorial helps to get started with ShapeTools and shows how to approach problems in the troublesome field of software configuration management.

It's a toolkit! One of ShapeTools' principal design goals was to develop a software configuration management toolkit that can be easily integrated into various development environments. The individual programs have been carefully designed to fit the UNIX tool philosophy of orthogonal tools that can be easily combined via pipes or command scripts. The integration of SCM functionality in other types of environments (e.g. with graphical UIs) is straightforward.