Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Interner Bericht

Entwicklung von Nassi: Darstellung von Algorithmen mit Nassi-Shneiderman-Diagrammen

Thomas Eickermann, Wolfgang Frings, Anke Häming und Birgit Reuter

FZJ-ZAM-IB-9907

FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Entwicklung von Nassi: Darstellung von Algorithmen mit Nassi-Shneiderman-Diagrammen

Thomas Eickermann, Wolfgang Frings, Anke Häming und Birgit Reuter

FZJ-ZAM-IB-9907

Juni 1999

(letzte Änderung: 17. Juni 1999)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Bei der Beschreibung von Algorithmen ist die graphische Darstellung mit Hilfe von Diagrammen ein übliches Hilfsmittel. Häufig werden hier Nassi-Shneiderman-Diagramme oder auch Flußdiagramme benutzt.

Das in dieser Dokumentation beschriebende Programm *Nassi* überführt Programmteile, die in einer Programmiersprache oder in Pseudo-Code verfaßt sind, in eine solche Darstellung.

Die Entwicklung des Programms *Nassi* wurde durch die im ZAM durchgeführte Ausbildung der Mathematisch-technischen Assistenten angestoßen. In dieser Ausbildung spielt unter anderem das Erlernen von Algorithmen eine zentrale Rolle. Nicht nur für die Erstellung von Lehrtexten [2] sondern auch für die Programmdokumentation und -entwicklung wäre ein Tools zur automatischen Erstellung von Nassi-Shneiderman-Diagrammen hilfreich. Das Zeichnen dieser Diagramme ist eher mühevoll und aufwendig.

Die erste Version von *Nassi* ist 1988 unter dem Großrechner-Betriebssystem VM/CMS in der Programmiersprache Pascal entwickelt worden. Nach der Ablösung der Mainframes im ZAM stand die Portierung von *Nassi* auf die Plattform Unix an. Da Pascal sowohl als Implementationssprache

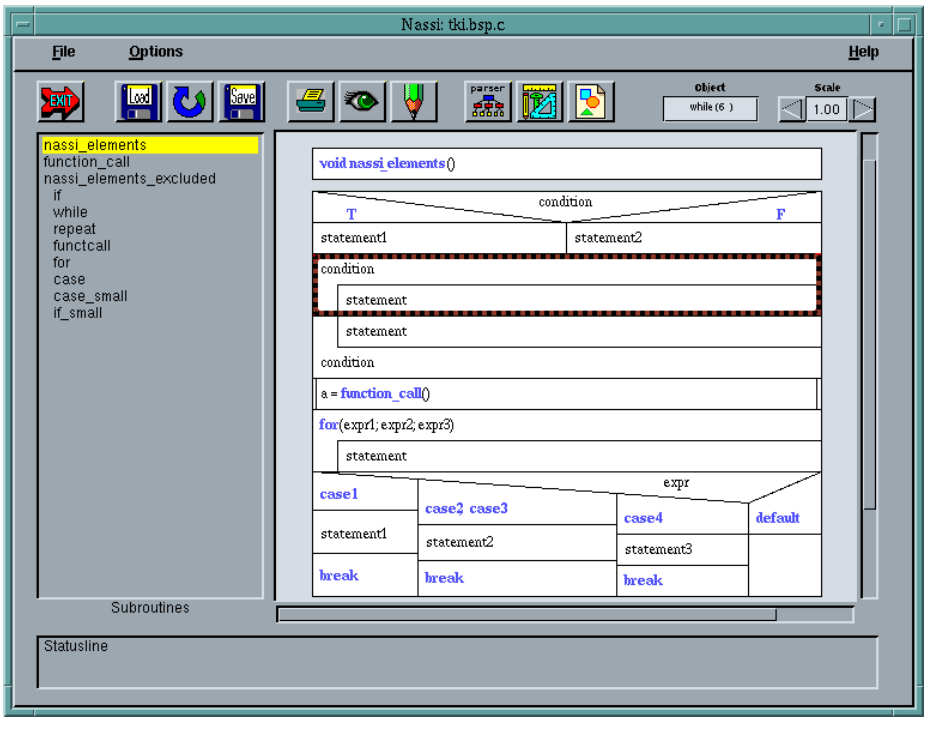


Abbildung 1.1: Oberfläche von Nassi (Unix-Version unter X11)

als auch als Eingabesprache von Nassi nicht mehr aktuell war, ist eine Neu-Implementierung unter Unix vorgezogen worden.
Ziele waren neben der Nutzung der unter Unix vorhandenen Tools im Parser-Bereich auch eine modernere graphische Benutzerschnittstelle und die leichte Erweiterbarkeit des Programms. Eine modulare Implementierung mit klaren Schnittstellen vereinfacht dabei die Erweiterbarkeit. Die hier beschriebene Version von *Nassi* ist hauptsächlich im Sommer 1996 unter der Mitarbeit der Autoren entstanden. Im darauffolgenden Jahr ist Nassi um ein Lizenz-Management erweitert worden und *Nassi* wird seitdem als Shareware [1] angeboten.

Sicherlich spielt *Nassi* in der großen Welt der SE-Werkzeuge nur eine kleine Rolle. Aber gerade in der Ausbildung und Programmdokumentation, bei der sich der Einsatz von SE-Werkzeugen nicht lohnt, kann *Nassi* als ein kleines Hilfsmittel einen Einsatzschwerpunkt finden.

Diese Dokumentation soll in erster Linie einen Überblick über die Entwicklung von *Nassi* geben. Da die Entwicklung der Unix-Version unabhängig von der bestehenden Großrechner-Version sein sollte, konnten hier Prinzipien aus dem Bereich des Software-Engeneerings angewandt werden. Im Design schlägt sich das durch den modularen Entwurf und in der Dokumentation der Entwicklung durch die einzelnen Schritte Analyse, Entwurf und Implementierung nieder. Die folgenden Kapitel beziehen auf diese Schritte.

Der Entwurf, der die Architektur der Programms definiert, ist dabei der aufwendigste Teil. Hier wurde deshalb zwischen einem groben Entwurf auf der Ebene der Module und dem Feinentwurf innerhalb der Module unterschieden. Die Implementierung kann nur an Beispielen den Source-Code von *Nassi* darstellen. Die vollständige Implementierung beläuft sich immerhin auf über 25000 Zeilen C-Code.

Im Kapitel Benutzung wird die englische Version der Benutzunghinweise wiedergegeben, die auch als Technische Kurzinformation TKI-0305 und als Online-Hilfe zur Verfügung steht.

Kapitel 2

Analyse

Das Programm *Nassi* soll aus bestehenden Programmtexten Nassi-Shneiderman-Diagramme generieren. Die interaktive Erstellung von Programmen, die von vielen SE-Werkzeugen angeboten wird, soll bei *Nassi* nicht im Vordergrund stehen. Viel mehr soll *Nassi* in der Dokumentation und Betrachtung von fertigen Programmen eingesetzt werden. Daher ist der Schwerpunkt bei der Implementierung auf die Parser der Eingabesprachen und eine qualitativ hochwertige selbständige Anpassung des Layouts zu legen. Der graphische Editor von Nassi soll nur die Bearbeitung des Layouts ermöglichen.

Als Eingabe sollen sowohl Programme in verschiedenen Programmiersprachen als auch Pseudo-Code möglich sein. Als Ausgabe soll neben der Anzeige am Bildschirm und die Ausgabe im PostScript-Format auch Formate von Graphik-Editoren (z.B. Tgif) unterstützt werden.

Neben dem interaktiven Betrieb soll auch eine vollständig von außen gesteuerte Konvertierung von Programmtexten möglich sein. Zusätzlich zu Konfigurationsdateien von *Nassi* muß dann eine Möglichkeit vorhanden sein, innerhalb des Programmtextes Layoutanweisungen abzuspeichern.

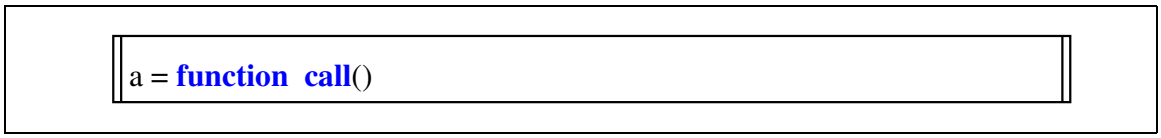
2.1 Nassi-Shneiderman-Diagramme

Nassi-Shneiderman-Diagramme bilden die einzelnen Strukturelemente einer Sprache in einer graphischen Form dar. Die Diagramme sollen, von oben nach unten gelesen, den Ablauf des Algorithmus darstellen. So teilen z.B. Fallunterscheidungen die Bereiche horizontal in zwei Hälften auf, von der nur einer nach unten durchlaufen werden kann. Schleifen umklammern Bereichen und deuten so an, daß der enthaltene Bereich mehrmals durchlaufen wird.

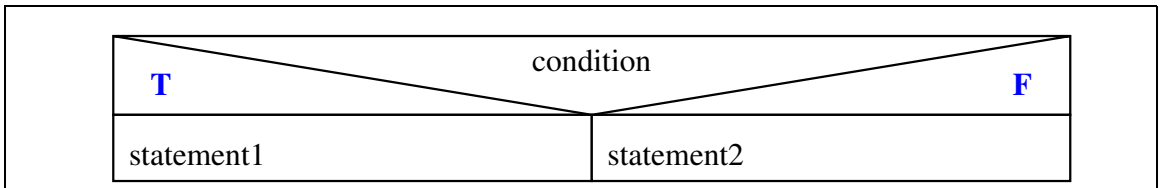
Sequentielle Anweisung:

Anweisung 1
Anweisung 2
Anweisung 3

Jede logische Anweisung im Programmtext erzeugt im Diagramm eine neue Zeile, die mit einem Kasten umschlossen ist. Die Anweisungen werden wie im Programm hintereinander ausgeführt. Aufrufe von Funktionen oder Unterprogrammen werden mit seitlichen Doppellinien gekennzeichnet. Ein Sonderfall stellen dabei Funktionsaufrufe innerhalb der Bedingung einer Fallunterscheidung dar.

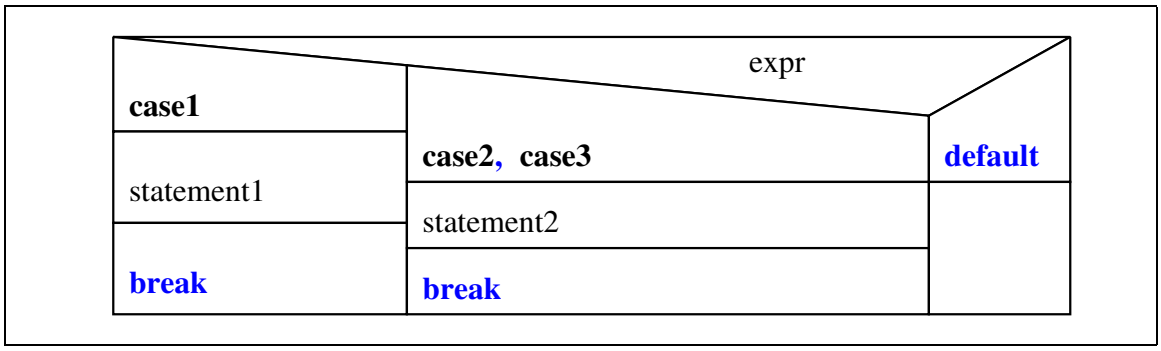


Fallunterscheidungen, wie z.B, `IF` oder `CASE` (Einfach- oder Mehrfachverzweigungen), teilen das Diagramm in mehrere Spalten auf, von denen immer nur eine durchlaufen wird.



Die Bedingung, die entscheidet, ob der Wahr- oder der Falschzweig durchlaufen wird, wird innerhalb des oberen Dreiecks gedruckt.

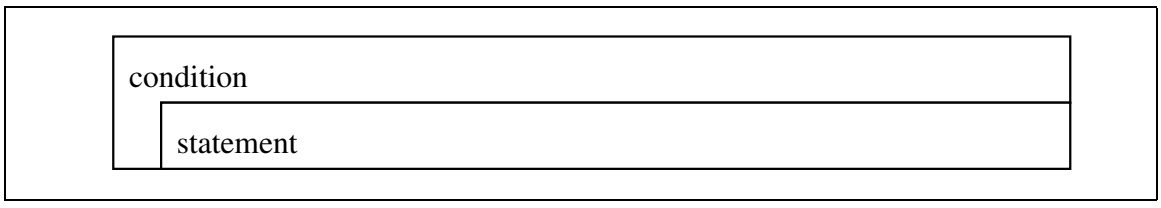
Bei einer Mehrfachverzweigung wird der zu durchlaufende Zweig durch den aktuellen Wert einer Variable bestimmt. Trifft kein Wertebereich zu, wird der Otherwise/Else-Zweig durchlaufen.



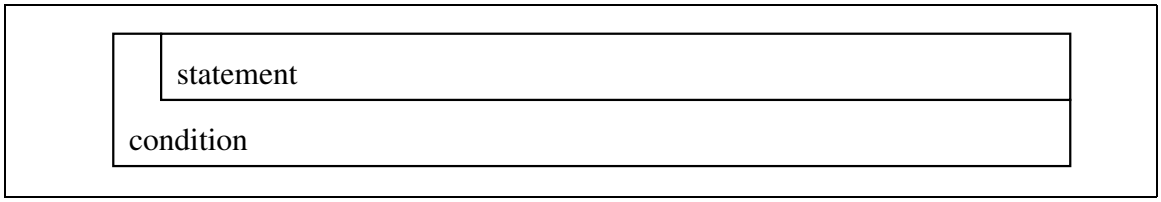
In gewisser Weise handelt es sich um mehrfach geschachtelten IF-Bedingungen (ELSEIF) auch um Mehrfachverzweigungen. Die Darstellung erfolgt dann aber in der einfachen Weise.

Schleifen führen einen Block von Anweisungen mehrfach aus. Man unterscheidet zwischen den Schleifen mit Anfangsbedingung, den Schleifen mit Ende-Bedingung und den FOR-Schleifen, bei denen die Anzahl der Durchläufe schon vorher bekannt ist. Im Nassi-Shneiderman-Diagramm werden diese Schleifen durch eine Umklammerung des Blockes ausgedrückt.

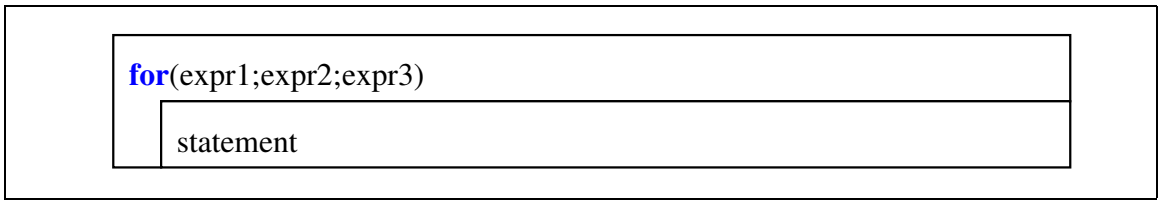
Bei der Schleife mit Anfangsbedingung erscheint die Bedingung als oberste Anweisung.



Eine Schleife mit Ende-Bedingung wird folgendermaßen dargestellt:



Eine FOR-Schleife stellt im Prinzip eine While-Schleife dar, in der Initialisierungsteil vor der Schleife ausgeführt, die Bedingung in der While-Bedingung überprüft und die Inkrement-Funktion zum als letzte Anweisung der Schleife ausgeführt wird.



Im Diagramm sollte auf eine solche Zerlegung verzichtet werden und die For-Schleife mit dem gleichen Symbol wie die While-Schleife dargestellt werden.

Eine besondere Art der Schleifen ist die Schleife mit innenliegender Abbruchbedingung. Das heißt die Schleife wird mitten im Anweisungsblock beendet. Die Darstellung solcher Schleifen ist nicht normiert.

An allen Stellen, an denen eine Anweisung steht, kann auch ein Anweisungsblock stehen, der mehrere Anweisungen zu einer Anweisung bündelt. Eine Anweisung kann wiederum aus den hier beschriebenen nicht-elementaren Strukturelementen bestehen, so daß eine Schachtelung von Schleifen und Verzweigungen möglich ist.

Nicht-strukturierte Anweisungen, wie z.B. Goto-Anweisungen haben keinen Gegenpart in den Nassi-Shneiderman-Diagrammen, da diese im Gegensatz zu den Flußdiagramme für die strukturierte Programmentwicklung definiert worden sind. Auch bei der strukturierten Programmierung kann ein Goto sinnvoll sein (z.B. im Fehlerfall). In den Nassi-Shneiderman-Diagrammen wird das Goto als normale Anweisung dargestellt.

Probleme und Festlegungen

Die bei der Analyse entdeckten Probleme beziehen sich zum größten Teil auf das Layout der Diagramme. Hier müssen entsprechende Festlegungen getroffen werden.

1. **Leerer Zweig bei einer Fallunterscheidung**
   Sowohl der Then- als auch der Else-Zweig der Fallunterscheidung kann leer sein. Für das Nassi-Shneiderman-Diagrammen wird festgelegt, daß auch ein solcher Zweig gezeichnet wird. Die entsprechende Spalte enthält keine Anweisung. Fehlen beide Zweige, wird die Anweisung nicht gezeichnet.
2. **Funktionsaufrufe in Bedingungen**
   Funktionen können an jeder Stelle eines Ausdrucks aufgerufen werden. Daher ist auch ein Funktionsaufruf innerhalb der Bedingung einer Fallunterscheidung oder einer Schleife möglich. Die Kennzeichnung dieser Strukturelemente durch Doppellinien führt häufig zu unübersichtlichen Diagrammen. Daher sollten die Doppellinien an solchen Stellen entfallen.
3. **Schleife mit innenliegender Abbruchbedingung**
   Diese Schleifen besitzen keine Anfangs- oder Ende-Bedingung. Innerhalb der Nassi-Shneiderman-Diagramme sollen sie daher als While-Schleife mit der Bedingung TRUE dargestellt werden. Die Leave-Anweisung, die die Schleife beendet wird als normale Anweisung innerhalb der Schleife gezeichnet.
4. **Diagrammgröße**
   Die Größe der Diagramme wächst natürlich mit ihrem Inhalt. Für die Darstellung von größeren Diagrammen gibt es mehrere Möglichkeiten:
   (a) Die *Reduzierung der Schriftgröße* kann das Problem nicht wirklich lösen, da die Lesbarkeit und die Darstellung der Strukturelemente die Reduzierung begrenzen.
   (b) Die *Aufteilung auf mehrere Seiten/Teile* wirkt sehr unübersichtlich, da z.B. Schleifen unterbrochen werden und deren Anfangs-Bedingung auf der folgenden Seite nicht mehr zu sehen ist.
   (c) Die *Auslagerung von Teilen des Algorithmus* ist wohl die beste Möglichkeit, da die Struktur des Programms weiterhin erkennbar ist. Dazu wird ein logischer Teil-Block

des Programms in ein einzelnes Diagramm gelegt und an der Stelle im Hauptdiagramm ein Verweis zu diesem Diagramm gedruckt.

Alle drei Möglichkeiten können vom Benutzer von Hand oder auch automatisch vom Programm aus erledigt werden. Der Benutzer hat den Vorteil, daß er die Bedeutung (Semantik) des Programm kennt und daher wahrscheinlich die bessere Aufteilung vornehmen kann.

5. **Layout**
   Die Gestaltung des Layouts der Diagramme ist die zentrale Aufgabe von *Nassi*. Hier sollten die Vorteile ausgespielt werden, die eine automatische Generierung von Diagrammen bietet. Die Strukturelemente sollen an deren Inhalt angepaßt werden, so daß von dem Diagramm immer der kleinste Raum beansprucht wird, ohne daß die Übersichtlichkeit verloren geht. In keinem Fall dürfen innerhalb der Diagramme Freiräume entstehen, die keine Strukturelemente enthalten.
   Das Layout kann über folgenden Parameter beeinflußt werden:

   Schriftgröße Sollte vom Programm nicht automatisch verändert werden. Eine Standard-Schriftgröße sollte für alle Texte des Diagramms gelten.

   Spaltenbreite Die Breite des gesamten Diagramm ist festgelegt (z.B. DIN A4). Bei Schleifen sollte die Spaltenbreite für den Anweisungsblock ein fester Anteil der übergeordneten Breite sein.
   Bei Fallunterscheidungen sollte die Spaltenbreite der einzelnen Zweige an deren Inhalt orientiert sein. Je mehr Anweisungen und Texte innerhalb einer Spalte vorhanden sind desto breiter sollte diese auch sein. Die Summe der Breite aller Spalten muß natürlich die übergeordnete Breite ergeben. Kürzere Spalten müssen auf die entsprechende Länge aufgeweitet werden.

   Zeilenabstand sollte innerhalb der einzelnen Strukturelemente gleich sein. Im Zusammenhang mit den Spalten der Fallunterscheidungen muß dieser ggf. erhöht werden.

   Formatierung der Texte Der Text einer Anweisung sollte im Blocksatz formatiert werden. Wenn möglich sollte eine automatische Trennung bei mehrzeiligen Texten vorgenommen werden. Dazu müssen gerade bei Programmtexten andere oder erweiterte Regeln gegenüber der üblichen Textverarbeitung gelten (z.B. bei Formeln).
   Bei den Bedingungen der Fallunterscheidungen sollte der Text in Dreiecksform gedruckt werden. Gegenüber dem Blocksatz bietet dieses Format eine höhere Platzersparnis.

2.2 Struktur von Programmiersprachen

Programmiersprachen, die strukturierte Programmierung unterstützen, verfügen in der Regel über die in Kapitel 2.1 auf Seite 3 beschriebenen Kontrollstrukturen, Anweisungen und Anweisungsblöcke sowie verschiedene Schleifen, Verzweigungen und Unterprogrammen bzw. Funktionen. Allerdings ist die Ausprägung recht unterschiedlich. In dem ursprünglich als Lehr- und Lernsprache entwickelten PASCAL sind diese Kontrollstrukturen z.B. in fast reiner Form vorhanden. In deutlichem Kontrast dazu steht die Sprache C. Einige der dort vorhandenen Kontrollstrukturen stellen Verallgemeinerungen der Grundtypen dar. Ein Beispiel ist die Mehrfachverzweigung Switch-Case, bei der der Programmfluß durch aufeinanderfolgenden Fälle (Cases) ”durchfällt”, wenn dies nicht durch eine break-Anweisung verhindert wird. Mit break und continue stehen zudem zusätzliche Mechanismen zur Schleifensteuerung zur Verfügung. Offenbar müssen also für jede Programmiersprache, die von *Nassi* als Eingabesprache unterstützt werden soll, gesondert Entscheidungen über die Abbildung ihrer Kontrollstrukturen auf die Nassi-Shneiderman Strukturelemente gefällt werden.

Da ein wesentliches Einsatzgebiet von *Nassi* die Dokumentation ist, sollte auch die Möglichkeit bestehen, Nassi-Shneiderman Diagramme zu erstellen, bei denen der Text in den Strukturelementen frei wählbar ist, also nicht den Regeln einer speziellen Programmiersprache entsprechen muß. Dazu könnte man eine spezielle "Pseudocode"-Sprache entwerfen. Einfacher für den Anwender ist, wenn er die Syntax einer ihm bekannten Sprache nutzen kann und den Text der einzelnen Anweisungen und Bedingungen frei wählt. Neben den Parsern für strukturierte Programmiersprachen sollten also auch "Pseudocode"-Parser für diese Sprachen implementiert werden.

Aufgrund der Modularität des Programms *Nassi* können neue Parser nach Bedarf entwickelt und integriert werden. Die erste Version sollte aber bereits mindestens zwei Parser enthalten. Auf diese Weise verringert sich die Gefahr von Entwurfsfehlern, die auf eine zu starke Fixierung auf die Eigenschaften *einer* Sprache zurückgehen.

Der Einsatz von *Nassi* bei der Ausbildung im ZAM erfordert zudem , daß mindestens die Programmiersprachen C und PASCAL unterstützt werden. Auch im Hinblick auf eine Vermarktung von *Nassi* erscheinen diese beiden Sprachen geeignet: C ist die am weitesten verbreitete strukturierte Sprache und PASCAL wird wegen seiner formalen Strenge immer noch häufig als Lehrsprache eingesetzt.

2.2.1 Die Programmiersprache PASCAL

Wie bereits erwähnt, entsprechen die in PASCAL verfügbaren Kontrollstrukturen im wesentlichen den Nassi-Shneiderman Strukturelementen. Im einzelnen sind dies:

`PROGRAM`	Hauptprogramm
`FUNCTION, PROCEDURE`	Unterprogramm mit/ohne Rückgabewert
`IF THEN ELSE`	einfache Verzweigung
`CASE OR OTHERWISE/ELSE`	Mehrfachverzweigung
`WHILE DO`	Schleife mit Bedingung am Schleifenanfang
`REPEAT UNTIL`	Schleife mit Bedingung am Schleifenende

Darüber hinaus gehen lediglich die folgenden Konstrukte:

`FOR TO/DOWNTO DO`	For-Schleife, spezielle Form der While-Schleife
`WITH DO`	.. wird wie While-Schleife dargestellt
`GOTO`	wird als einfaches Statement dargestellt

Charakteristisch ist die deutliche Strukturierung der Programme. Das Programm und alle Funktionen und Unterprogramme beginnen mit einem Deklarationsteil, in dem Datentypen, Konstanten, Variablen und lokale Funktionen bzw. Unterprogramme definiert werden. Da diese jeweils durch reservierte Schlüsselwörter (Token) eingeleitet werden, sind sie für einen Parser leicht zu identifizieren. Anschließend folgt die Liste der Anweisungen, die im Diagramm dargestellt werden soll. Anweisungen sind dabei durch ein Semikolon getrennt.

Ein Parser, der – wie hier benötigt – lediglich Strukturelemente und Anweisungen identifizieren soll, kann sich daher auf die Analyse des Anweisungsteils beschränken und muß auch den Text der einzelnen Anweisungen nicht weiter aufschlüsseln. Das erlaubt es, eine vereinfachte Grammatik zu definieren, die sowohl PASCAL-Programme korrekt beschreibt, als auch für die Eingabe von Pseudocode geeignet ist.

Eine Besonderheit, die auch in den anderen Modulen berücksichtigt werden muß, sind die bereits erwähnten lokalen Funktionen. Daß ein Programm mehrere (lokale) Funktionen gleichen Namens enthalten kann, bedeutet, daß diese nicht allein über ihren Namen identifiziert werden können.

2.2.2 Die Programmiersprache C

Die Kontrollstrukturen der Programmiersprache C sind denen von PASCAL eng verwandt, haben aber im Detail doch meist eine andere Bedeutung:

`if else`	einfache Verzweigung
`switch case default`	Mehrfachverzweigung. Im Unterschied zu PASCAL beendet ein neues **case** aber nicht den Programmfluß des vorhergehenden. Dies muß mit **break** explizit geschehen.
`while`	Schleife mit Bedingung am Schleifenanfang
`do while`	Schleife mit Bedingung am Schleifenende. Man beachte, daß es "while" und nicht "until" heißt.

Darüber hinaus gehen die folgenden Konstrukte:

`for`	eine For-Schleife, wird wie While-Schleife dargestellt. Im Unterschied zu PASCAL können beliebige Ausdrücke für Initialisierung, Iteration und als Abbruchkriterium angegeben werden.
`goto`	wird wie Statement dargestellt
`break`	verläßt die aktuelle Schleife, wird wie Statement dargestellt
`continue`	beendet den aktuellen Schleifendurchlauf, wird wie Statement dargestellt

Insbesondere die häufig genutzten Sprunganweisungen **break** und **continue** sprengen den strengen Rahmen der strukturierten Programmierung.

Im Unterschied zu PASCAL ist die C-Syntax nicht kontextfrei. So kann etwa `FILE *in;` sowohl eine Deklaration (der Variablen `in` vom Typ `FILE *` als auch eine Anweisung (Multiplikation der Variablen `FILE` und `in`) sein, je nachdem, ob `FILE` zuvor als Datentyp oder als Variable deklariert wurde. Das bedeutet, daß ein C-Parser Variablen- und Typ-Deklarationen registrieren und berücksichtigen muß. Auch ist eine genaue Analyse der einzelnen Anweisungen erforderlich, um diese von Deklarationen unterscheiden zu können.

Eine weitere Komplikation stellt der zu C gehörige Präprozessor dar. Makros, Include-Direktiven und bedingte Kompilierung haben zur Folge, daß der Quelltext für sich allein das Programm noch nicht vollständig beschreibt, insbesondere da Makros auch in der Kommandozeile des Compilers definiert werden können.

Dies alles schließt aus, daß man, wie bei PASCAL, eine vereinfachte Syntax definieren kann, die sich sowohl zum Parsen von echten C-Programmen als auch zum Parsen von C-Pseudocode eignet. Aufgrund des anvisierten Einsatzes als Dokumentationswerkzeug und der erheblichen Komplexität eines echten C-Parsers soll daher zunächst nur ein C-Pseudocode Parser entwickelt werden. Durch minimale Erweiterungen bringt dieser aber auch bei der Bearbeitung echter C-Quellen gute Ergebnisse:

- Um Funktionsdefinitionen sicher von Typdeklarationen unterschieden zu können, muß außerhalb der Funktionsblöcke relativ streng geparst werden. Hier ist kein Konflikt mit Pseudocode zu erwarten, da Pseudocode keine solchen Deklarationen enthalten sollte.
- Der Text einzelner Statements wird nicht weiter analysiert. Das erlaubt Pseudocode, hat aber zur Folge, daß Deklarationen in C-Programmen wie Statements behandelt werden.
- Präprozessoranweisungen werden ignoriert. Das hat bei Include-Direktiven meist keine negativen Auswirkungen, entstellt aber Programme mit bedingter Kompilierung.

Um die beiden zuletzt genannten Einschränkungen aufzuheben, sollte für den Anwender die Möglichkeit bestehen, Anweisungen im Quelltext so zu markieren, daß sie nicht in die Nassi-Shneiderman Diagramme übernommen werden. Hierzu kann der gleiche Mechanismus benutzt werden, der auch bei den unter .. beschriebenen Layout-Direktiven zur Anwendung kommt.

2.2.3 Pseudocode

Insbesondere bei englisch-sprachigem Pseudocode kann es vorkommen, daß der Text reservierte Wörter (for, if, while, until, ...) oder Zeichen (Semikolon, Komma, geschweifte Klammern, ...) enthalten soll. Hier muß eine Möglichkeit der Eingabe geschaffen werden. Ein einfacher und (zumindest für UNIX-Anwender) intuitiver Weg ist das Voranstellen eines Backslashs vor das Zeichen bzw. Schlüsselwort.

Um Umlaute und andere nicht ASCII-Zeichen zu drucken, muß ebenfalls eine Festlegung gewählt werden. Hier wird das TEX-übliche `\"`-Präfix unterstützt.

2.2.4 Direktiven

Wie bereits erwähnt, muß eine Möglichkeit geschaffen werden, Steueranweisungen u.a. für das Generator-Modul, die zusätzliche Informationen zu einzelnen Anweisungen, Gruppen von Anweisungen oder dem gesamten Quelltext enthalten, zu verwalten. Diese (im folgenden Direktiven genannt) könnten entweder in einer gesonderten Datei oder direkt im Quelltext eingebaut werden. Dem Nachteil der zweiten Variante, daß der Quelltext modifiziert werden muß, steht die praktische Undurchführbarkeit der ersten Variante gegenüber. Bei nachträglichen Änderungen des Quelltextes durch den Benutzer würde die Datei mit den Direktiven i.a. unbrauchbar. Direktiven sollten in Kommentare der jeweiligen Sprache eingebettet werden, um deren syntaktische Korrektheit zu erhalten.

Eine einfache, aber hier ausreichende Syntax für die Direktiven ist eine Aufeinanderfolge von *name* = *wert* Paaren. Leerzeichen und Zeilenumbrüche gelten als Trennzeichen. Soll der Wert Leerzeichen enthalten, so kann er in " eingeschlossen werden. Um Direktiven von gewöhnlichen Kommentaren unterscheiden zu können, müssen sie mit einer bestimmten Kennung beginnen. Hier wird `NSC:` (für Nassi Special Comment) benutzt. Sollte die Syntax der Eingabesprache derartig formatierte Kommentare nicht zulassen, müssen andere Lösungen gefunden werden. Für PASCAL und C ist das aber nicht erforderlich.

2.2.5 Parse-Tree eines Programms

Sowohl in C als auch in Pascal ist durch die Reihenfolge der Anweisungen und die Schachtelung der Kontrollstrukturen eine eindeutige hierarchische Beziehung zwischen den Programmteilen gegeben. So kann z.B. die folgende C-Funktion (`dekrement`) in einen Baum (siehe Abb. 2.1 auf der nächsten Seite) zerlegt werden.

```
void* dekrement ( int a )
{
  printf("Anfang a=%d\n",a);
  while(a>1) {
       a--;
       printf("a=%d\n",a);
  }
  printf("Ende a=%d\n",a);
}
```

Die Zerlegung beginnt bei der Funktion selber, die aus dem *Header* (Name, Parameter) und einem *Body* besteht. Der *Header* wird nicht weiter zerlegt[1]. Der Body der Funktion ist der mit geschweif-

[1] Aus der Sicht eines Parser würde eine Zerlegung bis hin zu den einzelnen Token gehen. Dies ist aber für die Darstellung von Nassi-Shneiderman-Diagrammen nicht notwendig. Die Information, daß es sich bei der Zeichenkette um die Parameterliste einer Funktion handelt, reicht für die Positionierung innerhalb des Diagramms aus.

ten umklammerte Rest der Funktion. Dadurch entsteht der erste und oberste Knoten des Parse-Trees. Dieser verweist auf einen Body und enthält die Informationen über die Funktion selber.

Eine Klammerung faßt Anweisungen zu einem Block zusammen. Im Baum spiegelt sich das als ein Knoten wieder, der selbst keine Information trägt, sondern nur auf eine Reihe von weiteren, in diesem Block enthaltenen Anweisungen, zeigt. Diese Liste muß geordnet sein, da innerhalb eines Programms die Anweisungen sequentiell in der Reihenfolge der Kodierung abgearbeitet werden. Im Beispiel verweist der Block-Knoten auf drei Anweisungen: `printf`, `while...` und `printf`. Die erste und letzte Anweisung sind im Sinne von Nassi nicht weiter zerlegbar (Elementaranweisung). Bei der zweiten Anweisung handelt es sich um eine Kontrollstruktur (Schleife), die weiter zerlegt werden muß. Der zugehörige Schleifenknoten besitzt als Attribut die Schleifenbedingung. Die Anweisung, die von der Schleife wiederholt werden soll, wird in einem neuen Knoten gespeichert. Durch die geschweiften Klammer ist dies wieder ein Block mit den beiden Anweisungen Dekrementieren und Ausgabe.

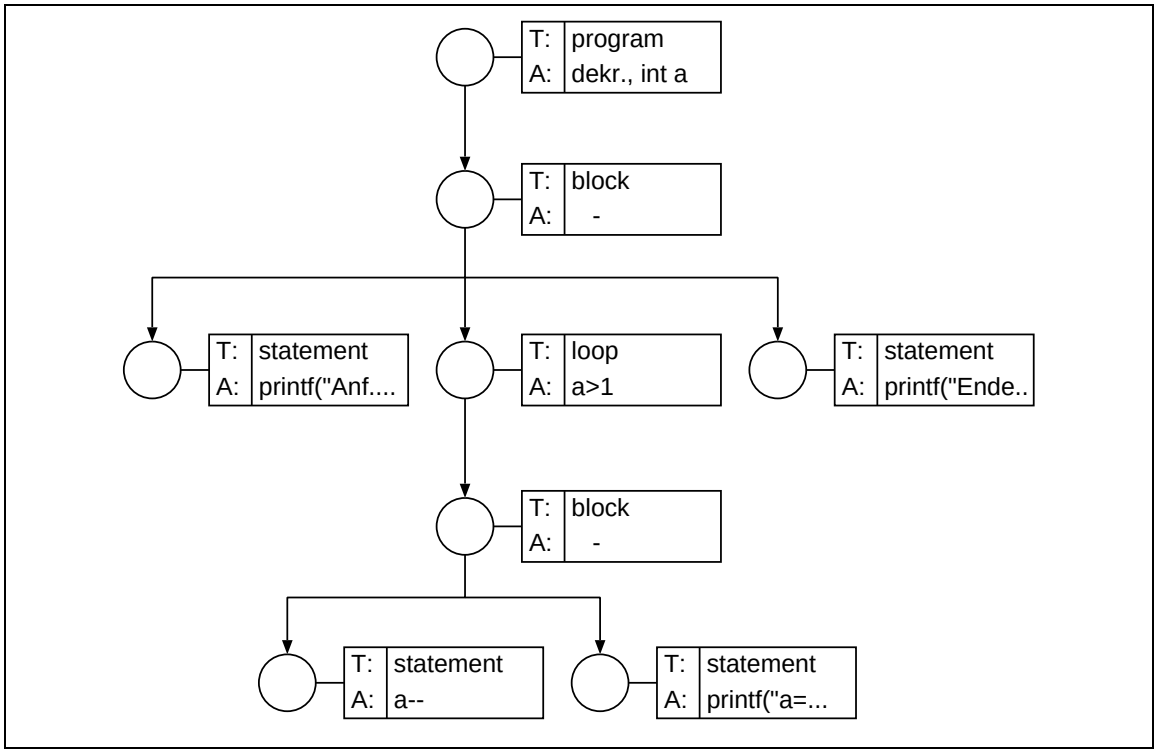


Abbildung 2.1: Parse-Tree eines kleinen C-Programms. Jedem Knoten ist eine Typ (T) und Attribute (H) zugeordnet. Die Anzahl der Söhne, die ein Knoten besitzen kann ist variabel. Wird der Baum mit einer Tiefensuche durchlaufen, ergibt sich wieder der ursprüngliche Programmtext.

Für die Generierung der Nassi-Shneiderman-Diagramme enthält der Baum alle nötigen Informationen. Die Verknüpfung der Knoten spezifiziert die Programmstruktur, die Attribute der Knoten die Programmtexte.

Jedes Programmiersprache, deren Programme eindeutig in solche Parsetrees zerlegt werden können, können auch als Eingabesprache für *Nassi* dienen.

2.3 Weitere Ziele der Entwicklung

Wie schon in der Einleitung aufgezeigt, soll *Nassi* als Hilfe bei der Dokumentation und Analyse von bestehen Programmen dienen. Diese Zielrichtung definiert eine genaue Abgrenzung zu den Case-Tools, mit denen Programmentwicklung betrieben wird. Die Ziele der Entwicklung können zusätzlich zu den bisherigen Ergebnissen der Analyse durch die folgenden drei Abschnitte charakterisiert werden.

2.3.1 Anforderungen an die Oberfläche

Die Oberfläche soll in erster Linie die Struktur des Eingabeprogramms darstellen. Dazu sollen in einem Teil des Fensters immer die vorhandenen Unterroutinen aufgelistet werden. Besteht zwischen den Routinen eine hierarchische Beziehung (z.B. in Pascal), soll diese auch der Auflistung zu sehen sein.

Die strukturelle Änderung von Programmteilen oder das Verändern und Einfügen von Texten ist innerhalb von *Nassi* nicht vorgesehen. Daher ist ein vollständiger Diagramm-Editor nicht nötig. Trotzdem sollte dem Benutzer die Möglichkeit geboten werden, Layout-Änderungen an den Diagrammen vorzunehmen. Dazu zählt z.B. das Ändern der Schriftart, -größe und -format. Abstände sowie Breite und Höhe des Diagramms sollten auch modifizierbar sein. Große, unübersichtliche Diagramme sollen durch das Auslagern oder Verstecken von Teilen verkleinert werden können.

Die Oberfläche sollte die Optionen und Merkmale, die der Benutzer einstellen kann, in einem persönliche Profile abspeichern und wieder einlesen können.

2.3.2 Ausgabeformate

Für die Dokumentation sollen im wesentlichen hochwertige Graphiken genutzt werden, die eine ansprechende Darstellung der Algorithmen mit Hilfe von Nassi-Shneiderman-Diagrammen erlauben. Dazu muß das die für die Textverarbeitung (unter Unix) wichtigen Ausgabemedium PostScript unterstützt werden. Um die Oberfläche und den darin enthaltenen Graphikeditor nicht zu überladen und die Programmentwicklung von *Nassi* überschaubar zu halten, sollen statt eigener Graphikfunktionen, wie z.B. Linien-Ziehen, die Formate anderer Graphik-Editoren unterstützt werden,. Das sind unter Unix *Tgif* und *Xfig*.

2.3.3 Erweiterbarkeit

Die Erweiterbarkeit von *Nassi* sollte für den folgenden Entwurf als eine wichtige Anforderung gelten. Da für fast alle Programmiersprachen Nassi-Shneiderman-Diagramme erstellt werden können, ist hier die einfache Einkopplung eines weiteren Parsers für eine neue Programmiersprache unabdingbar. Gleiches gilt für die Wahl eines neuen Ausgabeformates. Neue Teile sollten auch ohne die Kenntnis des ganzen Programms in *Nassi* eingefügt werden können.

Kapitel 3

Entwurf und Modularisierung

Im der Analyse wurde schon die Erweiterbarkeit und Modularität des Programms in den Vordergrund gestellt. Da das Programm mit mehreren Autoren erstellt und eine gewisse Komplexität erreichen wird, ist eine Aufteilung des Problems in mehrere einzelne Teilprobleme zwingend notwendig. Die Teilprobleme können dann von einzelnen Modulen gelöst werden. Für das Zusammenspiel der Module ist wiederum eine klare Schnittstellendefinition wichtig.

Da ein objektorientierter Entwurf und damit die Implementierung in einer objektorientierten Programmiersprache im Rahmen der Entwicklungszeit nicht durchführbar ist, wird hier ein modularer Entwurf des Programms vorgezogen.

Der folgende Abschnitt führt die Module auf, die in dieser Entwicklungsphase erkennbar sind. Dabei wird hier als Modul eine Anzahl von Funktionen oder Prozeduren und dazugehörige Daten bezeichnet, die ein Teilproblem lösen. Die Schnittstelle der Module untereinander, die hierarchische Beziehung der Module und die Aufgaben, die durch die Module erledigt werden, müssen genauestens festgelegt werden. Die Gestaltung und Implementierung der Module selbst ist dann von der Gesamtkonzeption von *Nassi* unabhängig. Solange die vorgeschriebenen Aufgaben und Schnittstellen eingehalten werden, wäre sogar die Wahl der Implementationssprache frei.

Eine Aufteilung der Aufgabenstellung auf der prozeduralen Ebene würde zu einer zu feine Granularität und zu komplexen Definition der Schnittstellen führen.

Das modulare Konzept bringt beim Entwurf eine weitere Zwischenschicht ein, die eine grobe Aufteilung des Problems erlaubt. Nach der Definition der Module kann die weitere Entwicklungsarbeit weitgehend unabhängig voneinander ausgeführt werden.

Die in diesem Abschnitt beschriebenen Module sind das Ergebnis mehrerer Iterationen bei der Definition. Die Notwendigkeit einiger Module, wie z.B. FontGen, wurde erst bei der Verfeinerung des Generator- und Ausgabe-Modul sichtbar. Die Schnittstelle zwischen den Modulen, die im zweiten Teil dieses Kapitels beschrieben wird, mußte auch an einigen Stellen erst später erweitert werden. Durch die Kapselung der Daten waren die Änderungen bei den anderen Modulen aber sehr gering.

3.1 Module

Folgende Teilprobleme lassen sich aus der Analyse der Problemstellung direkt ableiten und als Module spezifizieren:

Modul	**Parsen der Eingabedaten**
S. 23	Die Strukturelemente und deren hierarchische Beziehung des Eingabeprogramms müssen erkannt und für die weiteren Schritte zwischengespeichert werden.

Modul	**Generieren der Diagramme**
S. 28	Die graphische Darstellung des Diagramm muß aus der Struktur und Inhalt der Anweisungen des Eingabeprogramms berechnet und nach den in der Analyse aufstellten Regeln optimiert werden.

Modul	**Ausgabe**
S. 42	Die aus dem Eingabeprogramm erzeugten Diagramme müssen in den Ausgabeformaten abgespeichert oder direkt zum Drucker geschickt werden.

Diese drei Teilprobleme spiegeln das *von-Neumann-Prinzip* (Eingabe-Verarbeitung-Ausgabe) wieder. Das Prinzip würde für die Batch-Verarbeitung, z.B. als Filter, genügen.

Unter Unix und X11 ist es zeitgemäß ein solches Programm mit einer graphischen Oberfläche auszustatten. Da mit der Oberfläche dem Benutzer aber eine interaktive Beeinflussung des Programm gestattet wird, ist dafür ein zusätzliches Modul **Oberfläche** zur Steuerung notwendig. Diesem Modul sind die drei oben genannten Module untergeordnet. Die Zeitpunkte, an denen die Funktionen der anderen Module abgerufen werden, sind nun nicht mehr durch eine Hauptprogramm gegeben. Stattdessen muß die Oberfläche auf die Aktionen (*Events*) des Benutzer reagieren. Wird z.B. über eine Dialogbox eine neue Eingabedatei ausgewählt, muß die Oberfläche danach den Parser für die Analyse der Programmstruktur aufrufen.

Modul	**Oberfläche**
S. 18	steuert die anderen Module und bietet dem Benutzer Elemente für das Laden von Programmen, generieren der Diagramme und das Abspeichern der Ausgabedaten an.

Die Anzeige der Diagramme am Bildschirm und die damit verbundene Möglichkeit der Layout-Veränderung birgt soviel Komplexität, daß diese Funktionen aus der eigentlichen Oberfläche herausgeschält und in ein eigenes Modul **Graphischer Editor** verlagert werden.

Neben den Elementen der Oberfläche, die zur Steuerung des Programmflussen dienen, sollte eine graphischer Editor zur Anzeige und zur Veränderung des Layouts vorhanden sein:

Modul	**Graphischer Editor**
S. 46	zeigt die berechneten Diagramme an und läßt gewisse Veränderungen des Layouts zu.

Um die Layoutveränderungen an Diagrammen über mehrere Aufrufe von *Nassi* erhalten zu können, müssen diese zusammen mit dem Source-Code abgespeichert werden.

Modul	**Update der Eingabedaten**
S. 27	speichert im Source-Code die Layoutanweisungen als Kommentare ab.

Als letztes Modul sollte hier der Fontgenerator aufgeführt werden, der dem Generator die Zeichengrößen der einzelnen Schriftarten bereitstellt. Sobald man in den Diagrammen nicht nur äquidistante Schriftarten, wie z.B. `Courier`, verwenden will, müssen die Breiten, Höhen und auch Tiefen der einzelnen Zeichen für den Textsatz bekannt sein. Diese Aufgabe wird dem folgenden Modul anvertraut:

Modul	**Fontgen**
S. 58	berechnet die Zeichengrößen und stellt diese dem Generator und Ausgabemodul zur Verfügung.

Die folgende Abbildung 3.1 faßt die einzelnen Modules des Grobentwurf zusammen.

Die gepunkteten Pfeilen zwischen der Oberfläche und den restlichen Modulen zeigen, daß die Oberfläche sämtliche Module steuert. Hier entsteht kein nennenswerter Datenfluß, da meistens Steuerbefehle und Optionswerte ausgetauscht werden.

Der Parser übergibt dem Generator die analysierten Eingabedaten, die diese in Nassi-Shneiderman-Diagramme überführt. Deren graphische Darstellung wird dann dem graphischen Editor und der

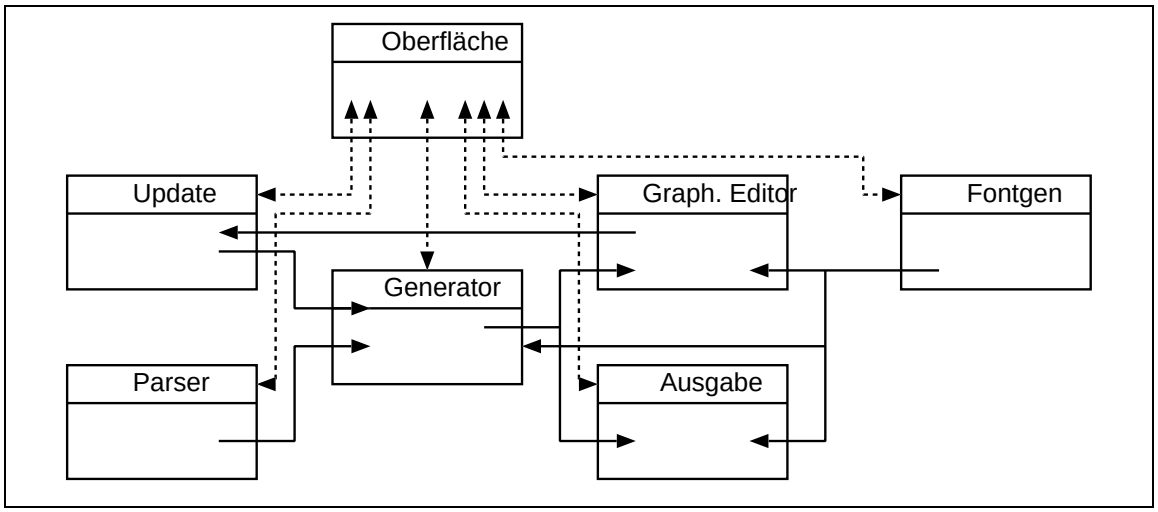


Abbildung 3.1: Module des Grobentwurfs: Die Verbindungspfeile zwischen den Modulen geben die Richtung des Datenaustausch wieder. Dicke durchgezogene Linien beschreiben den Fluß der Eingabedaten. Die gepunkteten Linien beschreiben den Fluß der Kontrollinformation.

Ausgabe weitergegeben. Zusätzlich gehen die Layoutänderungen, die der Benutzer im graphischen Editor vornimmt, direkt zurück zum Update-Modul.

Die Vielzahl von Datenflüssen zwischen den Modulen zeigen, daß hier eine Verfeinerung oder eine andere Lösung notwendig ist.

Bevor der Entwurf der einzelnen Module verfeinert wird, analysiert der nächste Abschnitt die Art des Datenfluß zwischen den Modulen genauer.

3.2 Datenaustausch zwischen den Modulen

Die Art des Datenaustausch zwischen den Modulen bestimmt auch zugleich die Schnittstelle zu den einzelnen Modulen. Die meisten Module benötigen neben dem Programmtext auch die im Abschnitt 2.2.5 auf Seite 9 beschriebene innere Struktur des Eingabeprogramms.

Abb. 3.1 zeigt, daß zwischen den Modulen viele Wege für den Austausch von Daten vorgesehen sind. Im wesentlichen handelt es sich bei den Daten um den Programmtext. Jedes Modul benötigt Teile oder die komplette Strukturinformation des Eingabeprogramms (z.B. der Generator). Je weiter die Daten vom Parser hin zur graphischen Ausgabe gelangen, desto mehr Informationen müssen pro Statement gespeichert werden.

Da die auszutauschende Information auf allen Datenpfaden die gleiche Struktur (Struktur des Eingabeprogramms) und viele gleiche Teile enthält und zudem der Umfang der Daten zu groß wäre um mehrfache Kopien der Information zu halten, wird für den Austausch der Daten ein gemeinsame Datenstruktur definiert. Diese existiert während der Laufzeit nur einmal im Hauptspeicher und die einzelnen Module erhalten bei ihrem Aufruf nur eine Referenz auf diese Datenstruktur.

Jedem Modul stehen also alle bisher gesammelten Informationen zur Verfügung. Die Struktur sollte der Struktur der Eingabedaten entsprechen, also ein Parse Tree des Eingabeprogramms. Die Austauschstruktur wird im weiteren mit **NXS** (*Nassi eXchange Structure*) bezeichnet (Abb. 3.2 auf der nächsten Seite).

Die NXS wird als dynamische Baumstruktur (Parse Tree) aufgebaut. Die Informationen werden in den Knoten gespeichert. Da jedes Modul lesend und schreiben auf die NXS zugreifen kann, müssen Regeln aufgestellt oder innerhalb der NXS getrennte Bereiche für die Module definiert werden, die einen geordneten Zugriff auf die NXS zulassen.

Die Knoten der NXS entsprechen den Statements im Eingabeprogramm. Elementare Statement (wie z.B. `printf`) bilden einen Endknoten (Blatt). Schleifenanweisungen und Fallunterscheidungen

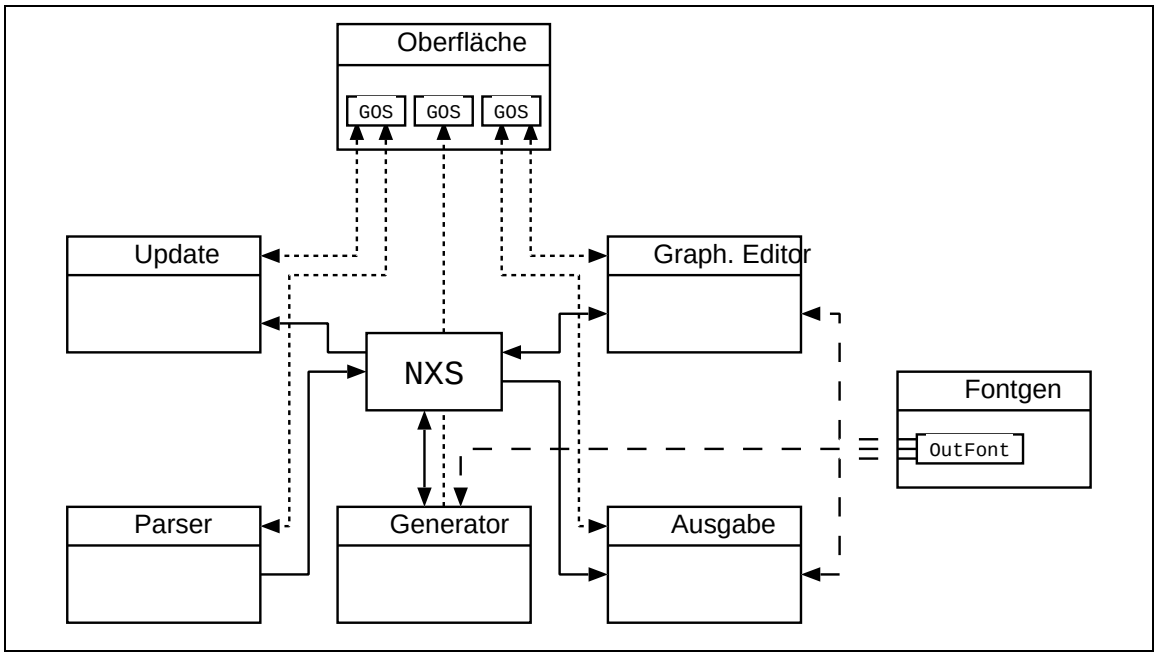


Abbildung 3.2: Neuer Grobentwurf: Für den Datenaustausch wird die Struktur *NXS* genutzt. Alle Module greifen lesend bzw. schreibend auf diese Struktur zu. Die Kommunikation zwischen den Modulen und der Oberfläche wird auch über gesonderte Strukturen vorgenommen (siehe Abschnitt 3.3.2 auf Seite 20). Die vom Modul FontGen bereitgestellten Informationen über Fonts werden als dritte Datenstruktur für die anderen Module *read-only* gespeichert(siehe Abschnitt 3.9 auf Seite 58).

enthalten wiederum Anweisungen und spiegeln sich daher als innere Knoten des NXS-Baumes wieder.

Folgende Informationen werden von den Modulen in einem Knoten gespeichert. Das in Klammern angegebene Modul hat dabei die Aufgabe, die entsprechende Information zu erstellen.

- laufende Nummer des Statements (Parser)
- Typ, z.B. Statement, While, If, ... (Parser)
- Strukturinformationen, z.B. Verweise auf weitere Statements (Parser)
- Verweise auf den originalen Source-Code, z.B. Position (Parser, Update)
- Originales Code-Segment und analysiertes Code-Segment (Parser)
- Hints und Flags für die Steuerung der Formatierung der Diagramme (Graph. Editor, Parser)
- graphische Objekte, die den Knoten im Diagramm repräsentieren (Generator)

3.2.1 Aufbau der Austauschstruktur NXS

Die Analyse eines Eingabeprogramms (Kap. 2.2.5) zeigt die mögliche Zerlegung eines Programms in eine Baumstruktur. Diese Darstellung soll für die Struktur der NXS beibehalten werden. Da die Eingabeprogramme verschiedene Größe und Strukturierung besitzen muß die NXS dynamisch an das aktuelle Eingabeprogramm angepaßt werden.

Abbildung 3.3 auf der nächsten Seite zeigt die vorläufige Struktur der NXS, die zu diesem Entwicklungszeitpunkt ersichtlich ist.

Die Struktur NXS besteht aus einer Anzahl von skalaren Werten: laufende Nummer (`nr`), Type des Knotens (`type`), einem Flag, das anzeigt, ob dieses Statement Änderungen gegenüber dem Eingabeprogramm enthält (`dirty`) und einem Verweis auf Anweisungen, die in diesem Knoten enthalten sind (`*body.`) Diesem Zeiger gehört die zentrale Aufgabe beim Aufbau der rekursiven Struktur der NXS. Durch `union` wird angedeutet, das der Zeiger auf Elemente verschiedenen Typs verweisen kann. So ist die Art der darin enthaltenen Anweisungen abhängig von Typ der Anweisung.

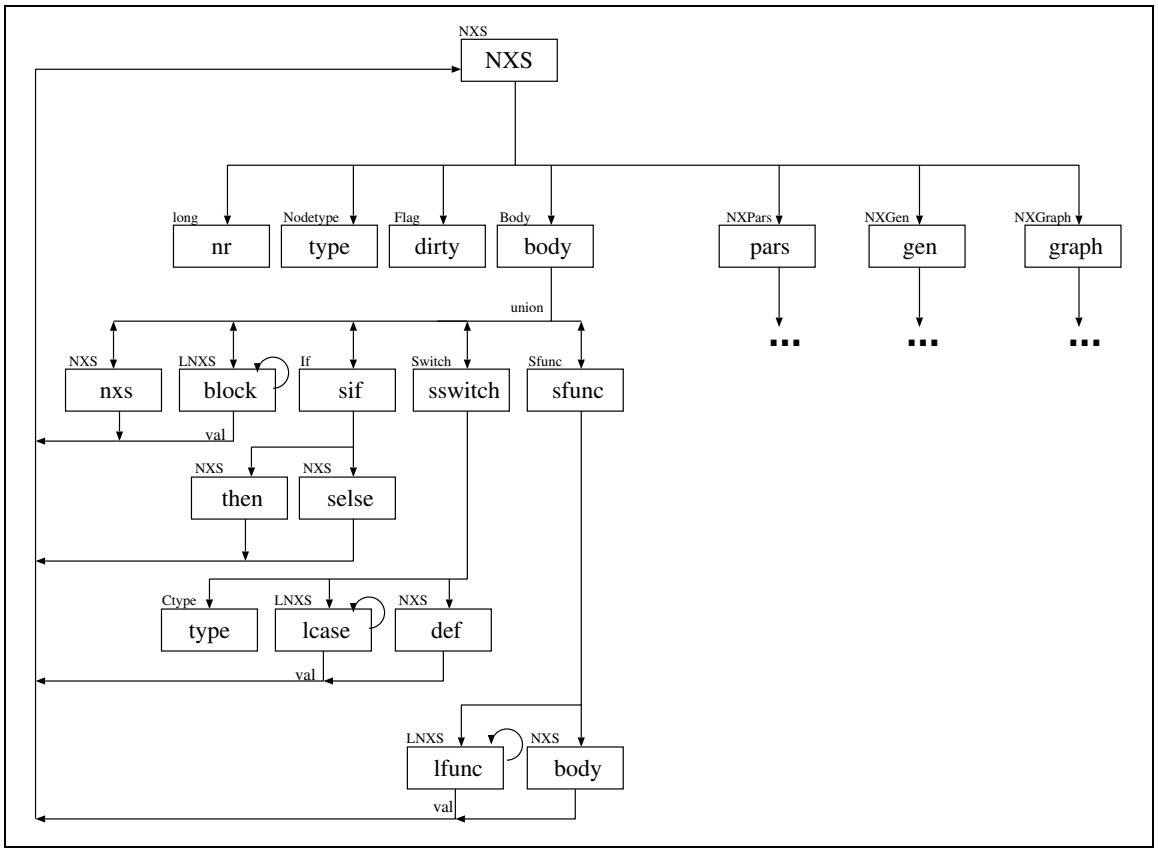


Abbildung 3.3: Vorläufige Struktur der NXS: Die mit $\star$ bezeichneten Elemente sind Zeiger (Verweise) und ermöglichen den dynamischen Aufbau der NXS. Elemente, die mit einem runden Pfeil auf sich selbst versehen sind, stellen eine geordnete Liste dar, die mehrere Elemente speichern kann. Oberhalb der Elemente steht immer der Typ des Elements. `NXS` ist die Struktur selber, `LNXS` ist eine Liste von Zeigern auf die Struktur `NXS`.

Eine While-Schleife benötigt nur einen weiteren Verweis auf die Anweisung, die durch die Schleife iteriert wird. Bei einem If-Statement gibt es zwei Anweisungen die je nach Wert der If-Bedingung durchlaufen werden. Dort werden zwei Verweise (*then*- und *else*-Anweisung) benötigt. Die Liste von NXS (`⋆block`) wird bei *Compound-statement* benötigt. Diese meistens mit Klammern (oder `begin-end`) zusammengefaßten Anweisungen bilden einen Block und werden im Programm in der dort aufgeführten Reihenfolge durchlaufen. In der NXS spiegelt sich dieser Block durch die geordnete Liste von Verweisen auf NXS wieder. Der Verweis `⋆sswitch` steht für die Speicherung von C-Switch- oder Pascal-Case-Strukturen zur Verfügung. Hier müssen Verweise auf eine Liste von Case-Blöcken und ein Verweis auf den Alternativ-Block (`default` in C, `otherwise/else` in Pascal) gespeichert werden. `⋆sfunc` wird nur bei Pascal-Programmen benötigt. Dort besteht die Möglichkeit, Funktionen zu schachteln, also lokal zur der aktuellen Funktion/Prozedur zu definieren. Diese lokal definierten Funktionen werden im `funktion`-Knoten über die Liste `⋆lfunc` eingehängt. `body` ist dann der Verweis auf den Funktionen-Rumpf.

Dieser erste Teil der NXS wird im wesentlichen vom Parser erzeugt. Die restlichen Elemente der NXS sind für die Informationen der anderen Module reserviert (`⋆pars` $\rightarrow$ Parser, `⋆gen` $\rightarrow$ Generator und `⋆graph` $\rightarrow$ Generator/Graphischer Editor).

Änderungen an der Struktur sollten im wesentlichen nur die Teilbereiche der einzelnen Module betreffen. Die Verbindung der Knoten untereinander durch die linke Hälfte der Abbildung sollte für die Darstellung der meisten Programmiersprachen genügen. Für andere Verbindungen, also neue Kontrollstrukturen im Programmtext, müßte auch eine neue Darstellung im Nassi-Shneiderman-Diagramm nach sich definiert werden.

3.2.2 Abspeichern der Austauschstruktur NXS

Für die Entwicklungsphase sollten zuerst zwei Routinen erstellt werden, die diese dynamische Struktur von einer lesbaren ASCII-Datei einlesen und darin auch wieder abspeichern können. Als Format für diese lesbare Darstellung bietet sich hier das Stanza-Format an. Mit diesen Routinen ergibt sich der Vorteil, daß die Entwicklung der Module quasi zeitgleich beginnen kann. Jedes Modul kann die NXS über die Stanza-Datei einlesen, die eigenen Modifikationen vornehmen, und wieder abspeichern. Erst beim Zusammenfügen der Module werden die beiden Routinen nicht mehr benötigt, da die NXS vom Parser generiert und nur zur Laufzeit im Hauptspeicher besteht.

3.3 Modul Oberfläche

Die Oberfläche dient zur Steuerung der Aktionen, die bei der Generierung von Nassi-Shneiderman-Diagrammen notwendig sind. Entgegen der Batch-Version von *Nassi* ist die Oberfläche (GUI) *Ereignis*-orientiert zu entwerfen. Hier gibt es keinen vorher festgelegten Durchlauf durch das Programm, sondern nur festgelegte Aktionen, die durch Ereignisse, meistens Benutzeraktionen, angestoßen werden.

Die Oberfläche soll das Laden, Darstellen und Speichern von Programmen, sowie das Ausgeben von Nassi-Shneiderman-Diagrammen ermöglichen. Der Benutzer soll während der Arbeit mit *Nassi* immer die Grobstruktur des Eingabeprogramm überblicken können. Dazu muß die Liste der in der Eingabedatei vorhandenen Prozeduren und Funktionen in einem Teil der Oberfläche zu sehen sein. Der Hauptteil des Bereichs soll für die Darstellung der Diagramme zur Verfügung stehen. Da immer nur ein Diagramm gleichzeitig dargestellt werden soll, muß eine Selektion der entsprechenden Routine möglich sein.

Alle Benutzeraktionen sollen über ein Menüsystem erreichbar sein. Dazu gehören neben dem Laden und Speichern der Programme auch das Generieren der Diagramme im Ausgabeformat. Umfangreiche Menü ergeben sich durch die Verwaltung von Optionen für die Formatierung und die Auswahl des Ausgabemediums. Die gebräuchlichsten Aktionen sollen zusätzlich zu den Menüs und Tastenkombinationen auch über eine Button-Zeile zur Verfügung stehen.

Die Interaktion mit dem Benutzer soll nur über ein Fenster laufen. Mehrere Fenster, die sich gegeneinander überdecken, aber gleichzeitig für die Eingabe bereitstehen, bringen eher Verwirrung und aus der Benutzersicht nicht mehr eindeutige Zustände. Daher soll das Hauptfenster von *Nassi* eine klare Struktur haben und alle möglichen Benutzeraktionen steuern können. Bei einem System wie *Nassi* ist die Interaktion über nur ein Hauptfenster möglich, da die Anzahl der Aktionen bei *Nassi* gegenüber Systemen mit GUI-Schwerpunkt eher gering ist. Abbildung 3.4 auf der nächsten Seite gibt einen groben Überblick über die Teilbereiche des Hauptfensters.

Am unteren Ende des Fenster steht ein Ausgabebereich zur Verfügung, der für Status- und Fehlermeldungen genutzt werden kann. Eine solches Protokollfensters hilft dem Benutzer bei der Kontrolle der durchgeführten Aktionen. Zum Beispiel sollten dort immer Meldungen erscheinen, wenn Dateien auf der Festplatte angelegt, verändert oder gelöscht worden sind.

Die Steuerung von *Nassi* geschieht Ereignis-orientiert. Die GUI-Bestandteile von *Nassi* gliedern sich in zwei Teile: Die Definition der Oberfläche und die Beschreibung der Aktionen, die beim Eintreffen von Events durchgeführt werden sollen.

Die Beschreibung der Oberfläche umfaßt die Objekte, Menüs und Eingabefenster, die auf dem Bildschirm zu sehen sind. Dieser Teil kann mit GUI-Buildern *halb-automatisch* erledigt werden. Nach diesem Schritt steht ein Prototyp der Oberfläche zur Verfügung, der aber noch keine Aktionen, wie zum Beispiel das Laden von Dateien oder der Erstellen von Diagrammen erlaubt.

Im zweiten Teil werden sogenannte *Callback*-Routinen definiert, die von der Oberfläche beim Eintreten eines Ereignissen (Drücken einer Taste, Anklicken eines Icons) aufgerufen werden.

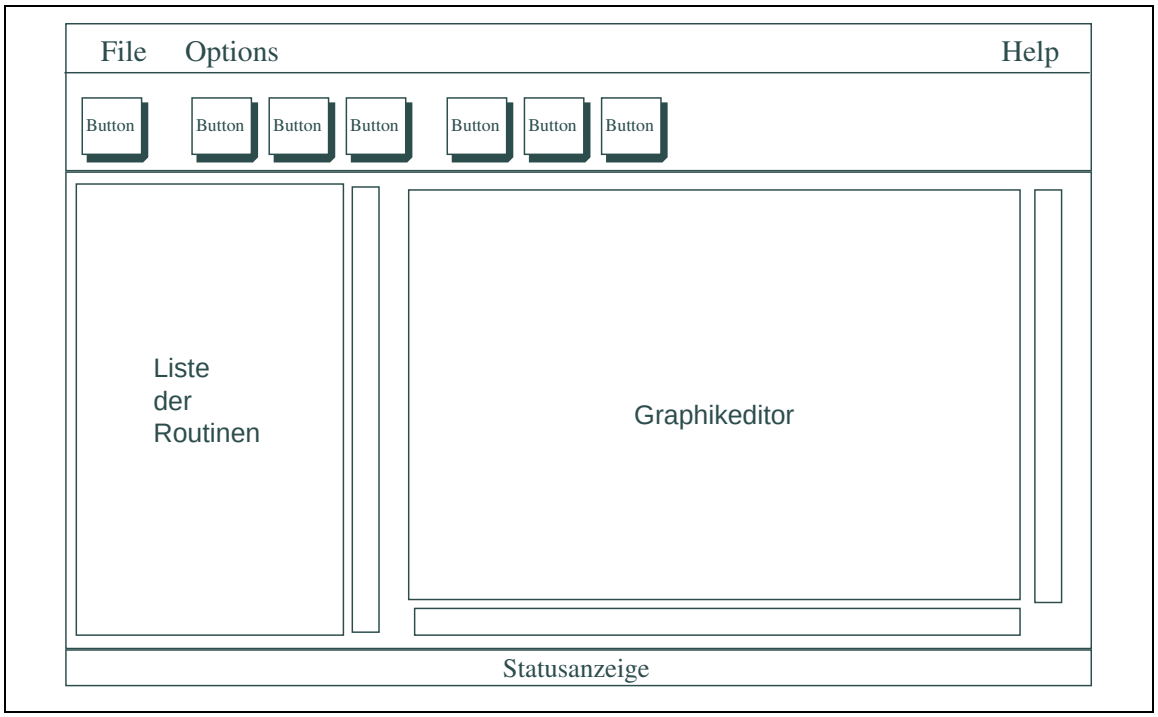


Abbildung 3.4: Grobstruktur des Hauptfensters von *Nassi*: Die beiden größten Bestandteile des *Nassi*-Fensters sind die Liste der vorhandenen Routinen und der Graphikeditor, der das Nassi-Shneiderman-Diagramm anzeigt. Beide Teilbereiche sind mit Scrollbars versehen. Die Button-Leiste unterhalb der Menüzeile soll ein schnelles Auffinden der häufig benutzen Funktionen ermöglichen.

Das Hauptprogramm von Nassi gliedert sich somit in zwei Teile:

1. Definition der Oberfläche
2. Endlosschleife zur Abarbeitung der Ereignisse

Der zweite Teil deutet an, daß es bei graphischen Benutzeroberflächen kein direktes Ende des Kontrollflußes im Hauptprogramm gibt. Das Beenden einer Oberfläche geschieht typischerweise über einen Menüeintrag `Exit`. Wird dieser vom Benutzer angewählt, wird die diesem Ereignis zugewiesene Callback-Routine aufgerufen. Während der Endlosschleife unterliegt die Ablaufkontrolle der Oberfläche. Die Kontrolle gelangt nur zu den Zeitpunkten zurück zu *Nassi*, wenn ein Ereignis eingetreten ist.

Dieses Ereignis-orientierte Verhalten ist eine Eigenschaft die bei allen Entwicklungstools für Oberflächen zu finden ist. Daher ist in dieser Entwurfs-Phase der Entwicklung keine Spezialisierung auf ein bestimmtes Tools notwendig.

Die Ereignisse und deren Behandlung und der Austausch der Information aus den Optionen-Menüs werden in den folgenden beiden Abschnitten näher betrachtet.

3.3.1 Ereignisse und deren Behandlung

Die vom Benutzer angestoßenen Ereignisse bestimmen das Zusammenspiel der restlichen Module von *Nassi*. So muß z.B. nach dem Neuladen einer Eingabedatei der Parser angestoßen werden. Die folgende Liste gibt eine Übersicht über die möglichen Ereignisse und die sich darin anschließenden Aktionen von *Nassi*.

Beenden von Nassi (Exit) Überprüfen, ob Änderungen noch nicht gespeichert sind, Verlassen von *Nassi*

Laden einer Programmdatei (Load) Aufruf des Parsers, Anzeige der im Eingabeprogramm enthalten Dateien, Löschen der Graphikanzeige

Speichern einer Programmdatei (Save) Aufruf des Update-Moduls, Speichern der Datei

Erneutes Laden einer Programmdatei (Reload) Aufruf des Parsers, Anzeige der im Eingabeprogramm enthalten Dateien, Löschen der Graphikanzeige

Drucken, Preview Abfrage des Druckers, Ausgabeformat einstellen, Aufruf des Generators für die entsprechenden Routinen, Ausgabeformat zurückstellen, Aufruf des Generators für die Anzeige

Graphikdateien erzeugen Aufruf des Generators für die entsprechenden Routinen, Ausgabeformat zurückstellen, Aufruf des Generators für die Anzeige

Veränderung von Optionen je nach Definition der Option, entsprechende Module aufrufen

Fenstern vergrößern, verkleinern eigenes Fenster anpassen, Graphikeditor informieren

Der Generator berechnet zu einer Routine des Eingabeprogramms das Nassi-Shneiderman-Diagramm. Die berechnete Graphikinformation wird in den zugehörigen Knoten der NXS abgespeichert. Da die Schriftgrößen, Abstände und Breite des Diagramm von dem Ausgabeformat abhängen, sind die erstellten Graphikinformation nur für ein Ausgabeformat gültig. Insbesondere muß für die Anzeige im Graphikeditor ein anderes Format eingestellt werden. Daher muß bei manchen Ereignissen von und nach der Aktion der Generator aufgerufen werden. Im ersten Aufruf wird das Diagramm für das in der Aktion erwartete Ausgabeformat generiert. Nach der Aktion wird das Diagramm wieder für das ursprüngliche Format hergestellt.

Die Definition der Anbindung von Scrollbars an Fenster oder von Tastenkombinationen sollten durch das Entwicklungstools der Oberfläche unterstützt werden. Einstellungen, die das Layout des Hauptfensters betreffen, sollten über X-Ressourcen definiert werden. Einstellungen die die Generierung und Darstellung der Diagramme betreffen, sollten in einem Profile gespeichert werden können.

3.3.2 Beschreibungsstruktur der Optionen-Menüs

Eine Anforderung der der Entwicklung von *Nassi* war, daß die Module klar voneinander getrennt sind. Die Verwaltung der Optionen wird durch diese Forderung schwieriger.

Zum einem sind bei der Definition der Oberfläche die Optionen der einzelnen Module nicht bekannt. Zum anderen ist den Modulen eine selbständige Definition der Optionenmenüs durch die eben beschriebenen Anforderung nicht erlaubt.

Würden die Module die Menüs selbst definieren, müßten im Oberflächen-interne Funktionen und Strukturen nach außen offengelegt werden. Ein einfaches Austauschen der Oberfläche unter Einbehalt der Schnittstellen, z.B. beim Plattform-Wechsel, würde dann nicht mehr funktionieren. Änderungen in den einzelnen Modulen wären nötig.

Daher muß auch hier eine klare Schnittstelle definiert werden. Zwei Möglichkeiten stehen zur Verfügung: Definition von öffentlichen Routinen auf der GUI-Seite, die Menüeinträge generieren und verwalten, oder die Definition einer weiteren Struktur, die eine Beschreibung der gewünschten Menüeinträge enthält. Für *Nassi* wurde die zweite Möglichkeit gewählt, da dort die Kontrolle der Menüs mehr im GUI-Modul liegt und damit der Verwaltung einfacher wird.

Die Optionen der einzelnen Module müssen nicht für den gesamten Programmlauf gleich sein. Wechselt der Benutzer zum Beispiel die Eingabesprache und Datei, ändern sich auch die Optionen, die der Parser zur Verfügung stellt. Gleiches gilt auch bei der Anwahl eines anderen Ausgabeformats. Dort kann das Ausgabemodul für PostScript mehr Einstellungen (z.B. Seitengröße und -orientierung) bieten, als das bei der Ausgabe im Tgif-Format der Fall wäre. Die Struktur GOS (General Options Structure) muß ähnlich der Struktur NXS dynamisch aufgebaut werden.

Die Optionen werden für jedes Modul einzeln verwaltet. Das Optionen-Menü von Nassi wird also in der obersten Ebene durch die Liste der Module (Parser, Generator und Ausgabe) gestaltet. Unter jedem dieser Punkte hängt dann ein durch die Struktur GOS definierter Menübaum, der die Optionen der des entsprechenden Moduls darstellt.

Die GOS wird durch eine geordnete Liste definiert. Jedes Listenelement beschreibt eine Option und enthält folgende Elemente:

- Beschreibung der Option, erscheint im Menü,
- Name der Option (Id, Hint-Name), für Speicher und eindeutige Identifikation der Option,
- Flag, ob die Option geändert wurde,
- Flag, ob neue Initialisierung der Menüs nötig ist,
- Flag, ob die Option verfügbar ist,
- Darstellungsart der Option,
- Liste der möglichen Werte der Optionen, abhängig von der Darstellungsart,
- Wert der Option

Die drei Flags helfen bei der Verwaltung der Optionen. Durch das erste Flag kann ein Modul schneller feststellen, ob sich die Option seit dem letzten Aufruf des Moduls verändert hat. Ist das zweite Flag gesetzt wird bei jeder Änderung der Option das entsprechende Modul direkt aufgerufen. So können Änderungen an Optionen direkt in die Darstellung überführt werden. Mit dem dritten Flag können Optionen zeitweilig ausgeschaltet werden. Die Optionen sollten dann in den Menüs versteckt, bzw. grau und inaktiv dargestellt werden. Tabelle 3.1 beschreibt die definierten Darstellungsarten.

Art der Option	Beschreibung
CHOICE	eigenes Fenster mit Liste der möglichen Werte untereinander; es kann nur eines ausgewählt werden
RADIOBUT	mehrere nebeneinander stehende Möglichkeiten zu einer Option, von der nur eine ausgewählt werden kann
COUNTER	Anzeige des aktuellen Wertes mit nebenstehenden Pfeilen zur In- bzw. Dekrementierung
DOUBLE_SLIDER	Slider, mit dem double-Werte eingestellt werden können
ON_OFF	Schalter, um Option an- bzw. auszuschalten
SUBMENU	neues Fenster mit Liste von weiteren Optionen
TEXT	Eingabefeld für beliebigen Text

Tabelle 3.1: Darstellungsarten in den Optionen-Menüs

Von der Darstellungsart hängt die Interpretation der Liste möglicher Werte ab. Bei einem Slider definiert die Liste Anfangs- und Endwert des Sliders. Bei einem Radiobutton definieren die Listenelemente die anzuwählenden Werte.

Ein besonderes Element bietet der Darstellungsart SUBMENU, mit der ein weiteres Untermenü eingehängt werden kann. Dadurch ist eine rekursive Definition der GOS möglich. Optionen können gruppiert und die Menüs übersichtlicher gestaltet werden.

Jedes Modul, für das eine Verwaltung der Optionen von der GUI erwünscht ist, muß eine Routine `init_opt_`*`modul`* definieren, die einen Verweis auf die Optionenliste zurückgibt. Diese Routine wird beim Start der Oberfläche oder bei einer erneuten Initialisierung von *Nassi* aufgerufen. Damit sind die Optionen den Moduls in der GUI bekannt. Die aktuellen Werte der Optionen liegen immer im GUI-Modul. Den anderen Modulen wird ein Zeiger auf die Optionen-Liste bei jedem Aufruf mitgegeben, so daß dort die aktuellen Werte ausgelesen werden können. Zusätzlich dazu soll die GUI auch eine Abfragefunktion anbieten, die zu einer Option (über ID spezifiziert) den aktuellen Wert zurückgibt.

Eine zweite Routine `reinit_opt_`*`modul`* wird von der Oberfläche nach jeder Änderung aufgerufen. Diese dient zur Überprüfung von Abhängigkeiten zwischen den einzelnen Optionen. Wird zum Beispiel das Ausgabeformat geändert, muß vom Modul Ausgabe die Aktivierung der restlichen Optionen angepaßt werden.

Die Speicherung der Optionen soll entweder in einem Benutzerprofile oder im Programmtext selber geschehen. Optionen in einem Benutzerprofile gelten für alle Anwendungen von Nassi. Optionen im

Programmtext gelten nur für diesen Programmtext. Bei der Speicherung der Optionswerte müssen von der Oberfläche die aktuellen Werte aus den Optionen-Listen ausgelesen und mit der eindeutigen ID versehen im Profile oder Programmtext abgespeichert werden. Bei Start von *Nassi* oder dem Laden einer Programmdatei werden die Werte eingelesen, die Optionen mit der entsprechenden ID in den Listen gesucht und die Werte aktualisiert. Die Speicherung der Optionen im Programmtext muß an die Programmiersprache angepaßt sein. Zur Speicherung muß daher das Update-Modul eine Schnittstelle bieten.

Die unabhängige Definition der Optionen unterbindet bei der Definition der Module die Kontrolle über die Layoutgestaltung der Menüs. Die Layoutgestaltung der Menüs liegt nun bei der GUI, Inhalt und Form der Menüs sind voneinander getrennt.

3.3.3 Schnittstelle zu den anderen Modulen

In diesem Abschnitt werden nochmal die Schnittstellen zu den anderen Modulen zusammengefaßt. Da die Oberfläche in der Hierarchie der Module die oberste Ebene darstellt und somit alle anderen Module direkt oder indirekt aufruft, ist hier die Schnittstelle durch die Aufrufe dieser Module und den dabei verwendeten Parameter definiert.

In einigen Situationen muß es neben dem üblichen Aufruf des Moduls (z.B. `draw(...)`) auch Aufrufe weiterer Routinen des Moduls geben. Dies ist z.B. für die Definition der Optionen notwendig.

`module()`	(Haupt-)Aufruf des Moduls
`NXS`, Eingabestruktur	Eingabedaten (Programmtext und Struktur)
`GOS`, Optionenstruktur	Definition der Optionen
`init_opt_`*`modul`*	Initialisierung der Optionen eines Moduls
`reinit_opt_`*`modul`*	Reinitialisierung der Optionen
`save_hints`	(Update) Abspeichern der Optionen im Programmtext
`read_hints`	(Update) Lesen der Optionen aus dem Programmtext
`redraw_canvas`	(Graphikeditor) Aktualisierung des Graphikeditors bei Veränderung des Fensterlayouts
`add_status_line`	Ausgabe einer Status-, bzw. Fehlermeldung

Tabelle 3.2: Schnittstelle der Oberfläche zu anderen Modulen

3.4 Modul Parser

Die Aufgabe des Moduls Parser ist es, Quelltexte zu analysieren und daraus alle Informationen zu extrahieren, die von den anderen Modulen benötigt werden. Im einzelnen sind folgende Daten zur Verfügung zu stellen:

- Generator: Dieses Modul benötigt die syntaktische Struktur des Programms in Form der Baum-Struktur der NXS. Zu jedem Knoten muß dabei auch der auszugebende Text verfügbar sein. Dieser sollte zudem mit Hinweisen über den Inhalt der Texte enthalten, um eine syntaxabhängiges Layout zu ermöglichen (also unterschiedliche Darstellung von Schlüsselwörtern, Funktionsnamen, etc.).
- GUI: Das GUI bietet eine Liste von Funktionen zur Darstellung an. Diese wird vom Parser in Form einer Liste von NXS-Bäumen generiert. Die Namen der Funktionen müssen dem GUI als Attribute dieser Bäume zur Verfügung gestellt werden. Der graphische Editor erlaubt das Ändern von *hints*, die im Quelltext als NSCs abgelegt werden können. Der Parser stellt dazu eine Funktion zur Verfügung, die die Integration neuer/geänderter Hints in den Quelltext durchführt. Ferner soll über den graphischen Editor eine Navigation zwischen Funktionen möglich sein. Dazu muß zu jedem Funktionsaufruf ein Verweis auf die aufgerufene Funktion (also deren Toplevel NXS) vorhanden sein.
- Update: Dieses Modul benötigt den gesamten Quelltext incl. aller NSCs, um ihn in eine Datei speichern zu können.

Die Funktionen, über die das GUI mit dem Parser interagiert, wurden bereits im Abschnitt 3.8.1 auf Seite 46 beschrieben, deshalb kann sich hier auf die Datenstrukturen und deren Aufbau konzentriert werden. Die NXS-Struktur wird vom Parser aufgebaut. Da die für den Syntaxbaum wesentliche hierarchische Struktur von NXS bereits beschrieben wurde, beginnen wir mit dem darin enthaltenen Quelltext.

3.4.1 Die Datenstruktur Ptext

Aus Performancegründen sollte die Quelldatei nur einmal gelesen werden. Damit auch der Update ohne nochmaliges Einlesen der Datei erfolgen kann, muß der gesamte Text (also nicht nur der für den Generator sichtbare) im Hauptspeicher gehalten werden. Soll der Text nicht mehrfach im Speicher liegen, bietet sich zunächst folgendes Vorgehen an: der Originaltext wird linear im Speicher abgelegt und in der NXS-Struktur werden nur Verweise auf diesen Text verwaltet. Nachteil dieser Lösung ist, daß diese Verweise in der Regel mehr Platz benötigen als die Texte selbst (oft besteht ein Token nur aus einem Zeichen, etwa einer Klammer). Die Möglichkeit, *Hints* in dem Quelltext hinzuzufügen, zu ändern oder zu entfernen, bringt zusätzliche Komplikationen. Hier wurde daher ein anderer Weg beschritten. Der Quelltext wird vollständig auf die NXS-Knoten aufgeteilt und dort gespeichert. Jedem Textfragment werden die von den anderen Modulen benötigten Informationen in Form von Flags angehängt. Ein Textfragment bildet zusammen mit diesen Flags die Struktur `Ptext` (geParster Text). Zu jedem NXS gehört daher eine geordnete Liste von Ptext-Strukturen. Im folgenden werden die Attribute der Ptexte beschrieben, und welchem Zweck sie dienen:

- `Ttype`: Häufig enthält ein Ptext genau ein Token. Hier ist der Typ dieses Tokens abgelegt. Diese Information wird vom Generator für die Layout-Gestaltung genutzt, also z.B. unterschiedliche Farben/Fonts für Schlüsselwörter und Funktionen. Weitere Werte von `Ttype` sind Leerzeichen, Strings, Label (potentielle Ziele für Sprunganweisungen) und UNKNOWN, worin normaler Programmtext zusammengefaßt ist. Typen von Text, die vom Generator ignoriert werden müssen, sind z.B. Kommentare, Präprozessordirektiven und Deklarationen. Ein weiterer Ttype, der nur für das Update-Modul von Bedeutung ist, ist der 'SLOT'. Dieser markiert eine Stelle, an der Text eines nachgeordneten NXS-Knotens eingefügt werden muß.

- `hidden`: ein Flag, das anzeigt, ob der Text vom Generator berücksichtigt werden muß. Auf diese Weise kann der Generator ohne Kenntnis der für ihn nicht relevanten Ttypen arbeiten. Als `hidden` sind vor allem Kommentare markiert, aber auch strukturbildende Token, die in den Diagrammen nicht erscheinen (etwa Klammern in C, BEGIN END in Pascal).
- `ghost`: in bestimmten Fällen fügt der Parser Text ein, der vom Generator angezeigt werden soll, obwohl er im Quelltext nicht vorhanden war. Ein Beispiel ist, daß bei einem im Programm fehlenden `default` einer Mehrfachverzweigung ein leerer Default-Zweig erzeugt wird, der nur das Wort `default` enthält. Findet der Parser einen Syntaxfehler im Quelltext, so wird zudem eine Fehlermeldung generiert, die ebenfalls als `ghost` in die Ptext-Liste eingehängt wird, während der fehlerhafte Text `hidden` ist. Dieses Flag wird vom Update-Modul benötigt, da dieses alle Ghost-Texte ignorieren muß.

Das folgende Beispiel zeigt die Ptext-Liste eines Do-while-Statements.

```
/* Zaehler */
do
  i++;
while(i<10);
```

Text	Ttype	hidden	ghost
`/* Zaehler */`	COMMENT	yes	no
`do`	KEYWORD	yes	no
	SLOT	yes	no
`while`	KEYWORD	yes	no
`(`	UNKNOWN	yes	no
`a`	ID	no	no
`<`	UNKNOWN	no	no
`10`	UNKNOWN	no	no
`)`	UNKNOWN	yes	no
`;`	UNKNOWN	yes	no

Der Übersichtlichkeit halber sind die Leerzeichen und Zeilenumbrüche, die alle vom Typ SPACE und weder `hidden` noch `ghost` sind, nicht dargestellt. Nur der Text `i<10` erscheint im Diagramm, ist also nicht als `hidden` markiert. Der Schleifenkörper des do-while Statements ist ein Block, der in einen eigenen NXS beschrieben ist. Der `SLOT` markiert die Stelle, an der der Ptext dieses NXS eingefügt werden muß.

3.4.2 Auswahl eines Parsers

Da das Modul GUI den Parser nur über die Funktion `parser()` anspricht, muß dieser selbst entscheiden, welcher spezielle Parser für ein gegebenes Quellprogramm benutzt wird. Gesteuert wird diese Auswahl über Parser-Optionen im GUI. Es kann entweder explizit ein Parser ausgewählt werden oder eine automatische Wahl eingestellt werden. Im letzten Fall wird der Parser anhand der Datei-Endung ausgewählt. Die Funktionen, die das Interface zwischen Parser und GUI bilden, sind somit nur Mittler; sie entscheiden lediglich, welcher Parser benutzt wird und geben dann die Kontrolle an diesen weiter. Jeder Parser muß dazu nur die jeweiligen Funktionen zur Verfügung stellen, sowie zusätzlich eine Funktion, die anhand eines Dateinamens entscheidet, ob diese Datei geparst werden kann.

3.4.3 Aufbau des Syntaxbaums

Die vom Parser zu lösende Aufgabe, der Aufbau eines Syntaxbaumes, ist auch Teil der Aufgaben eines Compiler-Frontends. Daher ist es naheliegend, Werkzeuge aus diesem Umfeld zu benutzen.

Hier fiel die Wahl auf die für alle UNIX-Plattformen vorhandenen Werkzeuge `lex` und `yacc` (bzw. deren GNU-Varianten `flex` und `bison`). Beide Programme sind Code-Generatoren. Sie erzeugen C-Code. Daher ist zumindest für das Parser-Modul C als Implementierungssprache festgelegt. Allerdings gibt es ähnliche Werkzeuge auch für andere Sprachen wie Java und C++.

`flex` dient der lexikalischen Analyse des Quelltextes. Im wesentlichen muß nur eine Beschreibung der Token der Programmiersprache durch reguläre Ausdrücke vorgenommen werden. flex generiert daraus eine Funktion, die einen Strom von Eingabezeichen in eine Sequenz von Token verwandelt. Unter einem Token versteht man in diesem Zusammenhang die kleinsten syntaktischen Einheiten einer Programmiersprache. Das sind in C beispielsweise Schlüsselwörter, Bezeichner (Variablen- und Funktionsnamen), Operatoren und Klammern. Für verbreitete Sprachen wir PASCAL und C gibt es bereits frei verfügbaren `lex`-Codes. Für nassi müssen hier noch Erweiterungen vorgenommen werden. Diese betreffen vor allem die Kommentare, die noch auf die NSCs (also die in Kommentaren versteckten Hints) untersucht werden müssen. Außerdem müssen Präprozessor-Direktiven erkannt und ignoriert werden.

Das Tool `bison` ist ein Parser-Generator. Aus der Backus-Naur-Form der Grammatik einer Programmiersprache wird ein Parser generiert. Aufbauend auf den vom Lexer gelieferten Token werden rekursiv sogenannte Symbole definiert. Zu jedem Symbol kann man eine Sequenz von C-Anweisungen angeben, die bei Erkennen dieses Konstruktes ausgeführt werden soll. Die Token (und anderen Symbole) können als formale Parameter in diesen Anweisungen verwandt werden. Die Backus-Naur-Form der PASCAL und C Grammatik kann den Standardwerken von Wirth bzw. Kernighan & Ritchie [3] entnommen werden. Für den Einsatz in *nassi* sind auch hier Modifikationen notwendig. Vereinfachend wirkt sich aus, daß keine Zerlegung elementarer Anweisungen vorgenommen werden muß, da diese Anweisungen die kleinste syntaktische Einheit für *nassi* sind. Erweitert wurde die Grammatik um die NSCs.

Für den C-Pseudocode Parser ergibt sich eine weitere Modifikation. Zwei Token-Typen der C-Grammatik sind der Identifier, der eine Variable oder Funktion kennzeichnet und der *type specifier*, der einen Variablentyp bezeichnet. Wie bereits im Abschnitt 2.2.2 auf Seite 8 erläutert, können diese Token nicht unterschieden werden, ohne den kompletten Quelltext (incl. Präprozessor-Direktiven) zu parsen. Daher unterscheidet der Parser nicht zwischen Deklarationen und Anweisungen in einer Funktion. Wichtiger ist, bei Funktionen den Typ des Rückgabewertes von dem Namen zu unterscheiden. Dies gelingt durch eine Modifikation der Grammatik. Im folgenden ist die Grammatik anhand zweier Beispiele demonstriert.

```
statement
        : naked_statement
        | TOK_NSC TOK_NSC_VAL TOK_NSC_END naked_statement
        ;
```

Das `naked_statement` entspricht dem Statement der normalen C-Grammatik. Ein Statement im Sinn von nassi ist entweder ein `naked_statement` oder ein ein `naked_statement` dem Hints in Form von NSCs vorangestellt sind. Man beachte, daß der von flex generierte lexikalische Analysator (Lexer) diese Hints als ein Token liefert (`TOK_NSC_VAL`). `TOK_NSC` und `TOK_NSC_END` sind die ein- und ausleitenden Kommentare (`/* NSC:` und `*/`).

```
repeat_statement
        : TOK_DO statement TOK_WHILE '(' any_stuff ')' ';'
        ;
```

Hier ist ein Do-while-Schleife dargestellt. Typisch für die hier verwandte Grammatik ist, daß anstelle der `expression`, die normalerweise in den Klammern steht, hier `any_stuff` steht. Die rekursive Definition von `any_stuff` ist zu komplex, um hier komplett dargestellt zu werden. Im wesentlichen handelt es sich aber um eine Folge von Token, die nur wenigen Einschränkungen unterliegt. So dürfen etwa Klammern nur paarweise auftreten und keine Schlüsselworte wie `if`, `for`,

etc. vorkommen (andere wie `struct`, `union`, etc. dürfen hingegen).

Beide Beispiele definieren Statements, die im Syntaxbaum durch einen NXS-Knoten repräsentiert werden. Der Code, der bei Erkennen dieser Statements durchlaufen wird, muß also vor allem einen solchen Knoten aufbauen.

Die Werkzeuge `flex` und `bison` sind so aufeinander abgestimmt, daß die generierten Lexer und Parser zusammenarbeiten können. Der Parser wird vom Benutzer aufgerufen und ruft seinerseits den Lexer wiederholt auf. Bei jedem Aufruf gibt der Lexer ein neues Token zurück. Dieses nimmt den Wert eines vom Programmierer spezifizierbaren Typs an. Für die Symbole, die Anweisungen repräsentieren, wird das eine NXS-Struktur sein, für die vom Lexer gelieferten Token sinnvollerweise eine Ptext-Liste. Es ist also Aufgabe des Lexers, die Zerlegung des Quelltextes in Ptexte durchzuführen. Man beachte, daß die einem Token zugeordnete Ptext-Liste meist mehr als einen Ptext enthält, da alle Leerzeichen und Kommentare, die zwischen dem Token und dem vorangegangenen Token stehen, diesem Token zugeordnet werden.

3.4.4 Nassi Special Comments (NSC)

Wie bereits beschrieben, kann der Benutzer von *nassi* mit Hilfe von Direktiven oder Hints die Formatierung der Nassi-Shneiderman-Diagramme beeinflußen. Um einmal vorgenommene Formatierungen über einen Aufruf von *nassi* hinaus wiederverwendbar zu machen, müssen diese Hints in einer Datei abgelegt werden. Denkbar wäre, hierfür eine zusätzliche Datei zu benutzen, in der die Hints und Verweise auf die Anweisungen für die sie gelten gespeichert sind. Dem Vorteil dieser Lösung, daß der Quelltext nicht modifiziert werden muß, steht das Problem gegenüber, daß bei nachträglichen Änderungen am Quelltext der Bezug zwischen Hints und Statements verloren gehen kann. Daher wurde hier ein anderer Weg gegangen. Die Hints werden in speziell formatierten Kommentaren im Quelltext direkt vor der Anweisung eingefügt, für die sie gelten. Hints, die für den gesamten Quelltext gelten (globale Hints), werden am Anfang der Datei eingefügt. Auf diese Weise bleibt der Bezug zwischen Hints und Anweisung erhalten. Außerdem erhält der Benutzer die Möglichkeit, die Hints mit einem Texteditor im Quelltext nachzubearbeiten. Bei dieser Lösung wird ein Modul benötigt, daß die Hints in den Quelltext integriert und den modifizierten Text in eine Datei schreiben kann. Dieses Update-Modul wird im nächsten Abschnitt beschrieben.

Für die Kommentare, die die Hints enthalten, wurde die Bezeichnung NSC (Nassi Special Comments) gewählt. Jeder Kommentar im Quelltext, der mit dem Schlüssel `NSC:` beginnt, wird als NSC behandelt, der Hints für die nachfolgende Anweisung enthält. Globale Hints beginnen mit dem Schlüsselwort `NSC-GLOBAL:`. Für die Kommentare selbst gelten die Regeln der jeweiligen Programmiersprache. NSCs werden in C durch `/* */` und in PASCAL durch { } eingeschlossen. Für die Hints selbst wurde eine einfache Syntax gewählt:

```
<SCHLUESSEL> = "<WERT>"
```

Enthält der Wert keine Leerzeichen, so sind die einschließenden `" "` optional. Die Schlüssel müssen die Form von C-Bezeichnern haben, also mit einem Buchstaben (oder Unterstrich) beginnen und nur Buchstaben, Ziffern und Unterstriche enthalten. Die Leerzeichen zwischen Schlüssel, = und Wert sind optional. Ein NSC kann mehrere solcher Schlüssel-Wert Paare enthalten. Diese müssen lediglich durch Leerzeichen, Tabulator oder Zeilenumbruch getrennt sein. Neben der Möglichkeit, Hints in den Quelltext einzufügen, kann *nassi* globale Optionen auch in Konfiguratiosndateien sichern (rc-Files). Diese müssen der gleichen Syntax wie die NSCs folgen.

Aufgrund der einfachen Syntax der NSCs ist ihr Parsen mit geringem Aufwand verbunden. Wie bereits im Abschnitt 3.4.3 beschrieben, liefert die lexikalische Analyse des Quelltextes den gesamten Inhalt eines NSCs als ein 'Token'. Dieses wird einer separaten Analyse unterzogen, für die ebenfalls eine flex-generierte Funktion benutzt wird. Als Ergebnis wird eine Liste von GHints erzeugt, die in die NXS-Struktur eingefügt werden kann.

3.5 Modul Update

Die Aufgabe dieses Moduls ist, den Quelltext eines mit *nassi* bearbeiteten Programms in eine Datei zu schreiben. Da dieser Text vollständig in den Ptext-Strukturen enthalten ist, ist dazu lediglich der NXS-Baum des Programms zu durchlaufen und die Ptexte in eine Datei auszugeben.

Die einzigen Änderungen, die mit *nassi* am Quelltext vorgenommen werden können, sind Einfügen, Löschen und Modifizieren der Hints, die im Text in Form von `NSC`s erscheinen. Auch diese sind (als hidden markiert) in den Ptext-Strukturen enthalten. Um die Konsistenz zwischen Hints und Ptexten zu erhalten, müssen Änderungen an den Hints, wie sie von den Modulen Oberfläche und Graphischer Editor vorgenommen werden, stets auch in die Ptexte übernommen werden. Das Update-Modul stellt dazu eine Funktion (`hints_to_ptext`) zur Verfügung, die die Ptexte eines NXS aktualisiert. Die Module, die Hints modifizieren, sind dafür verantwortlich diese Funktion aufzurufen.

Um eine gute Lesbarkeit der NSCs zu erzielen, wird bei der Formatierung der Texte zwischen lokalen und globalen Hints unterschieden. Bei den globalen Hints wird nur ein Schlüssel-Wert Paar pro Zeile geschrieben, alle lokalen Hints eines Statements werden dagegen in einer Zeile zusammengefaßt. Das folgende Beispiel zeigt einen C-Quelltext und einen NSCs versehenen Text, der vom Update-Modul erzeugt wurde. Bis auf die sprachspezifischen Zeichen zur Ein- und Ausleitung der Kommentare werden PASCAL-Programme identisch formatiert.

Originaltext:

```
void main() {

  printf("Hallo Welt!\n");
}
```

Mit einigen NSCs versehener Text:

```
/* NSC-GLOBAL:
FONT_NAME="Courier"
FONT_SIZE="12"
*/
void main() {

   /* NSC: KEY_FONT_NAME="NewCenturySchlbk" FILL_COLOR="red" */
   printf("Hallo Welt!\n");
}
```

3.6 Modul Generator

Der Generator ist für die Berechnung der graphischen Darstellung zuständig. Er kann dabei die vom Parser erzeugten Struktur NXS benutzen. Das Resultat der Berechnung soll wiederum in diese Struktur gespeichert werden. Dazu wird jedem Knoten eine Liste von graphischen Objekten zugefügt, die das in diesem Knoten gespeicherte Element des Eingabeprogramms im Nassi-Shneiderman-Diagramm darstellen.

In den folgenden Abschnitten wird zuerst der Zustand der Austauschstrukturen und die vorhandenen Hilfsmittel zusammengefaßt. Danach wird die Struktur der Ausgabe, also die Art und Venknüpfung der graphischen Objekte in der NXS erläutert. Auf dieser Basis können dann das rekursive Vorgehen bei der Erstellung der Diagramme, die Behandlung der Texte und dessen Umbruch definiert werden.

3.6.1 Vorhandene Strukturen und Hilfsmitteln

Vom Modul Parser ist jeder Knoten der NXS um eine Struktur `*gen` (für den Generator bestimmte Informationen) erweitert worden (siehe Abb. 3.5)

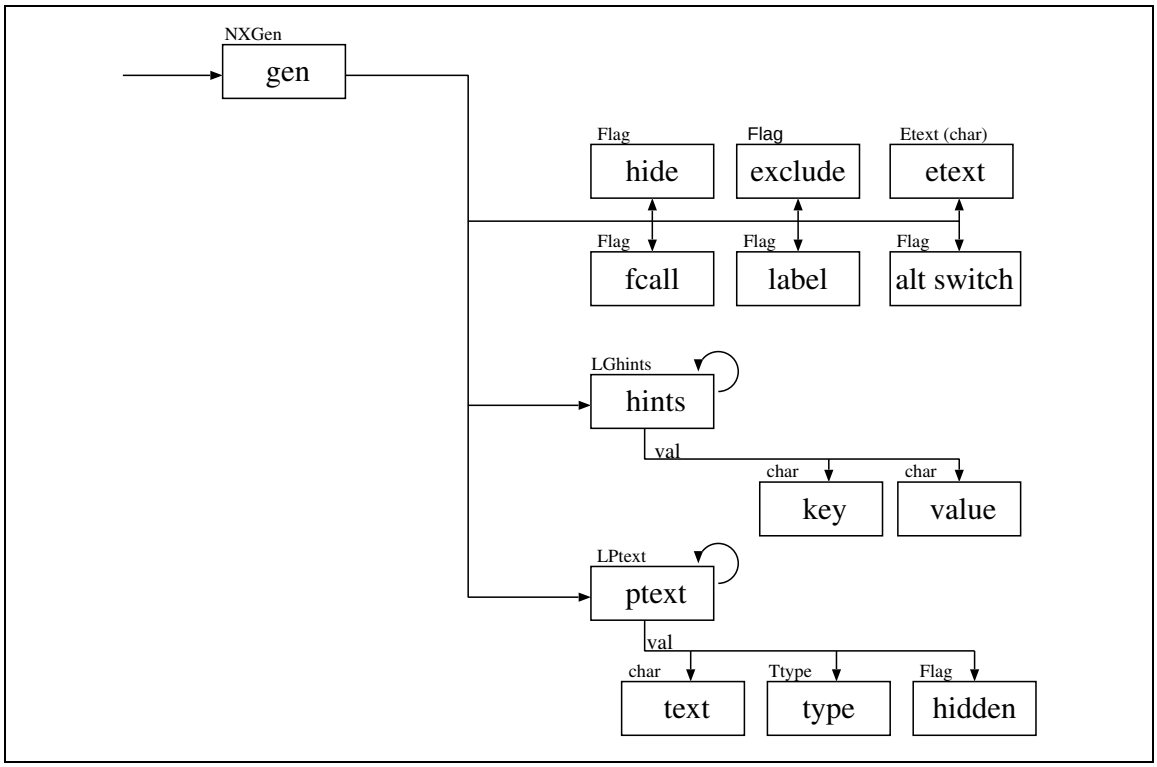


Abbildung 3.5: Teilstruktur `*gen` von NXS

Die in jedem Knoten des NXS gespeicherten Flags beschreiben die Gestaltung des Elements genauer. Die Flags beinhalten im wesentlichen dieselbe Information wie die Hints, bieten aber einen schnelleren Zugriff. Die Hints beschreiben Änderungen, die vom Benutzer über den Graphikeditor oder direkt im Quelltext über NCS-Kommentare eingebracht worden sind:

`LINE_SKIP`	Zeilenabstand
`LANGUAGE`	Sprache für Literale (z.B. True oder Wahr beim If-Statement)
`ALTERNATE_SWITCH`	Case-Zweige untereinander statt nebeneinander
`FCALL`	Markieren von Funktionaufrufen
`FUNCTIONHEAD`	Funktionskopf zeichnen
`MIN_WIDTH`	Mindestbreite eine Spalte bei If/Case (in %)
`COLSPEC_IF`	Aufteilung der Spalten beim If-Statement (%:%)
`[KEY]FONT_NAME`	Fontname der normale/hervorgehobenen Schrift

[KEY]FONT_FORMAT	Format der normale/hervorgehobenen Schrift
[KEY]FONT_SIZE	Größe der normale/hervorgehobenen Schrift
[KEY]FONT_COLOR	Farbe der normale/hervorgehobenen Schrift
SIDE_DISTANCE	Abstand zwischen Text und Seitenlinien
VERTICAL_DISTANCE_TOP	Abstand zwischen erster Zeile und Kopflinie
VERTICAL_DISTANCE_BOT	Abstand zwischen letzter Zeile und Fußlinie
FILL_COLOR	Hintergrundfarbe
LINE_COLOR	Linienfarbe
HYPHEN	Trennvorschläge für dieses Statement
SHOW_REPEAT	Anzeige des Schlüsselworts REPEAT
REPEAT_TEXT	alternativer Text für Schlüsselwort REPEAT

Sowohl die oben aufgeführten Schlüsselworte als auch deren Werte sind als Zeichenketten abgespeichert, müssen also bei der Bearbeitung des Statements interpretiert werden. Aus diesem Grund sind die wichtigsten Hints trotz des Double-Update-Problems als Flags gespeichert. Hints sind nur dann in der Liste gespeichert, wenn die eine Änderung gegenüber der globalen Einstellung bedeuten.

Die Hints gelten immer für den aktuellen Knoten und den darin enthaltenen Knoten. Die Werte werden also im NXS-Baum nach unten vererbt.

In der Liste `Ptext` ist der vom Parser analysierte Programmtext des Knoten enthalten. Der Text ist auf Wort-Ebene aufgebrochen. Jedem Wort wird vom Parser ein Typ und und ein Flag `hidden` zugeordnet. Vom Generator müssen nur Ptexte beachtet werden, deren Flag `hidden` nicht gesetzt ist. Mit dem Attribut `type` ist die Art des Ptexts genau geschrieben. Da der Generator die Texte nur in zwei verschiedene Schriftarten setzen soll (Keyword und normaler Text) müssen die Typen der Ptexte auf diese Schriftarten abgebildet werden: Ptexte vom Typ `T_UNKNOWN`, `T_STRING` und `T_FUNCTION` werden in normaler Schriftart und Ptexte vom Typ `T_KEYWORD` in der hervorgehobenen Schriftart gesetzt. Ein besonderer Typ ist `T_FORMAT`, durch den eine direkte Formatierung (z.B. zusätzliche Abstände) im Diagramm möglich ist.

Neben der NXS-Struktur stehen dem Generator die direkten Generator-Optionen aus der Oberfläche über die GOS zur Verfügung. Diese Optionen beschreiben die globalen Einstellungen, also die *Default*-Werte der Hints.

Als dritte Informationsquelle dient dem Generator die Funktion `fontmap` des Moduls fontgen, das zu einem Parametersatz (Name, Format, Große) einer gewünschten Schriftart eine Verweis auf eine reale im Ausgabeformat vorhandene Schriftart liefert.

3.6.2 Ausgabestrukturen

Für die Speicherung der vom Generator erstellten Graphikinformation steht die Teilstruktur *graph*(s. Abb. 3.6 auf der nächsten Seite) des NXS zur Verfügung, deren Aufbau in diesem Abschnitt erläutert wird. In jedem Knoten der NXS werden die zugehörigen Graphikelemente, wie z.B. Linien und Texte, abgespeichert. Jedem Graphikelement ist eine feste Position im Ausgabekoordinatensystem zugeordnet. Die Definition des Koordinatensystems ist an die PostScript-Darstellung angelehnt. Die Einheiten sind in $1/72$ Inch definiert, der Ursprung liegt aberlinks oben. Die Y-Achse zeigt nach unten, die X-Achse nach rechts. Als Graphikelemente stehen Boxen, Linien, geschlossene und offene Linienzüge und Texte zur Verfügung. Aus diesen Elementen muß der Generator die Nassi-Shneiderman-Diagramme zusammensetzen. Das Diagramm selbst hat eine vorgeschriebene Breite einzuhalten, die Länge des Diagramm ist nicht begrenzt.

Die Struktur *graph* besteht im Wesentlichen aus einer Liste von diesen Graphikelementen und zwei Angaben zur Position innerhalb der Koordinatensystems und die Größe (Dimension) der Graphikelemente. Die zweite Angabe beschreibt die Größe der Box, die durch die alle Graphikelemente der Liste gebildet werden. Diese beiden Angaben helfen zum Beispiel dem Graphikeditor bei der Selek-

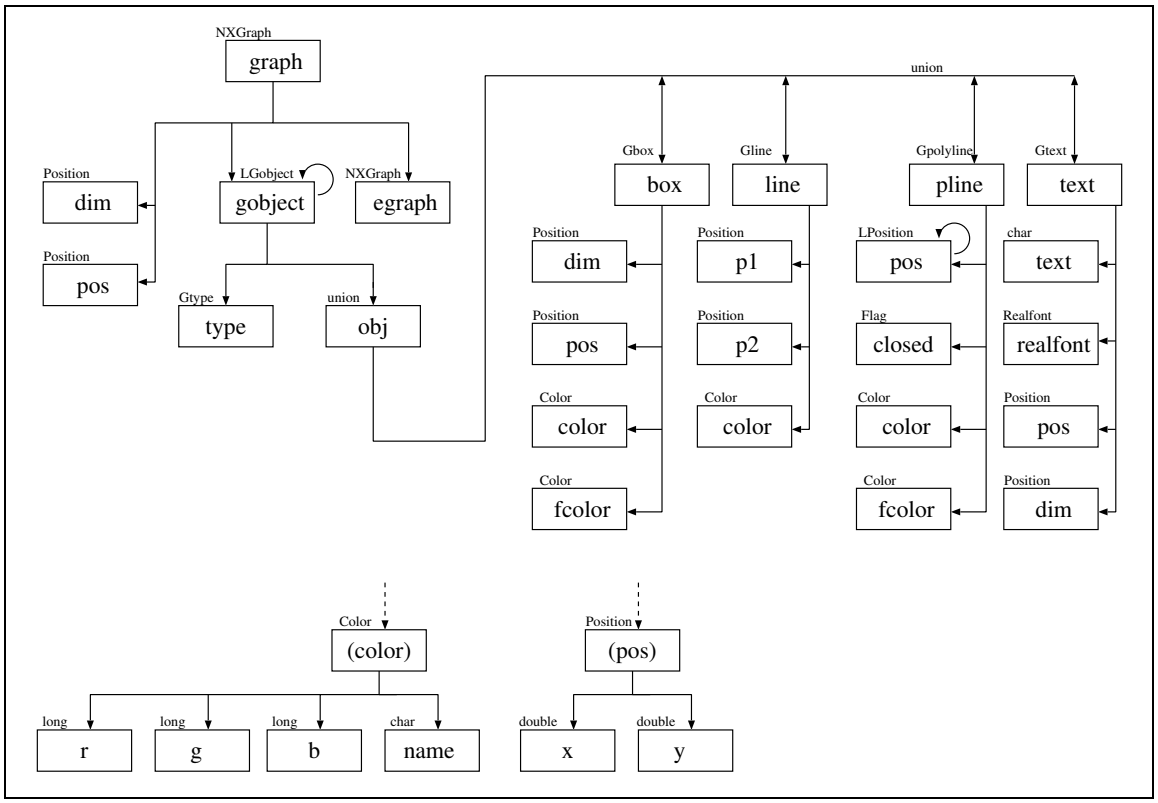


Abbildung 3.6: Teilstruktur `*graph` von NXS: Die beiden Strukturen *pos* und *color* werden an vielen Stellen benötigt. Sie beschreiben zum einen eine durch x und y beschriebene Position im Ausgabekoordinatensystem und zum anderen eine mit Name und R(ot)G(rün)B(lau)-Anteilen definierte Farbe. Die Struktur Realfont wird vom Modul `fontgen` definiert.

tion. Die Liste enthält die einzelnen Graphikelemente. Zum Beispiel kann eine einfache Anweisung durch ein Box- und ein bzw. mehrere Text-Elemente dargestellt werden.

Die Box kann zur Darstellung des Rahmens einer Anweisung genutzt werden. Mit der Box wird eine Rahmen- und eine Füllfarbe definiert. Von der Definition her, könnte ein solches Graphikelement auch aus einem Linienzug gebildet werden. Für die Effizienz und Einfachheit der weiteren Verarbeitung durch Graphikeditor und Ausgabe ist es aber sinnvoll die Graphikelemente in der höchstmöglichen Form der Abstraktion abzuspeichern. Die zweite Farbangabe beschreibt die Füllfarbe der Box. Sie bestimmt den Hintergrund der Box und impliziert, daß alle darin enthaltenen Elemente darüber gezeichnet werden müssen.

box	Darstellung eines Kasten
pos	Position (x,y)
dim	Größe, Dimension (dx,dy)
color	Linienfarbe
fcolor	Füllfarbe

Die Linie wird zum Beispiel für die Markierung von Funktionaufrufen benutzt. Sie wird von einem Anfangs- zu einem Endpunkt gezogen und besitzt als Attribut eine Linienfarbe.

line	Einzelne Linie
p1	Anfangsposition (x,y)
p2	Endposition (x,y)
color	Linienfarbe

Mit einem Linienzug können zum Beispiel die Dreiecksstrukturen eines If-Statements gezeichnet werden. Die Anzahl die Stützstellen ist variabel, so daß die Punkte in einer Liste gespeichert werden. Nur wenn es sich um einen geschlossenen Linienzug handelt wird die Füllfarbe für den

Hintergrund genutzt.

pline	geschlossener oder offener Linienzug
pos	Liste von Stützstellen
close	Flag, ob Linienzug geschlossen ist
color	Linienfarbe
fcolor	Füllfarbe bei geschlossenem Linienzug

Die Text-Elemente bestimmen den wesentlichen Anteil der Diagramme. Jedes Textelemente beschreibt eine zusammenhängende Folge von Zeichen, die im Attribut *text* gespeichert sind. Jedem Text ist als Anfangsposition der linke Eckpunkt der Basislinie zugeordnet, Als Dimension in X-Richtung ist die Breite des Textes und in Y-Richtung die Texthöhe zu sehen. Unterlängen werden bei diesen Angaben nicht berücksichtigt. Da bei der Generierung der Diagramme nicht nur äquidistante Schriften wie z.B. Courier verwendet werden, ist die Breite des Texte von der Breite der einzelnen Zeichen im angewählten Font abhängig. Der Generator ist auch für die korrekte Berechnung der Textbreite zuständig. Texte, die nicht in eine Zeile passen müssen vom Generator automatisch umgebrochen werden.

text	Textbaustein
text	Zeichenfolge
realfont	Verweis auf verwendeten Font (`FontGen`)
pos	Anfangsposition der Basislinie
dim	Größe (Breite und Höhe) des Textes
color	Textfarbe

3.6.3 Vererbung der Attribute

Die in im ersten Abschnitt 3.6.1 auf Seite 28 beschriebenen Gestaltungsattribute, die in über den Graphikeditor oder direkt im Quelltext über die NCS-Kommentare definiert werden können, gelten immer für diesen Knoten und den darunterliegenden Knoten in der NXS. Das heißt, daß die beschriebenen Attribute bei der Bearbeitung im NXS-Baum nach unten vererbt werden müssen. Bei der Behandlung eines untergeordneten Knoten, müssen die Attribute nach folgendem Schema gesetzt werden.

1. Setze Wert des aktuellen Attributs auf den im Menü der GUI gesetzten Wert (globaler Wert), falls oberster Knoten, oder
2. übernehme Wert des übergeordneten Knotens, falls innerer Knoten,
3. überschreibe den aktuellen Wert des Attributs mit dem in der Hints-Liste aufgeführten Wert.

Bei manchen Hints besteht die Möglichkeit, die Vererbung des Wertes nach unten zu unterbinden. Ist dies der Fall, muß für die Bearbeitung des untergeordneten Knotens der Wert aus 1 oder 2 genommen werden.

Diese Vererbung der Attribute bestimmt somit auch die Abarbeitungsreihenfolge der Knoten der NXS.

3.6.4 Rekursives Vorgehen

Angefangen vom obersten Knoten der NXS, der die komplette zu zeichnende Funktion beschreibt, wird der Baum nach unten hin durchgearbeitet. Besitzt ein Knoten mehrere Söhne, so werden diese in der in der Liste definierten Reihenfolge abgearbeitet.

Die Abarbeitung entspricht auch der graphischen Anordnung der Elemente im Nassi-Shneiderman-Diagramm. Jeder Knoten beschreibt ein Rechteck im Diagramm, dessen Breite durch die Diagrammbreite oder durch die Breite und Art der übergeordneten Knoten bestimmt ist. Die Höhe

des Rechtecks ist durch dessen Inhalt bestimmt.

Bei der Bearbeitung eines Knotens sind daher nur die Position der linken oberen Ecke des Rechtecks und dessen Breite wichtig. Nachdem ein Knoten und dessen Teilbaum bearbeitet ist, beschreibt dieser Teilbaum ein Rechteck im Diagramm, des Position, Breite und Höhe bekannt ist. Für die Bearbeitung der sich im Diagramm anschließenden Knoten sind nur diese Angaben wichtig. Abbildung 3.7 beschreibt das Vorgehen beispielhaft an einem kleinen Diagramm.

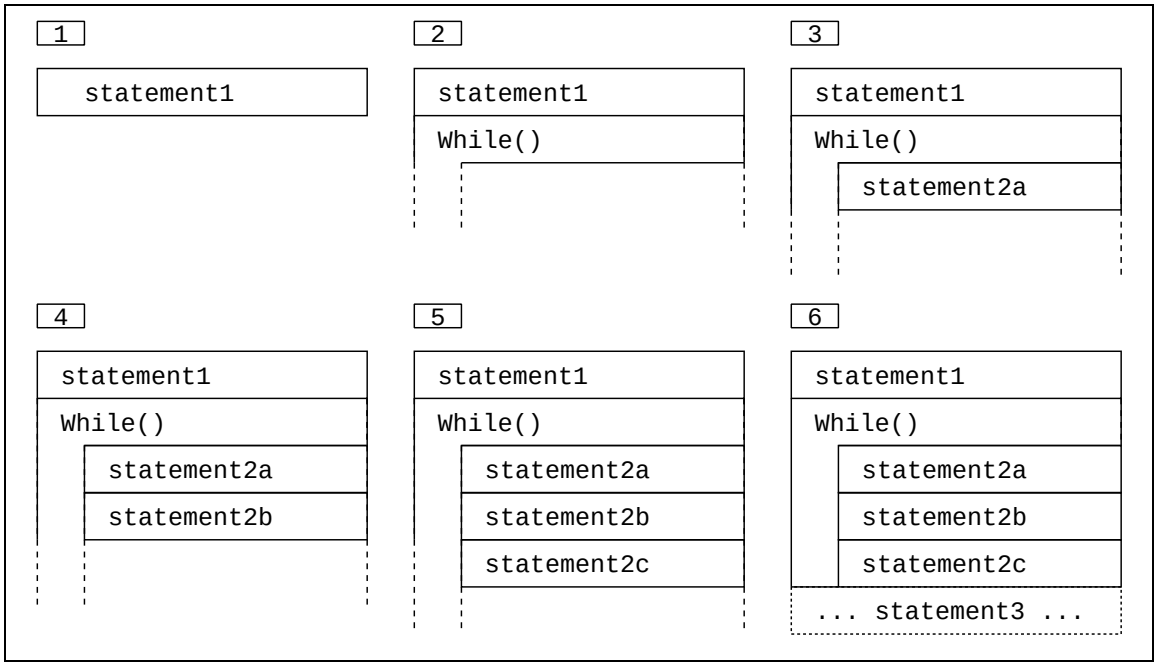


Abbildung 3.7: Rekursives Zeichnen von Anweisungen: Die darstellten Teilabbildungen beschreiben beispielhaft die Phasen, die beim Zeichnen von drei Anweisungen benötigt werden. Im Beispiel besteht die zweite Anweisung aus einer While-Schleife. Bei dem Setzen mehreren Anweisungen (Sequenz) verändert sich Breite und die X-Komponente der Position nicht. Die Y-Komponente wird um jeweils die Höhe der Anweisung nach unten gerückt. in Phase 2 beginnt die Abarbeitung einer While-Schleife, die als erstes die Bedingung im Diagramm mit voller Breite anfügt. Die in der Schleife enthaltenen Anweisungen werden einrückt dargestellt. Daher verändert sich für die Phasen 3-5 die Breite und auch die X-Komponente der Position. Erst nachdem die Anweisungen der While-Schleife bearbeitet sind, die dem untergeordneten Teilbaum entsprechen, kann die While vollendet werden. Danach ist die Breite und X-Komponente wieder maximal.

Die Entwicklung des Diagramm schreitet immer nach unten weiter. Erst wenn der komplette Baum des NXS bearbeitet ist, steht die Höhe des Diagramms fest.

Wie schon beschrieben, ist für die Bearbeitung eines Knoten nur die Position und Breite des zur Verfügung stehenden Bereiches wichtig. Daher bietet sich ein rekursives Vorgehen an. Bei der Bearbeitung der Anweisungen 2a-2c in Abbildung 3.7 spielt es keine Rolle, ob sie sich innerhalb einer While-Schleife oder auch der obersten Ebene befinden. Für komplexe Anweisungen, wie Schleifen oder Bedingungen, gibt es typischerweise Fragmente, die vor bzw. nach der Bearbeitung der darin enthaltenen Knoten gesetzt werden müssen. In der While-Schleife wird der Kopf zuerst, dann die inneren Knoten und abschließend die unterer Linie gezeichnet.

Der nächste Abschnitt beschreibt nun das Vorgehen für jeden Knotentyp. Durch die Definition des lokalen Vorgehens für einem Knoten(-typ) ist durch die Rekursion auch das globale Vorgehen und damit die Konstruktion des ganzen Diagramms definiert.

3.6.5 Behandlung der einzelnen Elemente

Das einfachste Element bei der Darstellung ist die ***Anweisung***, die aus einem Rechteck und dem Anweisungstext besteht. Der Text der Anweisung wird typischerweise in Blocksatz gesetzt, so daß

alle bis auf die letzte Zeile des Textes auf dieselbe Breite aufgeweitet werden. Der Umbruch des Textes auf mehrere Zeilen, dessen Formatierung und die Aufweitung wird im Abschnitt 3.6.7 auf Seite 39 erläutert.

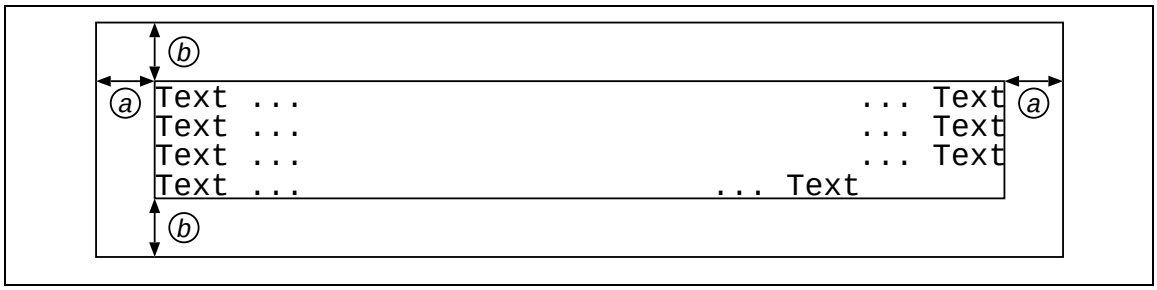


Abbildung 3.8: Schematische Darstellung einer einfachen Anweisung

Der Text wird um die Breite *a* eingerückt und um die Höhe *b* nach unten versetzt gezeichnet. Die Breite des Textes ist um $2 * b$ verringert. Die abschließende Linie des Rechtecks befindet sich im Abstand *b* unterhalb des Textes. Als Graphikelemente werden nur *box* und *text* benötigt.

Die ***Schleifen***, die durch die Typen While und Repeat dargestellt werden, zeichnen sich durch ein Kopf- bzw. Fußteil (Bedingung) und einem eingerückten Anweisungsteil aus. Für beide gibt es eine Einrücktiefe *a* und Höhenanstand *b* des Bedingungstextes und die Einrücktiefe *c* und Abstand der inneren Anweisungen *d*.

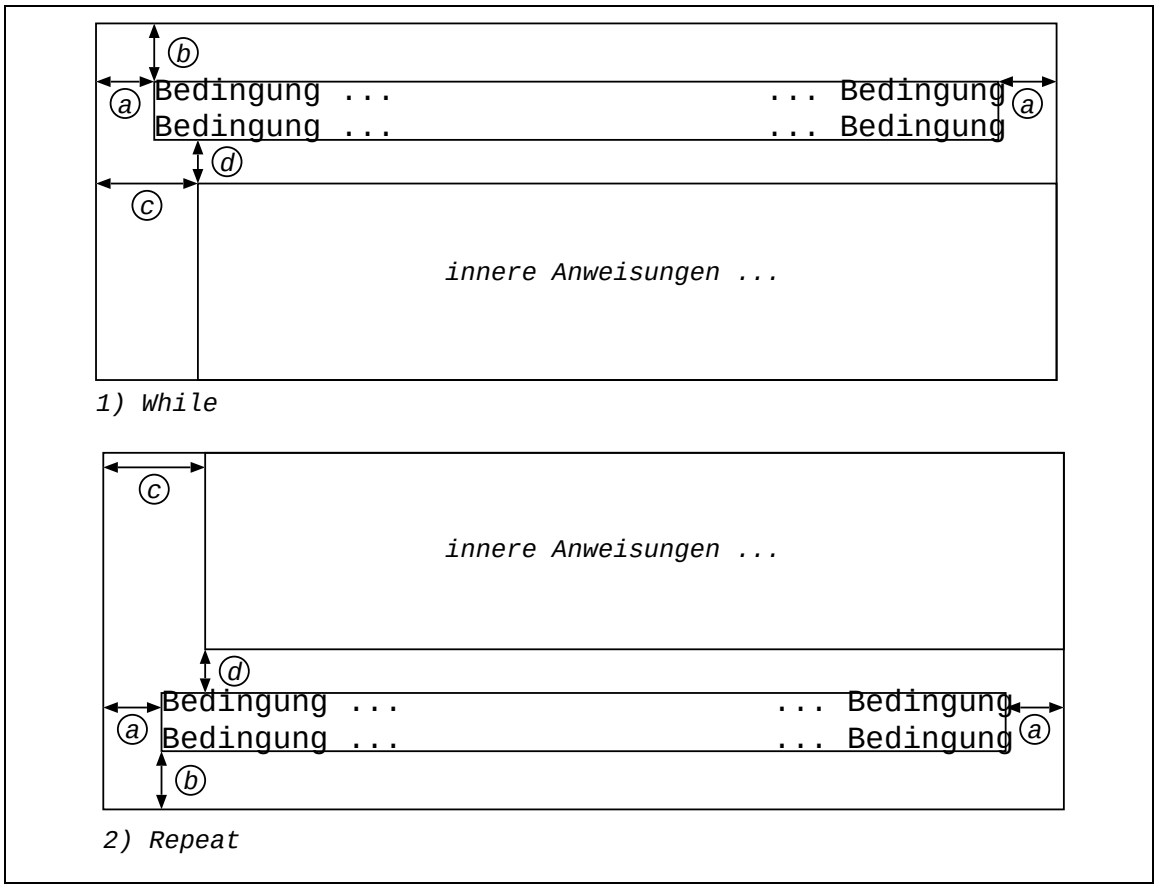


Abbildung 3.9: Schematische Darstellung einer Schleife

Die Breite des Rechtecks, das für die inneren Anweisungen zur Verfügung steht, ist um *c* verringert. Die X-Komponente der Position ist um *c* vergrößert. Im wesentlichen unterscheiden sich die Schleifetypen nur durch den Zeitpunkt, an dem die Bedingung gezeichnet wird, bei der While-Schleife vor der Abarbeitung der inneren Knoten, bei der Repeat nach deren Abarbeitung.

Erst wenn die inneren Anweisungen (untergeordnete Knoten) vollständig bearbeitet sind, steht auch die Höhe des Schleifenkörpers fest. Dann kann das Zeichnen der Schleife beendet werden.

Bei der ***If-Anweisung*** stehen dem Generator während der Optimierung der Darstellung mehr Freiheitsgrade zur Verfügung. Laut Definition der Nassi-Shneiderman-Diagramme müssen die Zweige des If-Statements nebeneinander dargestellt werden. Würde die vorhandene Breite in zwei gleiche Teile geteilt, und die Anweisungen der beiden Zweige in der üblichen Art gezeichnet, entständen Probleme, die in der folgenden Abbildung (3.10) zu sehen sind.

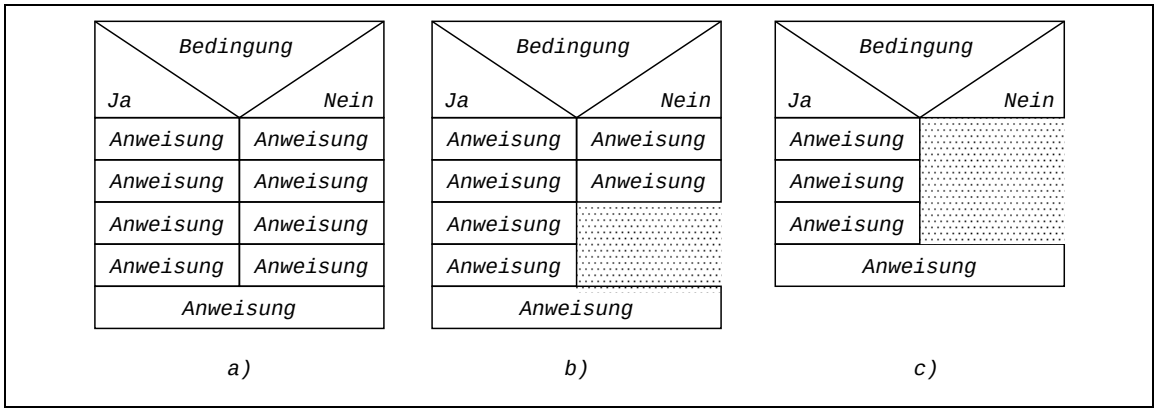


Abbildung 3.10: Darstellung eines If-Statement ohne Optimierung: Nur wenn in beiden Zweigen die gleiche Menge von Anweisungen enthalten ist, ergibt die Anordnung ein volles Diagramm. Enthält eine Seite weniger Text, ist diese kürzer und läßt am Ende eine Lücke. Fehlt eine Spalte, bleibt im Diagramm diese Seite leer.

Zur Vermeidung dieser Probleme, ist eine Aufweitung der kürzeren Spalte unumgänglich. Dabei sollten alle alle vertikalen Abstände um einen für die Spalte konstanten Faktor vergrößert werden. Dadurch wird ein Ausgleich zwischen den Spalten hergestellt. Ist eine leere Spalte vorhanden, sollte dort ein leeres Rechteck gezeichnet werden. Abbildung 3.11 zeigt diese Darstellung.

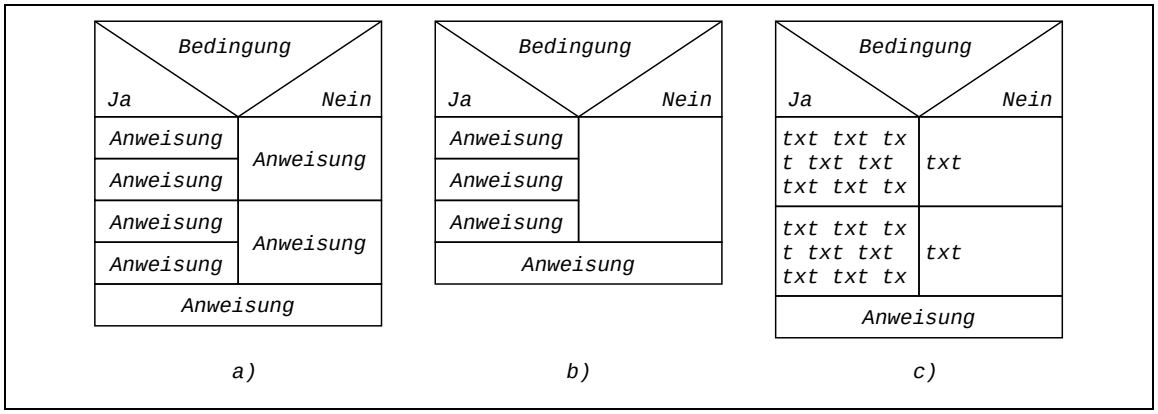


Abbildung 3.11: Darstellung eines If-Statement mit Längenausgleich

Da sich in einem Zweig der If-Anweisung weitere If-Anweisungen befinden können, die ihrerseits wieder solche Aufweitungsfaktoren definieren, muß ein rekursive Durchreichen der Faktoren angewandt werden. Jeder Knoten erhält vom übergeordneten Knoten einen Aufweitungsfaktor, der für die Aufweitung einer darunter liegenden Spalte mit einem weiteren Faktor multipliziert wird. Für den ranghöchsten Knoten des Baumes wird der Faktor auf 1 gesetzt.

Die Bearbeitung der einer If-Anweisung ist aufwendiger als die einer einfachen Anweisung, da mit der beschriebenen Optimierung genau zwei Durchläufe durch die Zweige nötig sind. Im ersten Durchlauf wird das Zeichnen mit dem unveränderten Faktor durchgeführt. Die Längen der so entstehenden Spalten definieren den Faktor, um den die kürzere Spalte aufgeweitet werden muß. Ist a die Länge der kürzeren, b die Länge der größeren Spalte, so ergibt sich der Faktor zu b/a. Im zweiten Durchlauf der Teilbäume, der den neuen Faktor mit berücksichtigt, werden die Spalten ausgeglichen dargestellt. Ist ein Zweig der If-Anweisung nicht vorhanden, entfällt der erste Durchlauf, da die verbliebene Spalte mit unveränderten Faktor gezeichnet werden kann.

Teil c) der Abbildung 3.11 auf der vorherigen Seite zeigt ein weiteres Layout-Problem der If-Anweisung. Ist in einem Zweig viel, im anderen wenig Text vorhanden, müssen sich beide Spalte den gleichen Raum teilen. Die vollere Spalte bestimmt durch zusätzliche Zeilenumbrüche die Länge der If-Anweisung, wobei die Anweisungen der leerere Spalte aufgeweitet werden muß. Dieses Ungleichgewicht der Textdichte kann durch die Verschiebung der Mitte, also die Veränderung der Spaltenbreiten kompensiert werden. Erhält die vollere Spalte einen größeren Anteil der Breite, kann ihr Text besser formatiert werden. Auf der Seite der leereren Spalte entstehen keine Nachteile, da hier nur Freiräume verschwinden.

Ein optimaler Mittelpunkt ist gefunden, wenn die Gesamtlänge der If-Anweisung minimiert ist. Die Länge einer Spalte hängt bei variabler Breite von vielen Faktoren ab. So können bei minimaler Änderung der Breite, Textumbrüche und Trennregel zu einer ganz anderen Anordnung und Länge der Spalte führen. Die Länge als Funktion der Spaltenbreite ist also nicht stetig, so daß eine Berechnung des Mittelpunktes über die Minimierung der Funktion nicht möglich ist.

In einem solchen Fall bietet sich ein iteratives Verfahren an, das den Mittelpunkt solange verändert, bis das ein Minimum der Gesamtlänge erreicht ist. Da bei vielen Eingabenprogrammen die If-Anweisungen mehrfach verschachtelt sind und das iterative Verfahren bei jeder If-Anweisung angewendet werden muß, steigt die Anzahl der Iterationen mit der Schachtelungstiefe exponentiell an.

Für *Nassi* ist dieses Verfahren zu aufwendig. Daher wird nur eine ungefähre Berechnung des Mittelpunkts vorgenommen. Dazu wird für jeden Zweig ein Gewicht berechnet, das aus einer Bewertung der darin enthaltenen Knoten ergibt. Die Gesamtbreite wird dann im umgekehrten Verhältnis der Gewichte zugeordnet. Das Gewicht eines Knotens ergibt sich im wesentlichen aus der Anzahl der Zeichen die im Diagramm erscheinen. Strukturen wie Schleifen oder If- bzw. Switch-Anweisungen werden mit einem konstanten Anteil (20) eingerechnet.

Die Wichtung der Spalten erfordert einen weiteren Durchlauf durch die Zweige, der zuerst ausgeführt werden muß. Das Zeichnen der Elemente ist hier nicht nötig. Der Aufwand ist also geringer als bei den anderen Durchläufen. Durch die Veschachtelung der If-Anweisungen steigt der Aufwand aber weiterhin exponentiell an.

Insgesamt ergeben sich bei der Berechnung einer If-Anweisung drei Durchläufe:

1. Bestimmung des Spaltengewichts für die Berechnung des Mittelpunkts,
2. Zeichnen mit unveränderten Aufweitungsfaktor zur Bestimmung des neuen lokalen Aufweitungsfaktors für die kürzere Spalte,
3. korrektes Zeichnen der Spalten mit angepaßten Faktoren.

Abbildung 3.12 zeigt Beispiele für If-Anweisungen, bei deren Gestaltung alle oben aufgeführten Regeln angewendet worden sind.

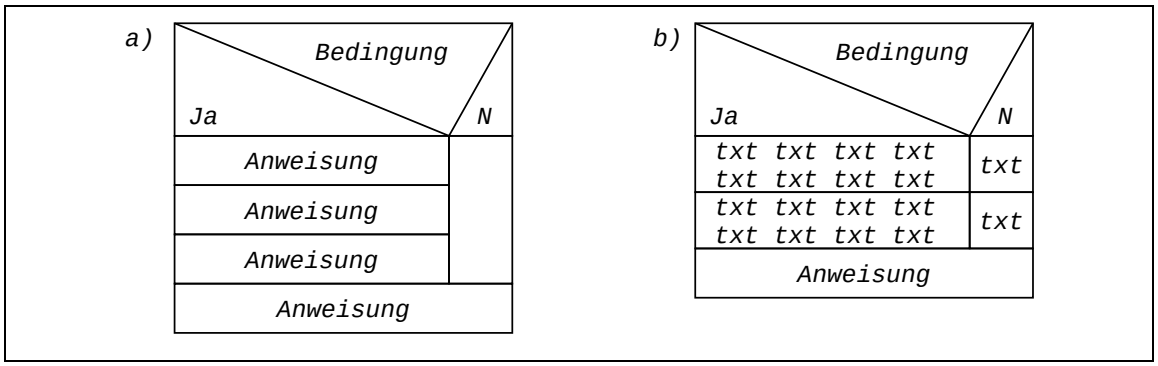


Abbildung 3.12: Darstellung eines If-Statement mit Längenausgleich und Gewichtung der Spalten: Teil a) zeigt, daß die Gewichtung keinen Vorteil bei der Länge des Diagramms bringt. Begründet ist dies dadurch, daß alle Anweisung einzeilig sind und die eine größere Breite nicht kürzer werden können. Trotzdem bringt hier die Gewichtung eine übersichtlichere Darstellung, da innerhalb des Diagramm der gefüllten Then-Spalte mehr Platz eingeräumt wird. In Teil b) ist durch die Gewichtung der Umbruch der Zeilen wesentlich besser und die Diagrammlänge verkürzt.

Die Darstellung der Bedingung bringt einige Erweiterungen bei Anwendung des Blocksatzes. So soll der Text möglichst platzsparend in das dafür vorgesehene Dreieck eingepaßt werden. Die X-Komponente des unteren Dreieckspunktes ist durch die Gewichtung der Spalten gegeben. Der Generator muß das Dreieck solange nach unten erweiten, bis das der komplette Text der Bedingung in das Dreieck eingepaßt werden kann. Dabei wird die Y-Komponente des unteren Punktes im Schritten der Zeilenhöhe erweitert. Da sich dadurch auch die Breite alle Zeilen im Dreieck ändert, muß bei jeder Iteration das gesamte Text neu gesetzt werden. Abbildung 3.13 zeigt das Vorgehen.

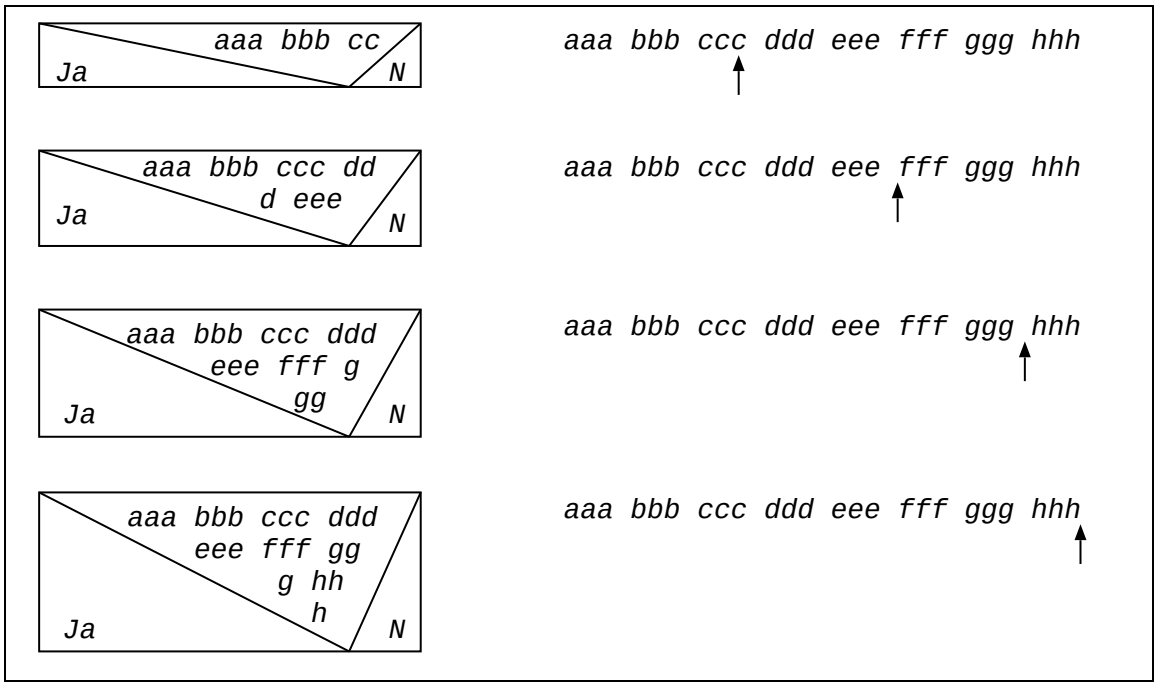


Abbildung 3.13: Blocksatz im If-Kopf: Erst nach vier Schritten ist der Text der Bedingung in das Dreieck eingepaßt. Auf der rechten Seite ist jeweils der komplette Text und die bisher erreichte Position zu sehen. Die Breite und Position der einzelnen Textzeilen ist durch die schrägen Linien der Seitenbegrenzung und den normalen seitlichen Abstände zum Text gegeben. Da die Seitenlinien bei eine Aufweitung des Dreiecks nach unten weiter vom Text wegrücken ist auch in den ersten Zeilen mehr Platz. Kann eine Zeile nicht ganz gefüllt werden, sollte deren Inhalt zwischen den beiden Begrenzungen zentriert werden.

Die Formatierung des If-Kopfes verändert nur dessen Höhe. Daher kann das Zeichnen des Kopfes schon nach dem ersten Durchlauf (Gewicht) geschehen. Die Literale `Ja` und `Nein` sollten bei schmalen Spalten abgekürzt oder weggelassen werden.

Zusätzlich zum automatischen Berechnen des Mittelpunktes bietet der graphische Editor auch die Manipulation per Hand an. Das neue Gewicht der Spalten wird dem Generator als Hint zur Verfügung gestellt. Ist das der Fall, kann der erste Durchlauf entfallen.

Die ***Mehrfachverzweigung***, die in der Programmiersprache C durch die Switch-Anweisung und in Pascal durch die Case-Anweisung implementiert sind, müssen entsprechend der If-Anweisung mit mehreren Durchläufen bearbeitet werden.

Abbildung 3.14 auf der nächsten Seite zeigt die Darstellung einer Mehrfachverzweigung. Die Anzahl der Fälle und damit die Anzahl der Spalten ist variabel. Wie beim If-Statement werden die einzelnen Spalten nebeneinander gesetzt. Der Alternativ-Zweig (default), der bei keinem Treffer durchlaufen wird, steht als letzte Spalte in der Anordnung und bestimmt den unteren Punkt des für die Bedingung (Expression) zur Verfügung stehenden Dreiecks.

Im ersten Durchlauf werden die Gewichte (g_i) der einzelnen Spalten berechnet. Die Breite der Spalten wird durch $\frac{g_i b_{\text{ges}}}{\sum_i g_i}$ bestimmt. Danach muß zuerst der Kopf gezeichnet werden. Durch in der Abbildung 3.14 auf der nächsten Seite gezeigten Anordnung der Spalten ergeben sich für die einzelnen Spalten verschiedenen Anfangshöhen. Da in den meisten Fällen die Dreiecke zwischen

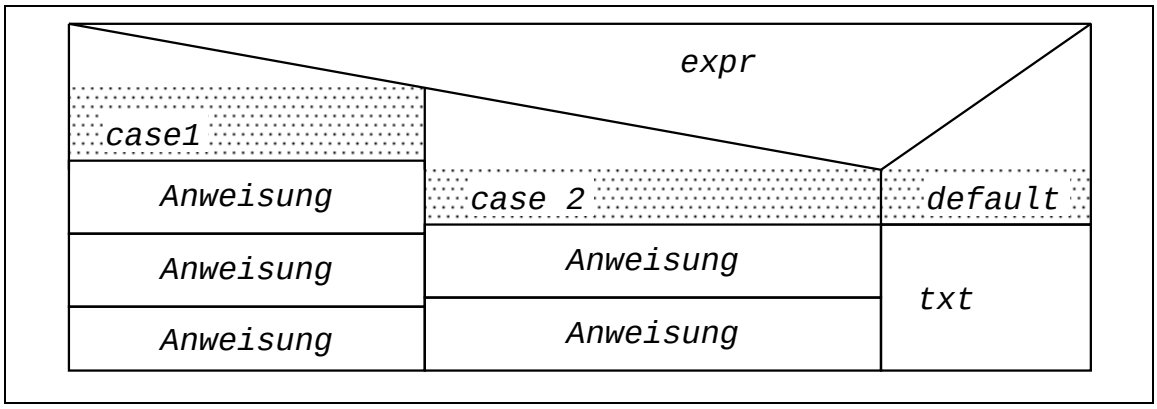


Abbildung 3.14: Schematische Darstellung einer Mehrfachverzweigung

Begrenzungslinien und Oberkante der Spalte für die Darstellung des Ausdrucks zu klein ist, wird dieser soweit nach unten verschoben, daß eine ganzes Rechteck (in Abbildung grau hinterlegt) entsteht.

Im zweiten Durchlauf werden die Spalten mit dem unveränderten Aufweitungsfaktor der übergeordneten Knoten gezeichnet. Die Aufweitungsfaktoren der Spalten ist dann durch die längste Spalte gegeben, auf deren Länge alle andere Spalten aufgeweitet werden müssen.

Im dritten Durchlauf werden die Spalten dann mit den veränderten Faktoren gezeichnet. Die Spalten sind damit ausgeglichen.

Bei Switch-Anweisungen treten meistens sehr viele Zweige auf. Überschreitet die Anzahl eine gewisse Grenze, sollte auf eine alternative Darstellung umgeschaltet werden. In dieser werden die einzelnen Zweige hintereinander aufgelistet.

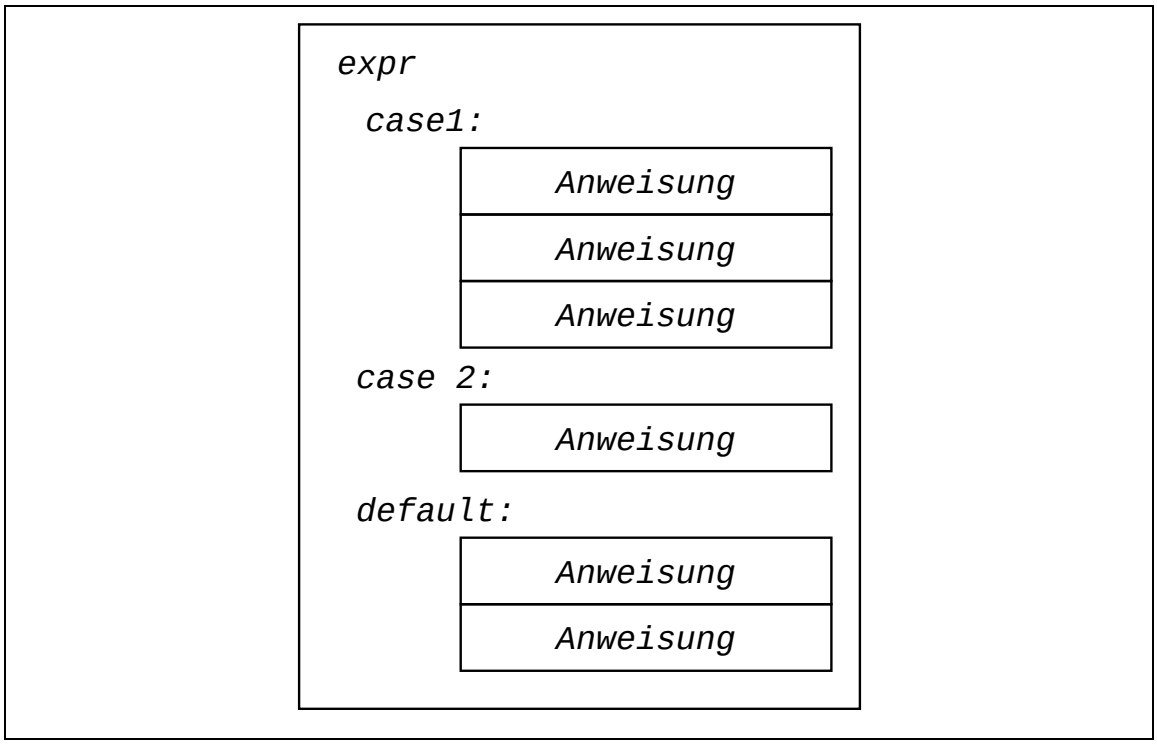


Abbildung 3.15: Alternative Darstellung einer Mehrfachverzweigung

Diese Darstellung ist zwar nicht ein direkter Bestandteil der Nassi-Shneiderman-Diagramme, ermöglicht aber auch die Analyse großer Programme. Würden bei der normalen Darstellung zu viele Spalten gezeichnet, enstehen Probleme beim Textumbruch. Die alternative Darstellung eignet sich besonders für die Switch-Anweisungen der Programmiersprache C. Dort wird der Programmfluß durch das break-Statement bestimmt, der vom Programmierer an letzter Stelle eines Zweiges set-

zen kann. Fehlt dieses Statement, wird der nächster Zweig auch durchlaufen. In solchen Fällen entspricht die alternative Darstellung eher der Wirklichkeit. *Nassi* sollte nicht versuchen, die break-Statement bei der normalen Darstellung zu entfernen. Dies würde die korrekte Darstellung des Eingabeprogramms verhindern. Zum Beispiel würde beim Fehlen des break-Statement in nur einer Spalte zu einem Informationsverlust führen.

3.6.6 Exclude-Blöcke

In Abschnitt 2.1 auf Seite 3 wurden schon Lösungsvorschläge bei der Darstellung von großen Routinen aufgezeigt. Im Entwurf muß dabei die Auslagerung von Blöcken in eigene Diagramme definiert werden. Diese sogenannten *Exclude*-Blöcke ermöglichen eine Vereinfachung der Diagramme von Benutzerseite aus. Im graphische Editor oder direkt über die NSC-Kommentare kann bei einem Block das Attribut `exclude` gesetzt werden. Das Element `etext` im NXS speichern dann den Namen oder die Beschreibung des Exclude-Blockes.

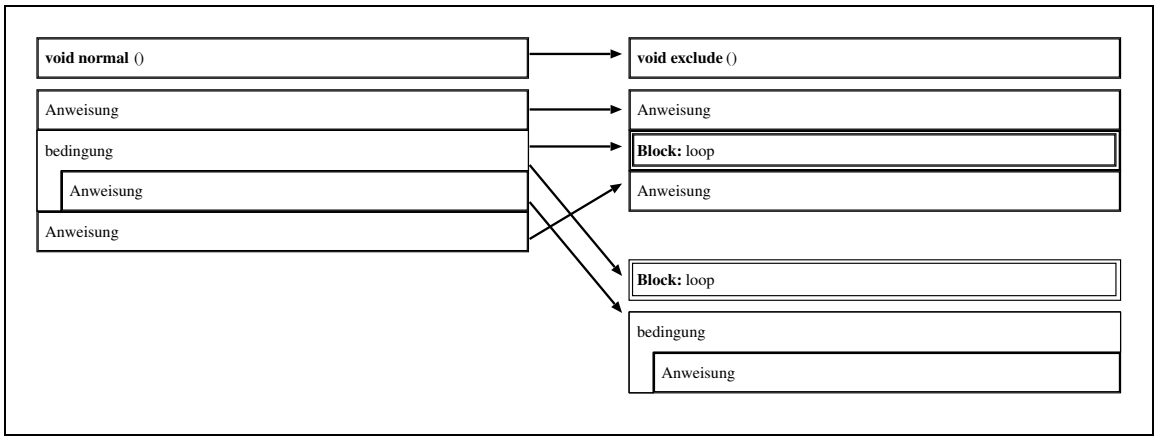


Abbildung 3.16: Auslagerung eines Blockes

Für den Generator bedeutet dies eine doppelte Behandlung des Blockes. Tritt der Block innerhalb eines Nassi-Shneiderman-Diagramms auf, also nicht als oberster Knoten des NXS-Baumes, handelt es sich um das Diagramm, aus dem der Block ausgelagert werden soll. Tritt er als oberster Knoten auf, besteht das Diagramm aus diesem Block und es muß zusätzlich ein Kopf gezeichnet werden.

Würde immer nur ein Diagramm vom Generator behandelt, könnte aufgrund der Position im NXS-Baum der Fall ausgewählt und entsprechende Graphikobjekte in der Teilstruktur `graph` der NXS gespeichert werden. Da aber sowohl der Generator als auch die Ausgaberoutine eine Liste von Diagrammen (NXS-Bäumen) akzeptieren, tritt der Knoten ggf. in mehreren NXS-Bäumen auf. Im Speicher ist er aber nur einmal vorhanden. Daher muß dafür gesorgt werden, daß der Knoten in beiden Diagramme die richtigen Graphikobjekte enthält, was in der Teilstruktur `graph` der NXS durch den zusätzlichen Eintrag `egraph` geschieht. Dieser Eintrag enthält eine weitere `graph`-Teilstruktur, die die Repräsentation des Exclude-Blockes im ausgelagerten Diagramm enthält (Kopf und Elementobjekte). In der `graph`-Struktur steht immer nur die Repräsentation des Exclude-Blockes im übergeordneten Diagramm.

Die Ausgaberoutinen der anderen Module müssen anhand der Position im NXS-Baum auswählen, welche Graphikobjekte gezeichnet werden.

3.6.7 Textdarstellung und dessen Umbruch

Für den Benutzer besteht die Möglichkeit, das Aussehen des Textes zu verändern. Dazu können für jedes Statement zwei Schriftarten angegeben werden. Die erste wird für normalen Text, die zweite für Schlüsselwörter oder ähnliches genutzt. Die vorhandenen Schriftarten variieren je nach Ausgabeformat. Um trotzdem eine ein einheitliche Auswahl der Schriftarten zu gewährleisten, wurden drei Attribute definiert, die eine Schriftart beschreiben. Dies sind der Name der Schriftfamilie (`FONT_NAME`: *times, ...*, das Format (`FONT_FORMAT`: *normal, italic, ...*) und die Textgröße (`FONT_SIZE`: *8,9,10, ...*). Die Zuordnung einer realen Schriftart aus dem ausgewählten Ausgabeformat nimmt das Modul *FontGen* vor. *FontGen* stellt zusätzlich die Größeninformationen der Schriftarten zur Verfügung. Mit diesen kann der Generator die Ausdehnung eines Textes berechnen.

Der Text eines Statements paßt oft nicht in eine Zeile, so daß eine Trennung vorgenommen werden muß. Grundsätzlich werden Zeilenumbrüche nur zwischen zwei Token der Eingabesprache oder bei Leerzeichen innerhalb von Strings eingefügt.

Der Benutzer kann zusätzliche optionale Trennstellen für Texte definieren, die nur im Bedarfsfall beachtet werden.

Es können lokale Trennstellen vorgegeben werden, die nur für die nächste Struktur gelten, und globale Trennstellen, die für das gesamte Eingabeprogramm gelten. Die Trennvorschläge sind dann in den Hints `HYPHEN` und `G_HYPHEN` gespeichert.

Bei den Trennstellen wird zwischen Trennung mit Bindestrich und ohne unterschieden. Um eine eindeutige Kennung bei der Eingabe der Stellen zu erhalten, werden 2 verschiedene Symbole benutzt (# ohne Trennsymbol, $\sim$ mit Trennsymbol). Gerade bei technischen Texten ist eine Trennung mit Bindestrich verwirrend. Eine Trennung ohne Bindestrich ist dann die bessere Alternative.

Der Generator soll die Trennstellen nur dann behandeln, wenn ein Zeilenumbruch nötig ist. Dann wird geprüft, ob ein Bindestrich eingefügt werden soll und kann. Wenn dieser ebenfalls nicht mehr in die Zeile paßt, wird das komplette letzte Textstück in die nächste Zeile geschrieben. Falls eine Zeichenkette keine Trennstellen besitzt und nicht mehr in die Zeile paßt, wird das Textstück hart abgetrennt. Dabei werden soviele Zeichen wie möglich (mind. 2) in der alten Zeile belassen. Wird dabei zwischen zwei Buchstaben getrennt, muß zusätzlich ein Bindestrich eingefügt werden.

Dem Generator stehen die Texte als eine Liste von `ptext`-Strukturen zur Verfügung. Die Texte sind dort vom Parser schon in die einzelnen Token aufgespalten worden.

Zur Bearbeitung und Formatierung werden die Texte in einer Liste von `Gtext`-Strukturen der NXS gespeichert. Diese neuen Strukturen speichern nun neben dem Text auch die Position, Ausdehnung und die Schriftart im Ausgabeformat. Innerhalb eines Statements sind nur zwei Schriftarten vorhanden[1]. Diese beiden Schriftarten werden vom `FontGen`-Modul auf reale Fonts abgebildet. Einen Hinweis auf einen der beiden realen Fonts wird innerhalb der `Gtext`-Struktur gespeichert.

Bevor die Formatierung der Texte durchgeführt wird, werden die einzelnen Token der `Gtext`-Liste mit den Trennvorschlägen verglichen. Bei einer Übereinstimmung wird das entsprechende Token gemäß den Trennvorschlägen aufgeteilt. Dabei werden die Trennvorschläge als eigenständige Elemente der `Gtext`-Liste abgespeichert und besonders markiert (Länge=-100). Trennvorschläge ohne Bindestrich müssen nicht vermerkt werden. Da das Token an dieser Stelle aufgetrennt wird, ist dieser Vorschlag dann implizit eingebunden.

Leerzeichen, die nicht in Stringkonstanten stehen, werden vom Parser zu einem Token (`SPACE`) zusammengefaßt. Stringkonstanten werden vom Parser als ein Token belassen, so daß dort mehrfache Leerzeichen erhalten bleiben.

Bei allen Token wird die Länge als Summe der Länge der einzelnen Zeichen berechnet und abge-

[1] Eine weitere Verfeinerung der, z.B. Schriftwechsel innerhalb eines Statements, ist dieser Version von Nassi nicht vorgesehen. Dazu wäre eine nicht struktur-orientierte Markierung innerhalb des Eingabeprogramms nötig.

speichert. Erst bei der Formatierung kann das letzte Feld (Position) gefüllt werden.

Danach steht der Text des Statements in der `Gtext`-Liste maximal verfeinert zur Verfügung. Der Umbruch kann nun durchgeführt werden. Dazu wird der in der Abbildung 3.17 dargestellte Zustandsübergangsgraph durchlaufen.

Die fünf Zustände des Graphen haben folgende Bedeutung:

`FITS` normaler Umbruch: Die Wörter (Token) werden in der Zeile gesetzt. Erst wenn das aktuelle Wort nicht mehr paßt oder ein Trennzeichen gefunden wird, wird in einen anderen Zustand übergegangen. Ist kein Wort mehr vorhanden, kann das Verfahren beendet werden, indem zum Zustand `E` übergegangen wird.

`TILDE` Es wurde ein Trennzeichen gefunden. Paßt auch das übernächste Token noch in die Zeile, kann das Trennzeichen vernachlässigt werden, der Trennvorschlag wird nicht benötigt. Paßt ansonsten das Trennzeichen, wird dieses gesetzt und ein Zeilenumbruch durchgeführt. Paßt auch das Trennzeichen nicht ist eine Sonderbehandlung nötig.

`HARD` Das aktuelle Wort muß aufgetrennt werden. Es werden soviele Zeichen in diese Zeile gesetzt, bis diese zusammen mit einem Bindestrich gefüllt ist. Könner weniger als zwei Zeichen abgetrennt werden, wird das ganze Wort in die nächste Zeile geschoben. Ist das aktuelle Wort auch noch das erste in der Zeile, konnte überhaupt kein Zeichen in die Zeile gesetzt werden und das Verfahren muß mit einem Fehlercode abgebrochen werden (z.B. bei If-Bedingungen).

`CRLF` Eine Zeile ist gefüllt und wird beendet.

`RECOVERY` Hinter diesem Wort konnte der Bindestrich nicht mehr gesetzt werden. Daher muß dieses Wort in die nächste Zeile verschoben werden. Ggf. muß eine Trennzeichen vor diesem Wort gesetzt werden. War dieses Wort das erste in der Zeile muß, um eine Endlos-Schleife zu verhindern, ein harter Umbruch erfolgen.

Folgende Aktionen werden im Diagramm durchgeführt.

`setze Wort` Die Position des Wortes wird auf den Wert der aktuelle Position gesetzt. Die

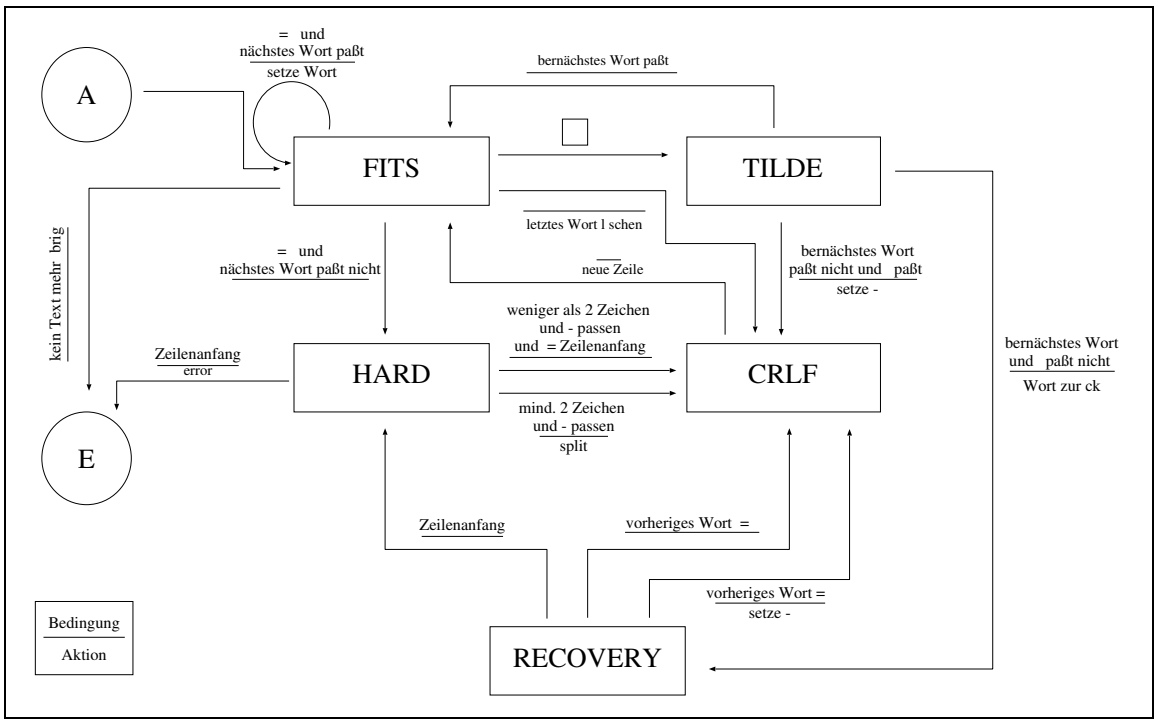


Abbildung 3.17: Zustandsübergangsgraph für Texttrennung: Der Graph wird vom Zustand `A` an durchlaufen. Ist die Bedingung oberhalb eines Pfeiles erfüllt, wird die Aktion unterhalb der Pfeiles ausgeführt und der Zustand wechselt zum Zielpunkt der Pfeiles. Das Verfahren ist beendet, wenn der Zustand `E` erreicht ist.

	aktuelle Position wird um die Länge der Wortes nach rechts verschoben.
`setze -`	setzt den Bindestrich als Trennung zwischen zwei Worten.
`Wort zurück`	Das letzte gesetzte Wort muß wieder an den Anfang der Liste der noch zu bearbeitenden Worte gespeichert werden, da eine andere Behandlung nötig ist.
`split`	Das aktuelle Wort wird aufgetrennt. Der erste Teil wird in die aktuelle Zeile gesetzt, der zweite Teil bleibt am Anfang der Bearbeitungsliste stehen.
`neue Zeile`	Die Anfangsposition einer neuen Zeile wird berechnet. Auch die Breite dieser Zeile wird hier berechnet. Dadurch ist auch eine Formatierung mit anderen Konturen, z.B. Dreieck bei If-Statements, möglich.

Das mit diesem Zustandsübergangsgraph beschriebene Verfahren ermöglicht eine einfache Implementierung des Zeilenumbruchs. Der Algorithmus besteht nur aus einer Schleife und einer Zustandsvariablen. Diese ist am Anfang mit dem Zustand `A` besetzt. Erst wenn die Zustandsvariablen den Wert `E` erhält, wird die Schleife beendet. In der Schleife werden die einzelnen Zustände mit eine Fallunterscheidung behandelt. Für jeden Fall werden die Bedingungen der Pfeile geprüft und bei Erfüllung die entsprechende Aktion durchgeführt und die Zustandsvariable mit dem neuem Wert belegt.

Die als relativ komplex anzusehende Lösung des Textumbruch-Problems kann mit dem Zustandsübergangsgraphen kompakt formuliert und implementiert werden. Erweiterungen, wie z.B. benutzerdefinierte Zeilenumbrüche können einfach als zusätzliche Pfeile und als zusätzliche If-Abfragen in der Abarbeitungsschleife implementiert werden.

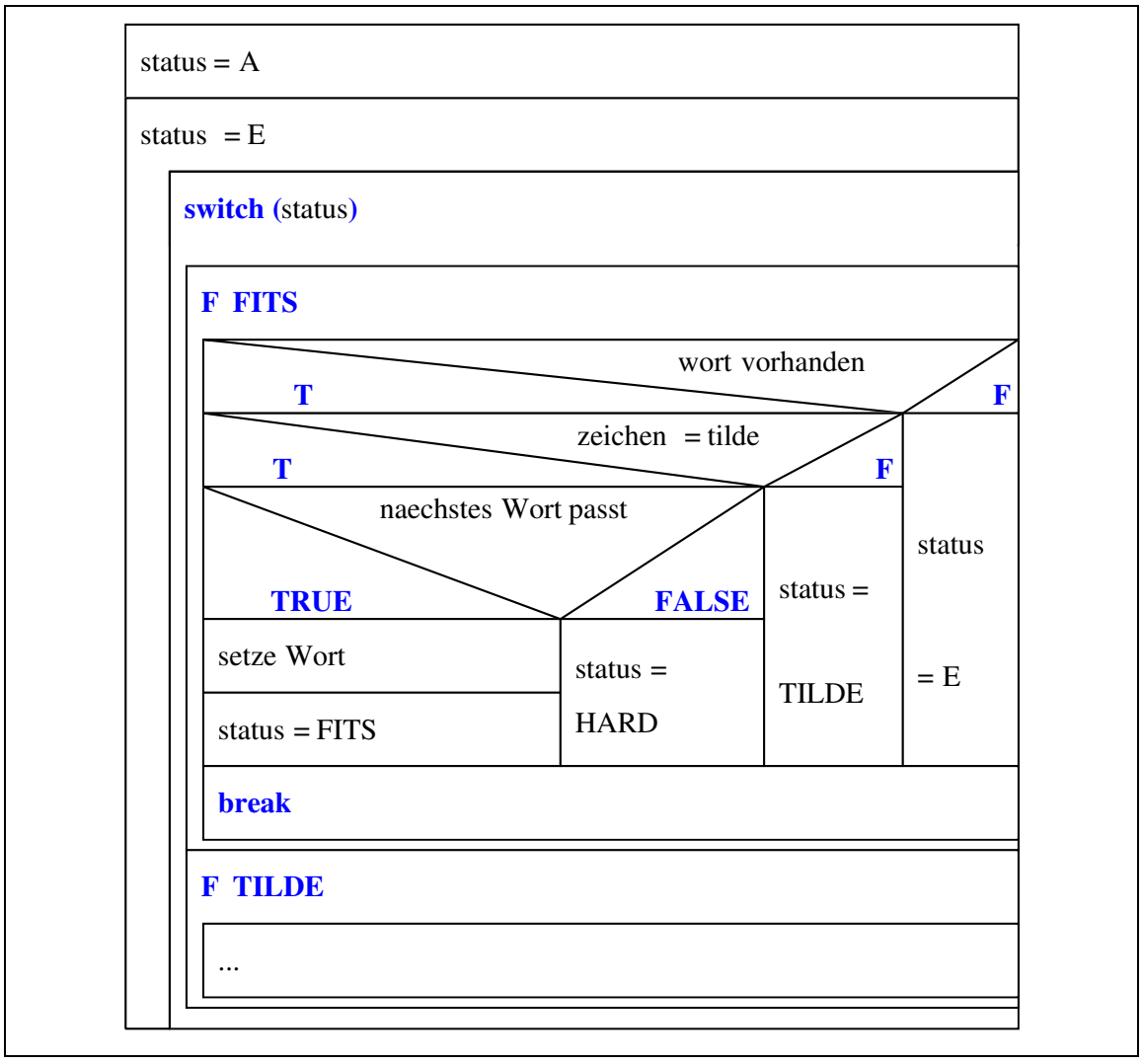


Abbildung 3.18: Nassi-Shneiderman-Diagramm der Textformatierung-Algorithmus

3.7 Modul Ausgabe

Das Ausgabe-Modul behandelt die komplette Ausgabe der Daten in den verschiedenen Formaten. Vorgesehen sind hier PostScript, Encapsulated PostScript (EPS) als druckbare Formate und Tgif und Xfig als Formate für die Weiterbehandlung der Diagramme.

3.7.1 Eingabedaten

Als Eingabedaten erhält das Modul Ausgabe die in den `graph`-Strukturen der NXS gespeicherten Daten (siehe Abbildung 3.6 auf Seite 30). Diese werden vom Modul Generator generiert und sind – bedingt durch die Schriftarten – vom Ausgabeformat abhängig. Beim Aufruf des Output-Formats stimmen die Graphikinformationen aber mit dem in den Optionen gesetzten Format überein, so daß das Modul Output diese nur in die Ausgabedatei schreiben muß. Als weitere Eingabedaten liegen die in den Hints abgespeicherten Werte der Optionen vor. Diese bestimmen im wesentlichen das Ausgabeformat, die Art der Dateinamen und das Layout der Ausgabe.

Art der Option	Beschreibung
OUT_FORMAT	Graphikformat der Ausgabe: *PS* (PostScript), *EPS* (Encapsulated PostScript), *Tgif* (Graphikeditor) und *xfig* (Graphikeditor)
OUT_METHOD	Art der Dateinamen: *name of function:* Bei den Ausgabeformaten *EPS* und *Tgif* wird bei den einzelnen Dateien der Name der dargestellten Funktion, bzw. des Exclude-Blocks genutzt. Leerzeichen werden dabei durch _ ersetzt. *prefix+number:* Für den Namen der Ausgabedatei wird die Option OUT_PREFIX benutzt, der eine laufende Nummer angehängt wird. Die laufende Nummer wird bei jedem Ausgabeaufruf wieder mit 1 initialisiert. *main input filename:* Für den Namen der Ausgabedatei wird der Name der Eingabedatei (ohne `.c`) benutzt, dem wiederum eine laufende Nummer angehängt wird. Die laufende Nummer wird bei jedem Ausgabeaufruf wieder mit 1 initialisiert.
OUT_PREFIX	Enthält den Namen, der mit einer laufenden Nummer versehen bei der Methode *prefix+number* für die Generierung der Dateinamen genutzt wird.
OUT_DIR	beschreibt das Verzeichnis, in dem die Ausgabedateien abgelegt werden. Hier kann sowohl ein relativer (z.B. `./out`) als auch ein absoluter Pfad (z.B. `/tmp/out`) stehen.
OUT_WIDTH	beschreibt die Breite des Diagramms. Die Höhe des Diagramms ergibt sich aus dessen Inhalt.
OUT_PAPER FORMAT	beschreibt das Papierformat: *A4* (DIN-Norm, 21cm∗29.7cm), *letter* (11in∗8.5in) und *legal* (8.5in∗14in)
OUT_PAPER ORIENTATION	definiert, ob das Diagramm hochkant (*portrait*) oder quer (*landscape*) gedruckt wird.
PS_HEADER	bestimmt, ob eine Header-Zeile mit Informationen über die Funktion, Seitennummer und Datum gedruckt wird.
PS_TOC	bestimmt, ob am Anfang der Ausgabe ein Inhaltsverzeichnis gedruckt wird.
PS_FIT	definiert, ob das Diagramm soweit skaliert wird, bis das es auf die Ausgabeseite paßt, oder ob das Diagramm auf mehrere Seiten aufgespalten werden soll.

Tabelle 3.3: Wichtige Hints, die vom Modul Ausgabe benutzt werden

Die letzten fünf Optionen sind nur beim Ausgabeformat PS wirksam, das als einzigstes die Ausgabe mehrerer Diagramme in eine Datei zuläßt.

3.7.2 Ausgabedaten

PostScript

Das wichtigste Ausgabeformat ist PostScript, das direkt zum Drucker geschickt, mit PostScript-Previewern betrachtet und in der EPS-Variante auch in Textverarbeitungssysteme, wie z.B. LaTeX benutzt werden kann. PostScript-Code wird in der umgekehrt-polnisch-Notation geschrieben. In einem Stack werden Parameter und Objekte zwischengespeichert. In sogenannten Dictionaries können Daten, mit einem Namen versehen, abgespeichert werden. PostScript-Code wird in Klartext-ASCII abgespeichert. Das Format ist relativ frei. Literale werden mit / eingeleitet und mit dem Befehl `def` mit einem Wert belegt. PostScript stellt mit den üblichen Konstrukten der Bedingungen und Schleifen und Unterroutinen eine komplette Programmiersprache dar. Meistens wird diese aber nicht genutzt. Es gibt eine Vielzahl von Graphikbefehlen, die eine vektor-orientierte Darstellung von Bilder erlauben. So können direkt Linienzüge, Splines gezogen, Flächen definiert, gefüllt und mit einem Muster versehen werden. Alle Zeichenbefehle beziehen sich auf die aktuelle PostScript-Seite. Das Drucken oder Anzeigen der Seite geschieht erst bei der Interpretation des Befehls `showpage`. Dieser Befehl gibt dem Drucker den Befehl die Zeichenfläche zu drucken. Danach wird die Zeichenfläche geleert und eine neue Seite kann erstellt werden.

Im Koordinatensystem von PostScript liegt der Ursprungspunkt in der linken unteren Ecke der Zeichenfläche. Als Einheit wird $1/72$ Inch (pt) benutzt, so daß eine DIN A4 Blatt in PostScript die Größe $595*842$ pt hat. Die Koordinaten müssen nicht ganzzahlig sein.

Es stehen unter PostScript auch Befehle zur Transformation des Koordinatensystems zur Verfügung, die die Skalierung, Drehung und Verschiebung von Objekten zulassen. Auf diese Weise werden auch EPS-Graphiken in anderen Dokumente eingebunden.

Es gibt in PostScript eine Anzahl von Schriftarten, die in jedem Drucker vorhanden. Weitere Schriftarten können im PostScript definiert und dann mit zum Drucker geschickt werden. Als Standard-Schriftfamilien sind `Times-Roman`, `Helvetica`, `Courier` und `NewCenturySchlbk` vorhanden. Alle Schriftarten liegen in den Formaten roman, *italic* und **bold** vor. Alle Schriftarten müssen für die Darstellung von deutschen Texten um die Umlaute erweitert werden. Die Umlaute sind zwar als Zeichen in der Schriftart vorhanden, werden aber in der Zuordnungtabelle (ASCII→Zeichen) nicht angesprochen. Deshalb müssen alle Schriftarten, die im Diagramm benutzt werden vorher umdefiniert werden.

Über ein besonderes Dictionary (`systemdict`) können dem Drucker Einstellungen bezüglich der Seitengröße mitgeteilt werden. Die Orientierung der Ausgabe ist aber allein Sache der erstellenden Software. Ein Druck in Landscape-Mode muß z.B. mit den Befehlen `90 rotate 0 -842 translate` eingestellt werden.

Die nachfolgende Tabelle gibt den PostScript-Code für die in einem Diagramm möglichen Graphikobjekte an:

Linie	`newpath` *x1 y1* `moveto` *x2 y2* `lineto stroke`
Liniezug	`newpath` *x1 y1* `moveto` *x2 y2* `lineto ...` *xn yn* `lineto stroke`
Box	`newpath` *x1 y1* `moveto` *x2 y2* `lineto ...` *xn yn* `lineto closepath stroke`
gefüllte Box	`newpath` *x1 y1* `moveto` *x2 y2* `lineto ...` *xn yn* `lineto closepath fill`
Text	`/Times-Roman-8 findfont` *size* `scalefont setfont` *x y* `moveto (`*text*`) show`
Farbe	*red green blue* `setrgbcolor`

Tabelle 3.4: PostScript-Code zum Zeichnen der einzelnen Graphikobjekte

Über Kommentare, die bei PostScript mit % eingeleitet werden und bis zum Zeilenende gehen, können Hinweise auf den Anfang einer neuen Seite, genutzte Schriftarten usw. gegeben werden, die von Previewern für die Anzeige genutzt werden.

Zur Erkennung muß in der ersten Zeile der PostScript-Datei die folgende Zeile stehen:
`%!PS-Adobe-2.0 EPSF-1.2`
Die Nummern beschreiben die Versionsnummer der benutzten PostScript-Sprache und EPS-Version.

Encapsulated PostScript (EPS)

Dieses Format entspricht im wesentlichen dem PostScript-Format. PostScript-Code im EPS-Format wird für die Einbindung in Textverarbeitungsysteme genutzt und muß daher einigen Einschränkungen genügen. So darf zum Beispiel kein Befehl genutzt werden, der die Ausgabeformate verändert (aus `systemdict`). Weiter soll der Befehl `showpage` nicht genutzt werden, da die Graphik ja in anderen PostScript-Code eingebunden wird und dieser für die Ausgabe der Seite sorgt.

Wesentlich sind auch die PostScript-Structure-Comments, die mit %% eingeleitet werden und weitere Informationen über die Graphik lesen. Zum Beispiel steht dort auch die Größe der Graphik (in PostScript-Koordinaten), die dem Textverarbeitungsystem Information über den freizulassenden Platz auf der Seite gegeben. Dieser BoundingBox-Kommentar sollte am Anfang der Datei stehen und hat folgende Gestalt: `%%BoundingBox: x1 y1 x2 y2`
Die Koordinaten `x1 y1 x2 y2` beschreiben die linke untere und die rechte obere Ecke der benutzten Zeichenfläche.

Tgif

Tgif speichert die Graphiken in einer Klartext-ASCII-Datei. Die darin enthalten Befehle besitzen sehr viele Parameter, deren Bedeutung nur aus dem Source-Code von Tgif entnommen werden kann. Das Tgif-obj-Format ist im wesentlichen für die Speicherung und weniger für den Austausch mit anderen Programmen gedacht. Zum Beispiel sieht eine Befehl für die Darstellung einer Linie folgendermaßen aus:

```
poly('black',2,[
        304,128,496,112],0,1,1,23,0,0,0,0,8,3,0,0,0,'1','8','3',
    "0",[  ]).
```

Eine Public Domain Library `TGIFCRTL` bringt hier Abhilfe. Diese von Mats Bergstöm (University of Lund, Sweden, Mats.Bergstrom@kosufy.lu.se) entwickelte Library stellte eine Anwendungsschnittstelle (API) in C zu den obj-Dateien von Tgif dar.

Diese API bietet Befehle an, die eine direkte Kodierung der Graphikobjekte von *Nassi* erlauben. Ein Skalierung-Befehl, der am Anfang abgesetzt wird, erlaubt auch eine direkte Angabe der Nassi-Koordinaten beim Aufruf der API-Funktionen. Der Ursprungspunkt des Koordinatensystems von Tgif ist die linke obere Ecke. Daher ist trotz des Skalierung-Befehls eine Umrechnung der Koordinaten von *Nassi* nach Tgif notwendig. Die Gesamtgröße des Diagramm, die bei der Umrechnung benötigt wird, ist im obersten Knoten des NXS-Baumes enthalten.

Die in Tgif zur Verfügung stehenden Schriftarten entsprechen denen von PostScript. Bei Anzeige am Bildschirm benutzt Tgif aber Ersatz-Fonts von X11, so daß Unterschiede zwischen der Bildschirm-Version und der über Tgif gedruckten Version eines Diagramm zu erwarten sind.

In den neueren Versionen von Tgif werden auch obj-Dateien mit mehreren Graphiken unterstützt. Für *Nassi* wird aber darauf verzichtet und jedes Diagramm in eine eigene Datei geschrieben.

Xfig

Im Gegensatz zu Tgif ist bei Xfig das Speicherformat *Fig* genau beschrieben. Zum Beispiel wird eine Linie im Fig-Format wie folgt beschrieben:

```
3 2 0 1 0 7 0 0 -1 0.000 0 0 0 2
         3285 3240 9720 3240
         0.000 0.000
```

Die Bedeutung der einzelnen Zahlen ist genau festgelegt. Entsprechend Tgif existieren für die Graphikobjekte von Nassi auch die Kodierungen beim Fig-Format.

Die in Xfig zur Verfügung stehenden Schriftarten entsprechen denen von PostScript. Es kann also

eine direkte Zuordnung erfolgen. Xfig benötigt für die Darstellung dieser Texte auch wieder Ersatz-Fonts von X11.

Xfig benutzt bei der Beschreibung der Koordinaten die Einheit $1/1200$ Inch. Als Ursprungspunkt wird die linke obere Ecke definiert. Also ist eine Umrechnung der Koordinaten von *Nassi* nach Xfig notwendig.

3.7.3 Durchlauf der NXS

Zum Zeichnen des Diagramms muß der NXS-Baum nur rekursiv durchlaufen werden. In jedem Knoten werden die darin enthaltenen Graphikobjekte (`graph`-Struktur) bearbeitet. Da jeder Knoten einen anderen Bereich des Diagramms beschreibt, ist die Abarbeitungreihenfolge hier nicht maßgebend. Für jedes Graphikobjekt wird das entsprechende Stück Code in der Ausgabedatei generiert oder die entsprechende API-Funktion aufgerufen.

Einziges Problem bei der Generierung der Ausgabedateien sind die Schriftarten. Zum Beispiel müssen beim Ausgabeformat PostScript die benötigten Schriftarten im Preamble der PostScript-Datei mit Umlauten versehen und definiert werden. Dazu ist ein zweiter Durchlauf durch den NXS-Baum notwendig, der vorher durchgeführt wird und nur eine Liste der benötigten Fonts aufbaut.

Reihenfolge beim Zeichnen

Wie schon beschrieben ist die Abarbeitungreihenfolge der NXS-Knoten beliebig. Dies gilt aber nicht für die Liste der Graphikobjekte eines Knotens. Ist zum Beispiel die Hintergrundfarbe eines Statements gesetzt, würde die Box des Statements bei falscher Reihenfolge des Definition den Text des Statements überdecken. Bei allen Ausgabeformaten übermalt das später gezeichnete, bzw. das an einer späteren Stelle in der Datei geschriebene Objekt das vorhergehende. Daher müssen die Objekte der einzelnen Knoten in folgender Reihenfolge bearbeitet werden:

1) Box 2) Line 3) Polyline 4) Text

Die unsortierte Liste der Graphikobjekte muß also viermal durchlaufen und nach den entsprechenden Objekten durchsucht werden.

3.7.4 Seitenumbruch

Nur bei der Ausgabe im PostScript-Format spielt die Länge des Diagramms eine wichtige Rolle. Bei allen anderen Formaten wird das Diagramm in der von Nassi gefundenen Länge abgespeichert. Die Skalierung und Bearbeitung ist dann Sache des weiterbearbeitenden Programms (Textverarbeitung oder Graphikeditor). Bei der Ausgabe auf Druckern muß das Problem überlanger Diagramme angegangen werden.

Die beste Lösung ist, das der Benutzer die Diagramme schon beim Betrachten mit Exclude-Blöcken in so kleine Teildiagramme aufteilt, so daß jedes auf eine Ausgabeseite paßt. Von dieser Lösung kann aber im Allgemeinen nicht ausgegangen werden.

Als automatische einsetzbare Lösung bleiben dann noch die Verkleinerung der Diagramme und das automatische Auftrennen der Diagramme.

Die erste Möglichkeit ist in PostScript einfach, da dort entsprechende Skalierungsbefehle am Anfang abgesetzt werden können. Die Methode wird in den meisten Fällen, besonders bei sehr langen Diagrammen, zur Unleserlichkeit führen.

Die zweite Möglichkeit beläßt die Diagramme in ihrer Größe, teilt diese nur auf mehrere Seiten auf. Die Auftrennung muß hart geschehen, also durch Abschneiden. Eine Auftrennung, die sich an die logische Struktur der Diagramme hält, ist nicht möglich, da Elemente über mehrere Seiten verteilt sein können (z.B. Schleifen). Auch hier bietet PostScript mit den Clip-Linienzügen eine Möglichkeit an. Diese Linienzügen beschreiben ein Gebiet, deren innenliegender Anteil gezeichnet und deren außenliegender Teil nicht gezeichnet wird. Bei Bearbeitung einer Ausgabeseite können also alle Graphikobjekte, von denen mindestens ein Eckpunkt auf der Seite liegt, gezeichnet werden. Der Clip-Linienzug sorgt dann dafür, daß überragende Anteile der Graphikobjekte abgeschnitten werden.

3.8 Modul Graphik-Editor

Das Modul Graphik-Editor ist für die Verwaltung des Zeichenbereich verantwortlich, der innerhalb der GUI liegt. Wie in Abbildung 3.4 auf Seite 19 dargestellt, ist dieses Bereich immer zu sehen und soll, sobald eine Routine aus der nebenstehenden Liste ausgewählt wurde, diese darstellen. Der Graphik-Editor zeigt also immer nur ein Diagramm an. Wird eine neue Routine angewählt, kann der Zeichenbereich gelöscht und das neue Diagramm darin dargestellt werden.

Neben der Darstellung der Diagramme soll der Graphik-Editor auch Möglichkeiten bieten, lokale Modifikationen an einzelnen Strukturen des Diagramms vorzunehmen. Dazu zählen neben der Änderung der Schriftarten und Layoutanpassungen auch das Auslagern von Blöcken. Dazu müssen Menüs angeboten werden, die die aktuelle Einstellung für diesen Knoten anzeigen.

Der Benutzer kann den Graphik-Editor mit der Maus und Tasten steuern. Die Programmkontrolle ist damit wie in der GUI event-orientiert. Nur wenn der Benutzer einen Event (z.B. Mausbewegung, Tastendruck) initiiert, erhält eine Callback-Routine des Graphik-Editors an die Programmkontrolle. Daher müssen wesentliche Status-Informationen zwischengespeichert werden.

Um lokale Änderungen an Strukturen durchführen zu können, müssen diese selektiert werden können. Dazu muß es zum einem eine eindeutige Abbildung der Mausposition auf einen Knoten der NXS geben und zum anderen müssen die Strukturen als selektiert markiert werden können. Über verschiedene Maustasten soll auch die Selektion mehrerer Strukturen möglich sein.

Die wesentliche Eigenschaft des Graphik-Editors ist also die Interaktion mit dem Benutzer. Die Lösung dieses Problems ist schwierig, da alle Möglichkeiten der Bedienung – korrekte und auch falsche Bedienung – abgefangen und behandelt werden müssen. Eine Oberfläche darf nie „tot“ auf dem Bildschirm stehen.

3.8.1 Schnittstelle zu den anderen Modulen

Als Eingabe dient dem Graphik-Editor wiederum der NXS-Baum, in dessen Knoten die vom Generator berechneten Graphikobjekte gespeichert sind. Neben den Graphikobjekten werden auch die Informationen aus den Hints der einzelnen Knoten benötigt.

Der Graphik-Editor besitzt keine eigenen globalen Optionen, die von der GUI verwaltet werden.

Nur im Graphik-Editor können die Hints der einzelnen Knoten verändert werden. Vom Parsers werden diese als eine Liste in jedem Knoten geliefert. Gleichzeitig stehen die Hints aber auch als NSC-Kommentar in Eingabeprogramm. Werden die Hints nun vom Graphik-Editor verändert, besteht eine Inkonsistenz zwischen der Hints-Liste und den in den `Ptext`-Listen gespeicherten Programmcode. Das Modul Parser bietet für die Angleichung beider Listen eine Routine `hints_to_ptext()` an, die nach jeder Änderung der Hints aufgerufen werden muß. Es ist sinnvoll diese Routine im Modul Parser zu belassen, da die Ptext-Liste vom Parser generiert und verwaltet wird.

Die Schnittstelle zur GUI ist vielfältig. Der Graphik-Editor muß Routinen anbieten, die als Callback-Routinen von der GUI an Tasten oder an die Maus angebunden werden. Dieses dürfen aber nur dann aufgerufen werden, wenn sich die Maus innerhalb des Zeichenbereichs befindet. Außerhalb des Bereiches können die Tasten eine andere Bedeutung haben und müssen von der GUI abgefangen werden.

Auch für die Definition der lokalen Menüs muß es ein ähnliches Verfahren wie des der GOS geben. Mit dieser Struktur werden die lokalen Menüs definiert und bei Bedarf dargestellt. Die einzelnen Optionenwerte werden in Hints-Liste zurückgeliefert. Die Oberfläche bietet zur Darstellung eine Routine `show_popup()` zur Verfügung. Der Graphik-Editor bietet eine Routine `apply_opt_draw()` an, die als Callback-Routine bei den Menüs angebunden wird.

3.8.2 Darstellung der Diagramme

X11 bietet eine Bitmap an, die als Objekt beim Aufbau der Oberfläche genutzt werden kann. Auf diese als *Canvas* bezeichnete Zeichenfläche kann mit einer Reihe von X11-Graphikbefehle gezeichnet werden. So existieren Befehle für das Zeichnen von Linien, Polygonzügen und gefüllten Flächen, das Setzen von Farben und das Schreiben von Texten.

Das Koordinatensystem von X11 richtet sich nach der Pixelauflösung des Bildschirms und die Einheiten sind ganzzahlig. Der Ursprungspunkt des Koordinatensystems liegt links oben. Bei der Umrechnung der Koordinaten muß bei einer üblichen Bildschirmauflösung von 1024x768 eine Skalierung und eine Rundung der Koordinaten vorgenommen werden. Gerade die Rundung kann zu Darstellungsproblemen führen und muß daher erst zum letztmöglichen Zeitpunkt bei der Koordinatenberechnung durchgeführt werden.

Auch für die Darstellung der Texte muß, ähnlich wie bei Tgif und Xfig, eine Näherungslösung genutzt werden. Die für die Berechnung der Diagramme genutzten PostScript-Fonts sind in X11 nicht vorhanden. In PostScript sind die Fonts auch frei skalierbar und daher auch alle Größenstufen vorhanden. Unter X11 hängt die Verfügbarkeit der Fonts vom X-Server ab, der die Fonts verwaltet.

Der X-Server ist die Komponente des X11-Systems, der den Bildschirm verwaltet. Der X-Client ist die Software, die X11 als Oberfläche nutzt. X11-Befehle, die im Client-Teil aufgerufen werden, werden über das X11-Protokoll an den X-Server weitergeleitet, der dann die Darstellung vornimmt. Diese Auftrennung der X11-Anwendungen zeigt, daß die vorhandenen Fonts je nach Bildschirm (Unix-Workstation, X-Terminal oder PC mit X-Emulation) verschieden und durch X-Client nicht veränderbar ist. Die Darstellung von Texten wird also von Bildschirm zu Bildschirm unterschiedlich sein. Ein aufwendige und für *Nassi* nicht sinnvolle Lösung wäre, die Fonts auf der Client-Seite bereit zu halten und als Bitmap an den X-Server weiterzugeben[2]. Den Ansatz, die Diagramme direkt mit den vorhandenen X11-Fonts zu generieren führt zu unterschiedlichen Diagrammen, je nach dem an welchem Bildschirm der Benutzer arbeitet[3].

Als Näherung bietet sich an, bei der Generierung der Diagramme weiter mit den PostScript-Fonts zu arbeiten und bei der Darstellung am Bildschirm auf ähnliche Fonts zurückzugreifen. Liegt ein Font nicht in der richtigen Größe vor, sollte auf den nächst kleineren ausgewichen werden. Am Bildschirm wird dann zwar eine leicht zerstückelter Textzeile entstehen, die Anfangspositionen der einzelnen Wörter sind aber korrekt und es gibt keine Überlappungen.

Das Zeichnen der Diagramme entspricht bis auf den Aufruf anderen API-Funktionen dem Zeichnen bei der Ausgabe. Der Art des Durchlaufs durch den NXS-Baumes ist beliebig. Die Reihenfolge bei Abarbeitung der Graphikobjekte eines NXS-Knotens muß aber in der auf Seite 45 beschriebenen Reihenfolge geschehen.

Auch beim Graphik-Editor stellt sich das Problem der langen Diagramme. Üblicherweise ist die Größe des Zeichenbereich sehr klein, so daß in den meisten Fällen das Diagramm nicht komplett dargestellt werden kann. Unter X11 wird daher oft eine mit Scrollbars versehene Zeichenfläche benutzt, die nur einen Ausschnitt der kompletten Zeichenfläche anzeigen. Dieser Ausschnitt kann mit Hilfe der Scrollbars über das Diagramm verschoben werden.

Unter X11 nutzt man dazu zwei X11-Bitmaps, die beide im X11-Server abgelegt werden. Die eine Bitmap (`all_bmp`) ist so groß, daß sie das ganze Diagramm aufnehmen kann. Die zweite Bitmap (`view_bmp`) entspricht dem zur Verfügung stehenden Bereich in der Oberfläche. Das Diagramm wird am Anfang einmal generiert und in `all_bmp` gespeichert. Bei jeder Scrollbar-Änderung wird nun das durch die Scrollbars beschriebene Stück in die Bitmap (`view_bmp`) kopiert. Da beide Bitmaps auf dem X-Server liegen ist diese Operation sehr schnell (ein `memcpy()`−`Aufruf`). Nach-

[2] Zum Beispiel wird dieses Prinzip vom TEX-Previewer `xdvi` angewendet. Diesem stehen die Fonts aber schon durch die Textverarbeitung zur Verfügung.

[3] Diese Methode wird zum Beispiel von Tgif angewendet, wodurch Bilder immer am gleichen Bildschirm bearbeitet werden sollen.

teil dieser Methode ist, daß die Bitmap (`all_bmp`) sehr groß werden kann und den teilweise recht kleinen Speicher des X11-Servers (z.B. bei X-Terminals) sprengen kann. Alternativ zu der Speicherung der Bitmaps auf dem X-Server blieben nur die Speicherung auf der Client-Seite, die aber hohe Übertragungsraten zur Folgen hätte. Bei jeder Bewegung des Scrollbars müßte die Bitmap `view_bmp` zum X-Server übertragen werden. Eine zweite Alternative ist jeweilige Neuzeichnen des Diagramms auf `view_bmp` (im X-Server). Neben Clipping-Problemen beim Abschneiden der einzelnen Graphikobjekte würde jeweils viele Operationen auf der Seite des X-Servers stattfinden, die z.B. bei der Anzeige von Texten relativ teuer sind. Beide Alternativen würden nicht zu einem ruckelfreien Scrolling des Diagramms führen.

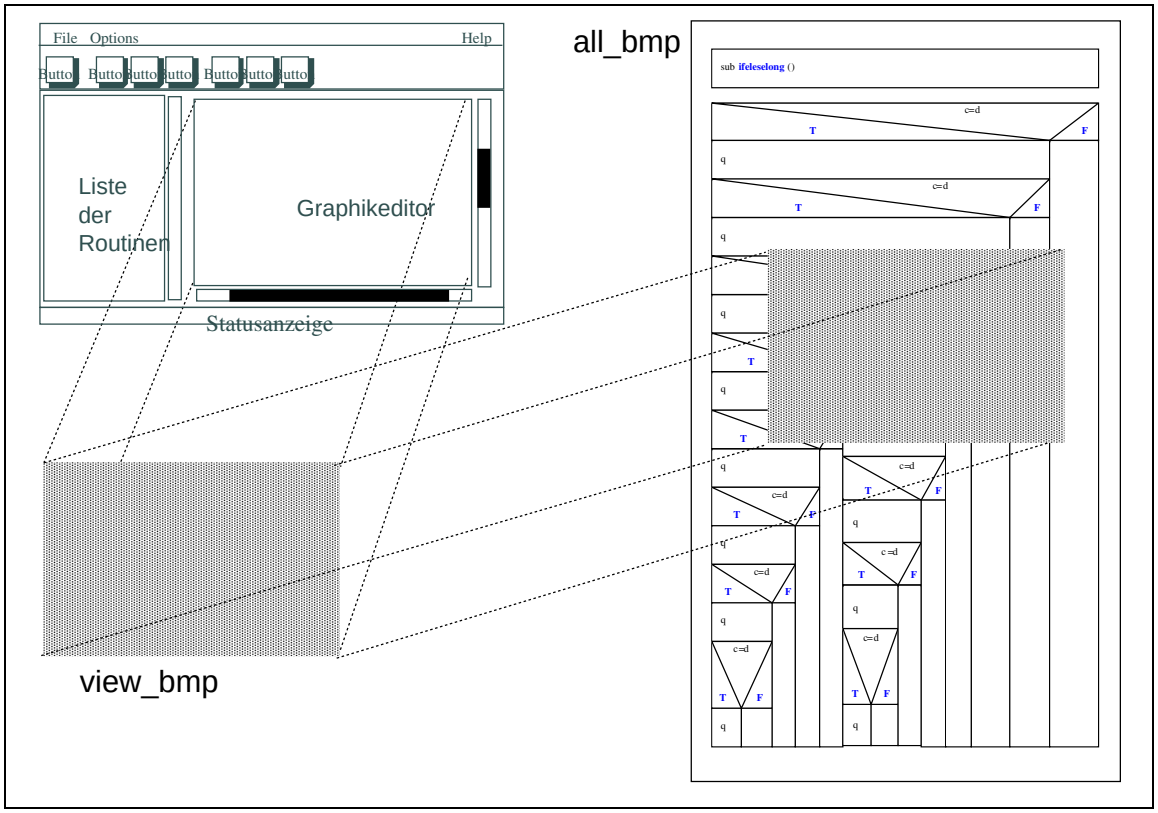


Abbildung 3.19: Darstellung der Diagramme mit X11-Bitmaps im X-Server: Sowohl die kleinere Bitmap `view_bmp`, die den sichtbaren Bereich des Diagramms enthält, als auch die große Bitmap, die das komplette Diagramm speichert, liegen auf dem X-Server. Bei jeder Scrollbar-Bewegung wird ein neuer Bereich von der großen zur kleinen Bitmap kopiert (in der Abb. von rechts nach links).

Der Benutzer soll über Tasten und andere Schaltfelder auch die Skalierung in verschiedenen Vergrößerungsstufen steuern können. Jede Änderung muß dabei durch ein Neuzeichnen des Diagramms quittiert werden. Die große Bitmap `all_bmp` muß dabei gelöscht und in der neuen Größe angelegt werden. Da im Gegensatz zu PostScript unter X11 die Skalierung nicht über einen Befehl global eingestellt werden kann muß der Skalierungfaktor beim Zeichnen des Diagramm mitgeführt werden. Jede Koordinate und Länge muß mit diesem Faktor multipliziert werden. Bei den Texten müssen auch entsprechend größere oder kleiner Fonts gesucht werden. Gerade hier ist zu erwarten, daß bei hoher Vergrößerung die Texte nicht mehr dargestellt werden können. Typischerweise liegen die Fonts im X-Server nur bis zur Größe 72pt vor.

3.8.3 Interaktion mit dem Benutzer

Die Interaktion mit dem Benutzer geschieht über Tasten, die Maus oder über Pulldown-Menüs, die jeweils Panels für die Veränderung von Strukturen aktivieren. Unter X11 lösen die Tasten und Maus ein Event aus, der vom Graphik-Editor abgefangen, ausgewertet und bearbeitet werden muß.

Belegung der Tasten und der Maustasten

Folgende Tasten sollen im Graphik-Editor belegt sein:

Taste	Belegung
PageUp	Diagramm weiter nach unten schieben
PageDown	Diagramm weiter nach oben schieben
Cursortasten	Sichtfenster in die entsprechende Richtung schieben
linke Maustaste (M1)	Struktur selektieren/deselektieren
Shift M1	weitere Struktur selektieren oder vorher selektierte Struktur deselektieren
mittlere Maustaste (M2)	verschiebt Diagramm, Taste muß dabei gedrückt bleiben
rechte Maustaste (M3)	öffnet Pulldown-Menü für Layout-Änderungen, nur wenn Strukturen selektiert sind
Control-M1	reduziert den Skalierungfaktor um eine Stufe
Control-M3	erhöht den Skalierungfaktor um eine Stufe

Tabelle 3.5: Tastenbelegung des graphischen Editor

Der Skalierungfaktor sollte zusätzlich über Schaltflächen veränderbar sein. Dabei soll die Veränderung nur schrittweise möglich sein. Mit der mittleren Maustaste kann das Diagramm „angefaßt“ und mit gedrückter Taste bewegt werden.

Events

Viele Benutzer-Aktionen können unter X11 an Routinen angebunden werden. Zum Beispiel kann unter X11 der Event `KeyPress` an eine Callback-Routine angebunden werden. Innerhalb dieser Routine muß dann der Event genauer untersucht werden. Dazu steht unter X11 eine Funktion zur Verfügung, die den Symbolnamen der zum Event gehörenden Taste zurückgibt. Diese Routine muß in der Callback-Routine als erstes aufgerufen und anhand des Symbols die dazugehörige Aktion ausgeführt werden. In der gleichen Weise gibt es einen Event `KeyRelease`, der beim Loslassen einer Taste ausgelöst wird. Die Events `ButtonPress` und `ButtonRelease` werden beim Drücken oder Loslassen einer Maustaste angestoßen.

Eine Besonderheit ist, daß das Drücken der Taste Shift einen einzelnen Event erzeugt. Für die Abfrage, ob die Tastenkombination Shift-M1 gedrückt wurde, muß also die Callback-Routine zum Event `ButtonPress` wissen, ob vorher die Shift-Taste gedrückt, aber noch nicht losgelassen wurde.

Um solche oder ähnliche Fälle abzufangen, müssen statische Status-Variablen vorhanden sein, die z.B. speichern, ob die Shift-Taste gedrückt ist. Die Callback-Routine zum Event `KeyPress` würde, falls es sich um die Shift-Taste handelt, die Status-Variable `shift_pressed` setzen, die Callback-Routine zum Event `KeyRelease` die Variable wieder zurücksetzen. Wenn der Benutzer nach dem Drücken der Shift-Taste die linke Maustaste drückt, wird die Callback-Routine zum Event `ButtonPress` aufgerufen, die nun anhand der Status-Variablen `shift_pressed` entscheiden kann, ob Shift-M1 oder nur M1 gedrückt worden ist.

Um eine Verschiebung des Mauszeigers festzustellen, steht der Event `MotionNotify` zur Verfügung, der die zugeordnete Callback-Routine bei jeder Mausbewegung innerhalb des Fensters aufruft. Diese Callback-Routine wird sehr oft aufgerufen, da eine Bewegung des Mauszeigers um ein Pixel schon diesen Event auslöst. Wenn dieser Event benutzt wird, muß er sehr effizient implementiert werden und nur die nötigsten Aktionen durchführen. Rechenintensive Aktionen würden zu einem „Ruckeln“ in der Bewegung führen.

Anhand der Funktion *Bewegen des Diagramms mit M2* soll das Verfahren nochmals genauer erläutert werden.

- Beim Drücken der mittleren Maustaste M2 wird die Callback-Routine zum Event `Button`

`Press` aufgerufen. Diese setzt nur die Status-Variable `button2_pressed`. In zwei weiteren Status-Variablen `last_x` und `last_y` speichert die Callback-Routine die aktuelle Position des Mauszeigers.

- Bei der nächsten Bewegung der Maus wird die Callback-Routine zum Event `MotionNotify` aufgerufen. Diese erkennt anhand der Status-Variablen `button2_pressed`, daß die mittlere Maustaste gedrückt ist. Das Diagramm wird dann innerhalb des Sichtfensters um die Differenz der neuen Mausposition und der in `last_x` und `last_y` gespeicherten Position verschoben. Die beiden Variablen werden danach auf die neue Position gesetzt.
  Solange der Mauszeiger bewegt wird, wird der Event `MotionNotify` immer wieder ausgelöst und dieser Schritt wiederholt.
- Wird die mittlere Maustaste losgelassen, wird die Callback-Routine zum Event `Button Release` aufgerufen. Diese setzt die Status-Variable `button2_pressed` zurück.
  Eine erneute Bewegung des Mauszeigers führt dann nicht mehr zur Verschiebung des Diagramms.

Zur besseren Übersichtlichkeit können die Callback-Routinen zu den `Press`- und `Release`-Events in einer Routine zusammengefaßt werden. Die Art des Events kann nachträglich innerhalb der Callback-Routine abgefragt werden.

Die Callback-Routine zur den Events `KeyPress` und `KeyRelease` müssen die in der folgenden Tabelle aufgeführten Aktionen durchführen.

Taste	**Event**	**Aktion**
Shift	KeyPress	`shift_pressed` auf TRUE setzen
Shift	KeyRelease	`shift_pressed` auf FALSE setzen
Control	KeyPress	`control_pressed` auf TRUE setzen
Control	KeyRelease	`control_pressed` auf FALSE setzen
PageUp	KeyPress	Diagramm um die Hälfte der Fensterhöhe nach unten schieben
PageDown	KeyPress	Diagramm um die Hälfte der Fensterhöhe nach oben schieben
CursorTaste	KeyPress	Diagramm um einen festen Betrag verschieben

Tabelle 3.6: Tasten und Aktionen zum `Key`-Event

Falls eine Taste nur mit einer einmalige Aktion belegt ist, ist es egal, ob die Aktion beim Drücken oder beim Loslassen der Taste ausgeführt wird. In `Nassi` wird die Aktion beim Drücken der Taste ausgeführt. Die Tasten zur Verschiebung (PageUp, PageDown und Cursortasten) bewegen eigentlich das Sichtfenster über das Diagramm. Das Diagramm selbst muß also in die entgegengesetzte Richtung geschoben werden.

Bei den Maustasten werden die in Tabelle 3.7 beschriebene Aktionen ausgeführt. Einige der Aktionen erklären sich erst in den nächsten Abschnitten, sind aber zur Vollständigkeit auch aufgeführt.

Die Callback-Routine zum Event `MotionNotify` muß einige Aufgaben erledigen. Nicht nur

Taste	**Art**	**Aktion**
M1	Press	Falls Mauszeiger im If-Kopf und Mauszeiger auf Mittelmarker: Position merken, Dreieck invertieren und Status-Variable `if_snapped` setzen; ansonsten Struktur selektieren/deselektieren
M1	Release	Falls `if_snapped`: `if_snapped` zurücksetzen und Dreieck neu zeichnen
Shift-M1	Press	zusätzliche Struktur selektieren/deselektieren
Control-M1	Press	Skalierung eine Stufe niedriger setzen
M2	Press	Position merken und Status-Variable `button2_press` setzen
M2	Release	Status-Variable `button2_press` zurücksetzen und Diagramm um die Differenz des jetzigen und der gemerkten verschieben
M3	Press	lokale Optionen-Menü aufrufen; die Kontrolle der Menüs unterliegt dem Modul Oberfläche
Control-M3	Press	Skalierung eine Stufe höher setzen

Tabelle 3.7: Tasten und Aktionen zum `Button`-Event

wenn eine der Status-Variablen `button2_press` oder `if_snapped` gesetzt ist, wird eine Aktion durchgeführt. Auch für die die Darstellung eines Fokus muß bei jeder Bewegung eine Aktion durchgeführt werden.

Markierung und Selektion von Strukturen

Um an einer oder mehreren Strukturen des Diagramms über die lokalen Optionen-Menüs eine Änderung vornehmen zu können, müssen diese vom Benutzer selektiert werden. Für die Selektion der Strukturen muß auch erkennbar sein, welche Struktur bei einem Tastendruck selektiert würde. Dazu muß die entsprechende Struktur unter dem Mauszeiger markiert werden (Fokus). Es gibt also zwei Arten von Markierung: Fokus und Selektion. Der Fokus soll dem Benutzer nur einem Hinweis geben und kommt auf dem Bildschirm nur einmal vor. Die Markierung zur Selektion kann in Prinzip bei jeder Struktur des Diagramms vorliegen.

Beide Arten markieren immer dasjenige Rechteck, das die Struktur vollständig umschließt. Bei der Markierung einer While-Schleife liegt zum Beispiel der Schleifenkörper in diesem Rechteck. Für die Selektion werden die Ecken des Rechtecks mit kleinen roten Markern versehen. Für den Fokus sollen mehrere Arten zur Verfügung stehen. Je nach Hintergrundfarbe und Linienstärke sind diese mehr oder weniger gut sichtbar. Als Default wird ein Rahmen um das Rechteck benutzt. Damit der Rahmen auch dann bei dunklem Hintergrund zu sehen ist, soll die Linien nicht mit einer Farbe ausgefüllt, sondern der Bereich der Linien invertiert werden. Damit ist sichergestellt, daß der Rahmen immer zu sehen ist. Als Alternativen zum Rahmen sind noch Linien außerhalb des Diagramm denkbar, die die Lage des Fokus im Diagramm andeuten. Weiter ist die Invertierung der gesammten Struktur möglich.

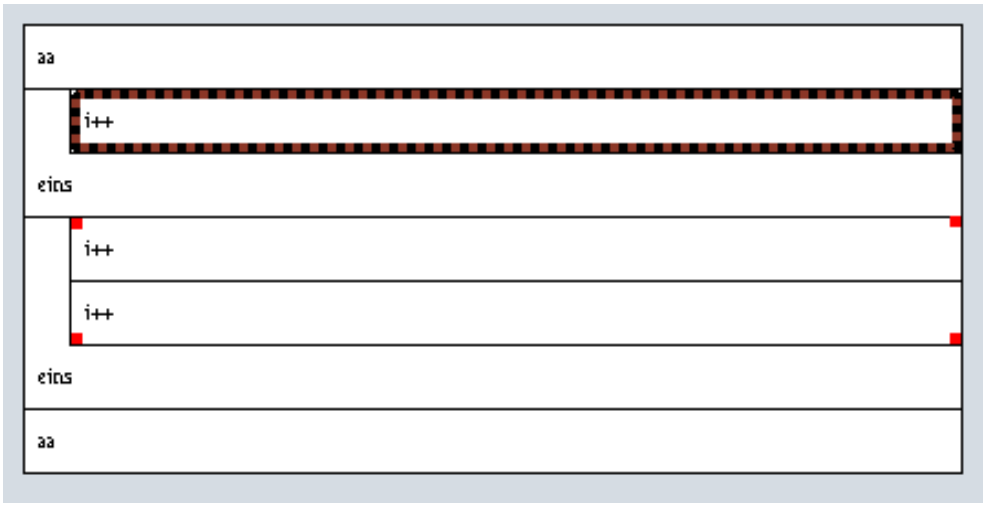


Abbildung 3.20: Selektieren von Strukturen: Die obere Umrahmung des ersten Schleifenkörpers stellt den Fokus dar. Dieser soll mit der Maus mitwandern und immer die aktuelle Struktur markieren. Der Schleifenkörper der unteren Schleife ist markiert. Änderungen in den lokalen Optionen-Menüs beziehen sich auf diesen Schleifenkörper. Im *Nassi* sollen die Markierungspunkte an den Eckpunkten in rot dargestellt werden.

Der Fokus soll immer die Struktur selektieren, die sich unter dem Mauszeiger befindet. Dazu muß der Graphik-Editor, bzw. die Callback-Routine zum Event `MotionNotify` bei jeder Bewegung den NXS-Baum durchlaufen und denjenigen Knoten suchen, dessen umschließendes Rechteck die Position des Mauszeigers enthält. Der Baum muß solange nach unten hin verfolgt werden, bis daß ein Knoten gefunden wurde, der entweder keine Söhne mehr hat oder dessen Sohnknoten die Position des Mauszeigers nicht mehr beinhalten (bei If-Statements möglich). Es wird also immer das kleinstmögliche Rechteck zum Fokus. Gleichzeitig sollen aber alle Strukturen des Diagramms selektierbar sein. Zum Beispiel stellt der Schleifenkörper auch eine eigene Struktur, eine Block dar.

Da der Block im Diagramm aber keine eigene Repräsentanz hat, würde dieser bei dem oben beschriebenen Verfahren nie zum Fokus werden. Um dies zu verhindern muß beim Durchlauf durch den Baum bei einem Block ein besonderes Verfahren anwenden: Der Block selbst wird selektiert, wenn der Mauszeiger sich in einem geringen Abstand zum Rand des Blockes befindet, der Baum wird nicht mehr weiter durchlaufen. Ist er weiter entfernt, wird der Baum weiter durchlaufen und ein im Block enthaltene Struktur selektiert. Für den Benutzer stellt sich das Verhalten des Fokus dann so dar, daß, wenn er von außen zu einem in einem Block enthaltenen Statements fährt, zuerst der Block und dann das Statement den Fokus erhält.

Ändert sich der Fokus, muß die neue Struktur mit dem Fokus versehen und der alte Fokus gelöscht werden. War der alte Fokus durch Invertierung entstanden, kann zum Löschen diese Invertierung wieder vorgenommen werden.

Alternativ dazu kann der Fokus nur auf der der kleinen Bitmap (`view_bmp`) gezeichnet werden. Beim Löschen kann der Bereich durch Kopieren aus der Bitmap `all_bmp` wiederhergestellt werden. Wird das Diagramm im Fenster verschoben, ändert sich zwangsläufig auch der Fokus und ein neuer Fokus wird in der kleinen Bitmap gezeichnet. Der alte Fokus wurde durch das Kopieren der Bitmap überschrieben und muß daher nicht mehr gelöscht werden.

Im Gegensatz zum Fokus, muß die Markierung der Selektion auch in den zur Zeit nicht sichbaren Bereichen markiert werden, da diese nach dem Verschieben des Diagramms auch noch sichtbar sein sollte. Hier muß auf jeden Fall in die Bitmap `all_bmp` gezeichnet werden. Da durch die roten Markierungspunkte Teile der Struktur überschrieben werden können, muß bei Aufhebung der Selektion die Struktur neu gezeichnet werden. Dies ist erheblich aufwendiger als das Verfahren beim Fokus, kommt dafür aber seltener vor.

Optimierung des Fokus

Aktionen, die bei der Bewegung der Maus ausgeführt werden, sollten sehr effizient implementiert werden. Nach der oben beschriebenen Methode wird bei jeder Bewegung ein Teil des NXS-Baumes durchlaufen. Dieser hoher Aufwand würde sicherlich zu einer ruckelnden Bewegung des Mauszeigers führen. Eine einfache Optimierung ist die, den Außenbereich des Diagramm von diesem Verfahren auszuschließen. Solange sich der Mauszeiger außerhalb des vom Diagramm im Sichtfenster belegten Platz befindet, kann die Callback-Routine zum Event `MotionNotify` wieder direkt verlassen werden. Die Eckpunkte des Diagramms sollten auch in statischen Variablen abgespeichert werden, um eine dauernden Zugriff auf die NXS-Struktur zu verhindern[4].

Diese Optimierung hilft aber nur, wenn der Mauszeiger sich nicht über dem Diagramm befindet. Für diesen Fall steht eine zweite Optimierung bereit, die dafür sorgt, daß nach einer Suche im NXS-Baum der Bereich der gefundenen Struktur bestimmt wird (*trust_area*), für den keine neue Suche nötig ist. Aus Performance-Gründen sollte es sich bei dem *trust_area* um ein Rechteck handeln. Die Koordinaten des Rechtecks und eine Referenz zum entsprechenden NXS-Knoten werden in statischen Variablen gehalten. Die Callback-Routine zum Event `MotionNotify` muß nur dann einen neuen Knoten im NXS-Baum suchen, wenn die neue Position des Mauszeigers sich nicht mehr in der *trust_area* befindet.

Für einfache Statements ist die *trust_area* der komplette vom Statement im Diagramm belegte Platz. Bei Schleifen-Strukturen hängt der Bereich von der Position des Mauszeigers innerhalb der Schleife ab. Bei der einfachen Fallunterscheidung kann der ganze Kopf als *trust_area* genommen werden. Bei einer Mehrfachauswahl muß das Rechteck auf die Breite der darunter liegenden Spalte beschränkt werden. Bei einer Block-Struktur ist der Bereich sehr klein, da ab einen gewissen Abstand direkt zu den darin enthaltenen Strukturen weitergegangen werden soll. Die *trust_area* für diesen Fall liegt an der jeweiligen Seite des Blockes. Abb. 3.21 auf der nächsten Seite zeigt einige Beispiele für die Bestimmung der *trust_area*.

[4]Die Koordinaten des Gesamtdiagramms stehen im obersten Knoten der NXS-Struktur, müssen aber noch in das Koordinaten-System des Sichtfensters (`view_bmp`) umgerechnet werden.

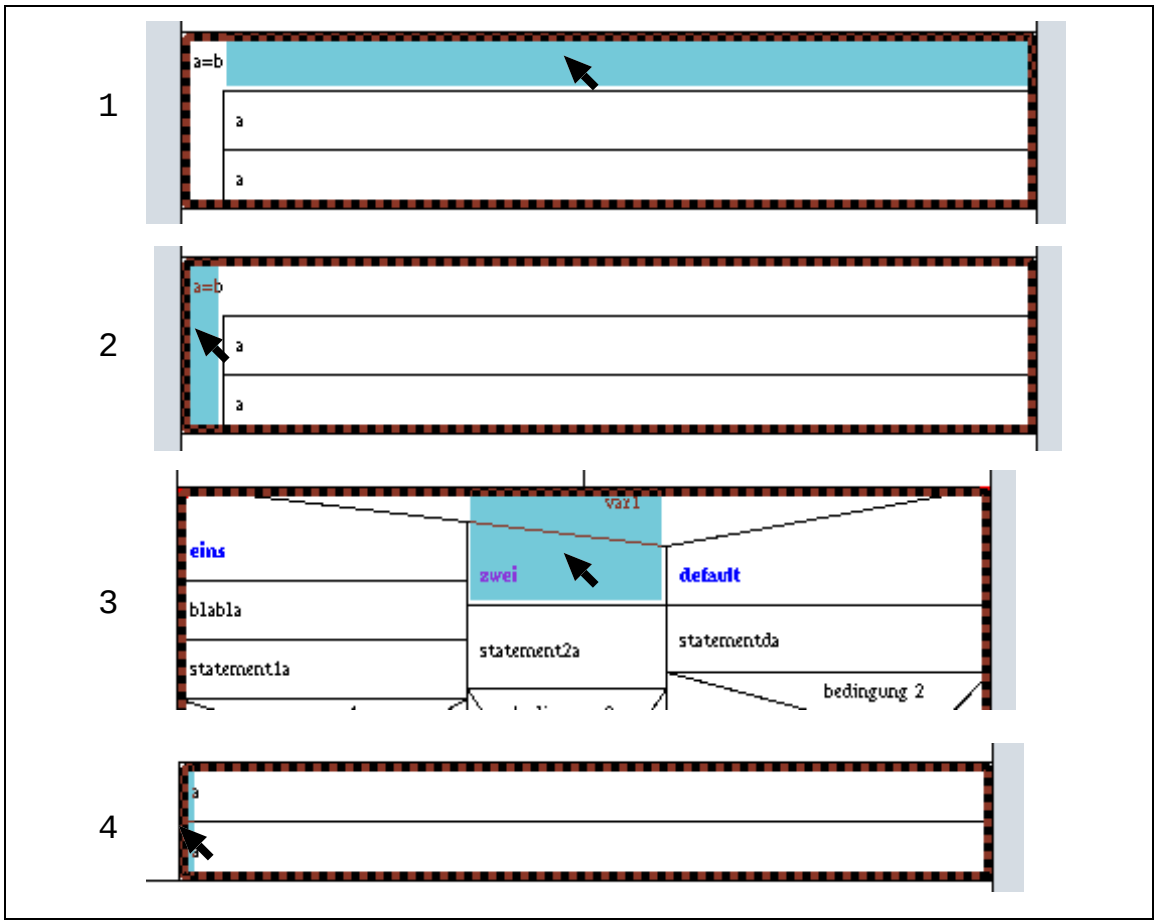


Abbildung 3.21: Optimierung des Fokus: Die vier Abbildungen zeigen die *trust_area* bei verschieden Strukturen eines Nassi-Shneiderman-Diagramms. Im Bild 1 befand sich der Mauszeiger in der rechten Hälfte der Schleifen-Struktur. Das grau hinterlegte Rechteck zeigt die *trust_area* in diesem Fall. Nur wenn dieser Bereich verlassen wird, muß eine neue Untersuchung des NXS-Baumes vorgenommen werden. Das zweite Bild zeigt die *trust_area* in dem Fall, daß sich der Mauszeigers im linken Teil der Schleifen-Struktur befand. Bild 3 zeigt die *trust_area* bei einer Switch-Struktur. Da der Kopf nach unten hin für jede Spalte einen anderen Abschluß besitzt, muß der Bereich auf die entsprechende Spalte beschränkt werden. Das letzte Bild zeigt die *trust_area* bei einer Blockstruktur, die dort sehr schmal ist.

Insgesamt bringen die beiden beschriebenen Optimierung einen erheblichen Performance-Gewinn. In den meisten Fällen kann die Callback-Routine zum Event `MotionNotify` direkt beendet werden. Eine ruckelfreie Bewegung des Mauszeigers sollte damit auch auf langsamen Rechnern möglich sein.

Verwaltung einer Selektions-Liste

Die Layoutänderungen der lokalen Optionen-Menüs können auf eine oder mehrere Strukturen des Diagramms wirken. Zur Selektion dieser Strukturen steht die rechte Maustaste zur Verfügung. Durch einfachen Anklicken wird die mit dem Fokus markierte Struktur selektiert. Durch nochmaliges Drücken wird die Selektion wieder aufgehoben. Die Tastenkombination Shift-M1 wirkt ähnlich, fügt die Struktur aber nur zu den bisher selektierten dazu, bzw. entfernt die Struktur davon.

Für die interne Verarbeitung der selektierten Strukturen muß eine Liste geführt werde, in die die Strukturen eingefügt, bzw. gelöscht werden. Die Aktionen hinter den Tasten können somit auf bekannte Listenoperationen zurückgeführt werden. Tabelle 3.8 auf der nächsten Seite faßt dies nochmal zusammen.

Die Selektionliste wird zum Beispiel von den lokalen Optionen-Menüs benötigt. Wird die Liste verändert, müssen sich die Menüs auch anpassen. Enthält die Liste nur eine Struktur, sollten die lokal gesetzten Hints in den Menüs angezeigt werden. Ist keine Struktur selektiert, sollte existierende

Taste	Struktur in der Liste	Operation
M1	nein	Liste entleeren und neuen Verweis auf Struktur in der Liste speichern
M1	ja	Liste entleeren
M1	nein/kein Fokus	Liste entleeren
Shift-M1	nein	Verweis auf Struktur an die Liste anhängen
Shift-M1	ja	Verweis auf Struktur aus der Liste löschen

Tabelle 3.8: Operationen auf die Selektionliste

Menüs gesperrt sein und keine neuen geöffnet werden können. Sind mehrere Strukturen selektiert, können keine lokalen Hints angezeigt werden. Die Selektionliste muß auch bei einigen Aktionen gelöscht werden. Zum Beispiel wenn das Diagramm gewechselt wird, oder ein Block ausgelagert wird.

PopUp-Menüs

Dadurch, daß zu jedem Knoten eine Liste von Hints verwaltet wird, ist es möglich alle Optionen, die über die globalen Menüs veränderbar sind, auch lokal für eine einzelne Struktur zu verändern. Das Modul Generator geht bei der Interpretation der Hints davon aus, daß, falls es bei einer Option kein lokaler Hint vorhanden, der Wert von dem darüberliegenden Knoten übernommen wird. Diese Vererbung ermöglicht es, auch Optionen für komplette Teilbäume zu verändern. Um z.B. die Schriftgröße für alle Statements einer While-Schleife zu verändern, genügt es diese bei der Schleife selbst zu verändern. Die veränderte Option wird an die in dem Schleifenkörper enthaltenen Strukturen weitergegeben. In den lokalem Optionen-Menüs muß für jede Einstellung auch den Wert *inherit* geben, mit dem angezeigt wird, daß der Wert lokal nicht verändert wird, der Wert also vom übergeordneten Knoten vererbt wird.

Da alle globalen Optionen auch als Hints vorkommen können und zusätzlich noch Hints zur Strukturierung der Diagramme (z.B. `exclude`) dazukommen, ist die Liste der lokalen Optionen relativ lang. Eine Aufteilung in Untermenüs ist daher unumgänglich. Das oberste Menü, das über die rechte Maustaste aufgerufen werden kann, sollte nur eine Auswahl der Untermenüs anbieten und nicht selbständig auf dem Bildschirm stehen bleiben. Die Untermenüs stellen dagegen Einstell-Panels dar, die – einmal aufgerufen – selbständig auf dem Bildschirm stehen können. Solange ein einziges Statement selektiert ist, enthält das Panel die dort lokal veränderten Optionen. Nicht veränderte Optionen werden im Panel als *inherit* vermerkt. Durch die Darstellung der Panel als eigenständige Fenster kann der Benutzer durch Anklicken weiterer Strukturen die lokalen Optionen überprüfen. Die Panel bleiben als Statusfenster neben dem Nassi-Window stehen. Die Panels benötigen neben den Buttons `Ok` (Bestätigen und Panel schließen) und `Cancel` (Panel schließen) auch eine Button `Apply`, der die neuen Werte bestätigt, das Panel aber nicht schließt.

Ist mehr als eine Struktur selektiert, können keine lokalen Optionen mehr angezeigt werden, da in verschiedenen Strukturen auch verschiedene Sätze von Hints gespeichert sein können. Alle Werte müssen in den Panels mit *inherit* belegt sein. Eine Änderung der Optionen ist aber möglich. Die veränderten Optionen werden dann in jedem selektierten Knoten als Hints gespeichert. Bei diesem Merge bleiben diejenigen Hints in den Knoten unverändert, deren Optionen in den Panel noch auf *inherit* stehen.

Bei Aufruf eines Menüs müssen folgende Schritte durchgeführt werden:

1. Werte der Optionen in der GOS-Struktur zurücksetzen (*inherit*)
2. lokal im Knoten veränderte Hints in die GOS-Struktur speichern, falls nur eine Struktur selektiert wurde,
3. Auswahl-Menü anzeigen oder bestehende Panels aktualisieren

Nach der Bestätigung der Werte im Panel (mit Apply oder Ok) sind folgende Schritte notwendig:

1. Werte, die verschiedenen von *inherit* sind, aus der GOS-Struktur auslesen, und
2. mit den lokal in den Knoten gespeicherten Hints zusammen mischen,
3. lokale Hints in der Ptext-Liste aktualisieren,
4. Diagramm aktualisieren

Nachdem die neue Werte der Optionen in die Liste der Hints gelangt ist, müssen diese auch bei der Darstellung der Diagramme berücksichtigt werden. Dazu muß der Generator ausgeführt und anschließend das Diagramm auf dem Bildschirm neu gezeichnet werden. Dieser Ablauf muß der Konsistenz wegen von einer Routine des Oberfläche durchgeführt werden. Da es auch eine Option gibt, die die Auslagerung (Exclude) einer Struktur beschreibt, muß in der Oberfläche auch die Liste der in der Eingabe enthaltenen Funktionen aktualisiert werden.

Ist keine Struktur selektiert müssen die Optionen in den Panels festgesetzt werden (Lock). Sie dürfen dann nicht geändert werden. Dazu steht in der GOS-Struktur das Flag `lock` zur Verfügung.

Die folgenden Listen geben die Optionen an, die in den einzelnen Panels angezeigt werden sollen.

Panel **Structure**:

`Exclude`	Diese und die darin enthaltenen Strukturen sollen ausgelagert, also in einem eigenen Diagramm gezeichnet werden.
`ExcludeText`	Beschreibung des ausgelagerten Blockes, der Text wird im übergeordneten Diagramm, als Titel im ausgelagerten Diagramm und in der Liste der Funktionen verwendet.

Panel **Fonts/Colors**:

`[KEY]FONT_NAME`	Fontname der normale/hervorgehobenen Schrift
`[KEY]FONT_FORMAT`	Format der normale/hervorgehobenen Schrift
`[KEY]FONT_SIZE`	Größe der normale/hervorgehobenen Schrift
`[KEY]FONT_COLOR`	Farbe der normale/hervorgehobenen Schrift
`FILL_COLOR`	Hintergrundfarbe
`LINE_COLOR`	Linienfarbe

Panel **Layout/Misc**:

`HyphenText`	Texteingabefeld, das zu Beginn den Text der selektierten Struktur enthält. Als einzige Tasten sind in diesem Feld `~`, `#`, Backspace und die Cursortasten zur Eingabe von Trennvorschlägen definiert.
`HyphenToGlobal`	Schalter, der angibt, ob die Trennvorschläge auch in den globalen Trennvorschlägen gespeichert werden sollen.
`SHOW_REPEAT`	Anzeige des Schlüsselworts REPEAT
`REPEAT_TEXT`	alternativer Text für Schlüsselwort REPEAT
`FCALL`	Markieren von Funktionaufrufen
`ALTERNATE_SWITCH`	Case-Zweige untereinander statt nebeneinander
`MIN_WIDTH`	Mindestbreite eine Spalte bei If/Case (in %)
`LANGUAGE`	Sprache für Literale (z.B. True oder Wahr beim If-Statement)
`SIDE_DISTANCE`	Abstand zwischen Text und Seitenlinien
`VERTICAL_DISTANCE_TOP`	Abstand zwischen erster Zeile und Kopflinie
`VERTICAL_DISTANCE_BOT`	Abstand zwischen letzter Zeile und Fußlinie
`COLSPEC_IF`	Aufteilung der Spalten beim If-Statement (%:%)
`HYPHEN`	Trennvorschläge für dieses Statement

Für die in der Option HyphenText veränderbaren Trennvorschläge stellt der Graphik-Editor den Text der selektierten Struktur zur Verfügung. Dazu werden die einzelnen Wörter aus der Ptext-Liste des Knotens entnommen und für den Panel-Eintrag zusammengestellt. Nach der Bestätigung der Optionen muß diese Wortliste auf Wörter untersucht werden, die Trennvorschläge enthalten. Diese müssen ggf. als Hint gespeichert werden. Sind keine vorhanden, wird auch kein lokaler Hint gespeichert. Für viele Optionen sind Slider sinnvoll, mit denen der Wert zwischen einem Minimal- und

einem Maximalwert verändert werden können. Für die Option `COLSPEC_IF` steht auch die interaktive Veränderung direkt im Diagramm zur Verfügung. Der nächste Abschnitt wird dies genauer erläutern.

Veränderung der Breite von Then- und Else-Spalten

Gerade die Einstellung der Breite der Spalten bei einem If-Statement beeinflußt die Darstellung der Diagramme sehr. Die Berechnung der Breite wird vom Generator zwar in einer aufwendigen Art mit mehrfachem Durchlauf durch den NXS-Baum vorgenommen. Trotzdem kann es vorkommen, das die Breite einer Spalte ein wenig zu klein ist und dadurch Worte schlecht getrennt werden. Die Option `COLSPEC_IF` die das Verhältnis der beiden Spalten eines If-Statements in Prozent angibt, bringt zwar Abhilfe. Die Eingabe über das Panel ist aber mühsam. Eine bessere und mit den Mitteln des Graphik-Editors einfach zu realisierende Möglichkeit ist die, die Spalte mit dem Mauszeiger zu verschieben. Dazu erhält der untere Eckpunkt des Dreiecks im If-Kopf eine Markierung, sobald das If-Statement selektiert wird (siehe Abbildung 3.22). Dieser Punkt kann mit der linken Maustaste angeklickt und verschoben werden.

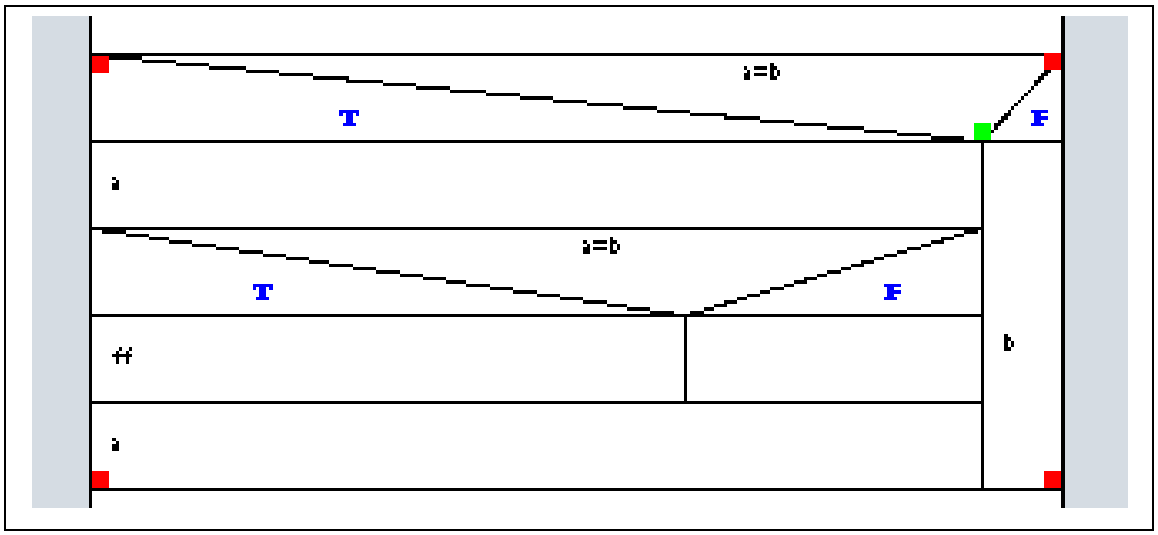


Abbildung 3.22: Marker zum Verschieben einer If_Spalte: Der untere Marker am Dreieck des If-Kopfes kann mit der linken Maustaste angeklickt und verschoben werden.

Das Verfahren ist ähnlich zum dem Verschieben des Diagramms mit der mittleren Maustaste. Bei der Selektion eines If-Statements wird ein Flag gesetzt und die die Koordinaten des neuen Markers gespeichert. Die Callback-Routine zum Event `ButtonPress` muß dann, falls dieses Flag gesetzt ist und die linke Maustaste gedrückt wurde, die aktuelle Position mit der gespeicherten des Markers vergleichen. Stimmen diese überein, wird ein weitere Flag gesetzt, das anzeigt, das der Editor sich jetzt im „if_snapped“-Modus befindet.

In diesem Bewegungsmodus müßte dann die Callback-Routine zum Event `MotionNotify` bei jeder horizontalen Bewegung des Mauszeigers das neue Verhältnis der Spalten zueinander in den Hints speichern, den Generator aufrufen und das Diagramm neu zeichnen. Gegen diesen Verfahren sprechen einige Gründe. Zum einem ist der Aufwand zu groß. Das Diagramm könnte sicherlich nicht in der gleichen Geschwindigkeit neugezeichnet werden, wie der Mauszeiger sich bewegt. Das würde aber dazu führen, daß das Diagramm hinter dem Mauszeiger hinterher „hinkt“. Ein zweiter Grund ist der, daß das Diagramm neu berechnet und durch die veränderten Diagrammgrößen das Diagramm an einer (leicht) verschobenen Stelle im Sichtfenster zu sehen ist. Der Marker würde sich nach einer ersten Verschiebung nicht mehr unter dem Mauszeiger befinden. Die Informationen zu der letzten Position könnten zwar über den Aufruf des Generators hinweg gespeichert und das Diagramm durch die Scrollbars wieder an die letzte Position gerückt werden, was aber aufwendig und wahrscheinlich zum „Flackern“ des Sichtfensters führen würde.

Daher wird während des Bewegungsmodus nur das Dreieck neu gezeichnet. Dieses Zeichnen kann direkt vom Graphik-Editor vorgenommen werden und entspricht im wesentlichen dem Zeichnen des Fokus. Bei jeder Bewegung wird das neue Dreieck durch Invertierung gezeichnet und das alte Dreieck durch Invertierung gelöscht. Zu beachten ist, daß das Dreieck nur innerhalb des If-Kopfes verändert werden darf (0-100%) und das die Mindestbreiten der Spalten berücksichtigt werden muß.

Erst wenn die linke Maustaste losgelassen wird, wird der Bewegungsmodus beendet, das neue Verhältnis der Spalten zueinander in den Hints gespeichert, der Generator aufgerufen und das Diagramm neu gezeichnet. Auch müssen die lokalen Optionen-Menüs aktualisiert werden, da dort die Option `COLSPEC_IF` zu sehen ist (siehe Abbildung 3.23).

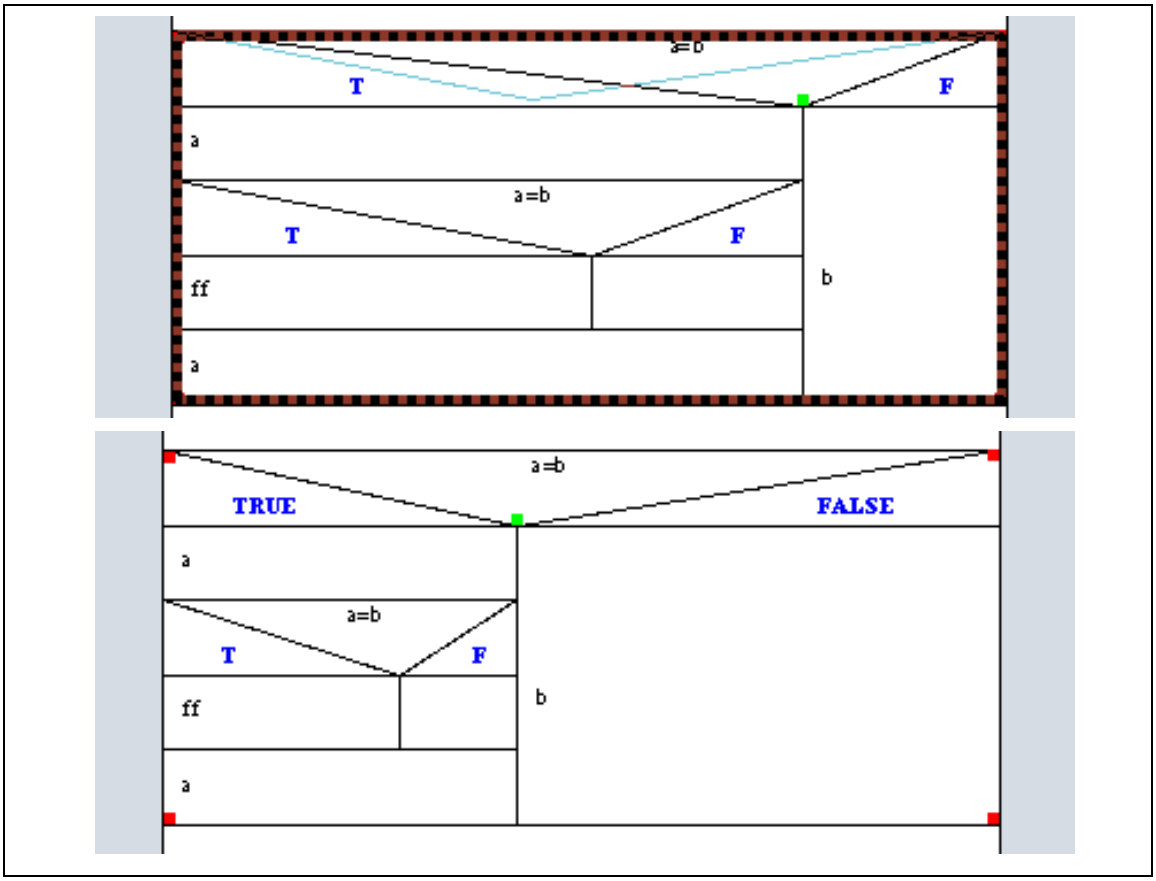


Abbildung 3.23: Verschieben einer If_Spalte: Nachdem der Marker angeklickt ist, wandert das Dreieck mit dem Mauszeiger mit. Erst beim Loslassen der linken Maustaste wird das Diagramm neu gezeichnet.

Erweiterungen

Der Graphik-Editor ist sicherlich dasjenige Modul, in das viele Erweiterungen eingebaut werden können. Zum Beispiel sollte die Verschiebung der Spalten auch bei Switch-Anweisungen möglich sein. Die in diesem Kapitel aufgezeigten Verfahren mit Status-Variablen und Event-Handling bieten dazu eine gute Basis. Eine Erweiterung zu einem richtigen Editor, mit dem Diagramme am Bildschirm erstellt werden können, war bei dieser Entwicklung nicht vorgesehen, sollte aber mit den vorhandenen Schnittstellen und Funktionen der Oberflächen möglich sein. Auch die Anbindung weitere Tasten ist sinnvoll und über die Callback-Routinen möglich.

Die Event-orientierte Behandlung und das Zeichnen der Graphikobjekte gilt sicherlich auch für andere Oberflächensysteme und kann dort genauso angewandt werden. Die Portierung auf andere Plattformen sollte also durch den Ersatz der Oberflächen-API möglich sein.

3.9 Modul Fontgenerator

Ein besonderes Problem bei der Anzeige und der Ausgabe von Nassi-Shneiderman-Diagramme stellen die Schriften dar. Wird eine nicht äquidistante Schriftart genutzt, hängt die Breite einer Zeichenkette von den einzelnen Zeichen ab. Um eine vernünftige Formatierung der Texte innerhalb eines Diagramms möglich zu machen, müssen die Dimensionen der einzelnen Zeichen *Nassi* (bzw. dem Generator) bekannt sein. Die Bereitstellung der Größen ist eine Aufgabe des Moduls `FontGen`.

Auch ist die Anzahl und die Art der Schriften von Ausgabeformat zu Ausgabeformat verschieden. Die Schriften werden im allgemeinen durch Attribute klassifiziert. Dies soll auch innerhalb von *Nassi* der Fall sein. Je nach Ausgabeformat gibt es also eine Menge von Schriften, die über ihre Attribute identifiziert werden können. Die zweite Aufgabe von `FontGen` besteht darin, die vom Benutzer angewählte Attribute einer gewünschten Schriftart auf die Attribute der im Ausgabeformat vorhandenen Schriften abzubilden.

3.9.1 Schnittstelle zu den anderen Modulen

Im wesentlichen benötigt das Modul `Generator` die Informationen über die Schriftarten zur Positionierung von Texten. Das Ausgabemodul und der Graphik-Editor benötigen nur die Attribute der Schriftart im Ausgabeformat. Die Ausformung der Schnittstelle zu diesen Modulen hängt von den Speicherung der Informationen ab und wird daher erst in den nächsten Abschnitten festgelegt. Als Schnittstelle sind zum einen Funktionen möglich, die zu jedem Attribut-Tupel und einem Zeichencode einen Wert zurückgeben und zum anderen sind auch offene Datenstrukturen – wie z.B. NXS auch eine ist – denkbar, die an die entsprechenden Stellen in der NXS angehängt werden.

Die Attribute einer gewünschten Schriftart stehen dem Modul `Generator` über die in der NXS gespeicherten Hints zur Verfügung. Die Attribute erlauben eine Auswahl der Schriftfamilie, des Schriftformats und der Schriftgröße[5].

In gewisser Weise sollten auch die Konfigurationdateien, die für die Speicherung der Zuordnungstabellen verwendet werden müssen, in diesem Abschnitt aufgeführt werden. Über Konfigurationdateien können noch bei der Installation von `Nassi` die Liste der vorhandenen Schriftarten angepaßt werden[6].

3.9.2 Schriftarten und deren Attribute in den Ausgabeformaten

In *Nassi* werden die Schriftarten über die Attribute *Familie*, *Format* und *Größe* beschrieben. In diesem Abschnitt wird nun untersucht, inwieweit diese Attribute auch in den verschiedenen Ausgabeformaten vorhanden sind.

Als Ausgabeformate werden in *Nassi PostScript* und die Speicherformate der Editoren *Tgif* und *Xfig* genutzt. Bei allen dreien basiert das Namensschema auf den PostScript-Bezeichnungen. Aber nur bei PostScript sind die Schriften frei skalierbar. Bei den anderen Formaten stehen pro Schriftfamilie nur einige wenige Größen zur Verfügung. Dies liegt an den gleichen Problemen, die auch bei dem Entwurf des Graphik-Editors von *Nassi* aufgetreten sind und mit der geringen Auswahl von Schriftarten unter X11 begründet werden können.

Um bei allen Ausgabeformaten eine gleichermaßen gute Abbildung der gewünschten Schriftart auf die vorhandene Schriftart zu ermöglichen, muß bei der Attributsauswahl für den Benutzer Einschränkungen getroffen werden. Maßgebend ist hier das Ausgabeformat mit den wenigsten Aus-

[5] Ein weitere Hint, der die Farbe der Schrift bestimmt, ist unabhängig für die Auswahl, da in alle Ausgabenformaten, die Farbe über getrennte Befehle gesetzt wird.

[6] Zum Beispiel haben die meisten Textverarbeitungsprogramme eine Library, in der die Schriftarten verwaltet werden und zusätzlich Dateien, die die Zuordnung der Schriftarten definiert.

wahlmöglichkeiten. *Tgif* stellt hier die Einschränkung dar und definiert damit die folgende Liste von vorhandenen Möglichkeiten der Schriftattribute.

Attribut	Wert	Beschreibung
Name	Times	Schriftfamilie mit Serifen, die häufig bei Fließtexten verwendet wird.
	Helvetica	Schriftfamilie ohne Serifen, die häufig bei plakativen Darstellungen verwendet wird.
	Courier	Schriftfamilie, deren Zeichen alle die gleiche Breite haben.
	NewCentury Schlbk	Alternative zu Times
Format	roman	Aufrechte Schriftart
	italic	Schrägstellung
	bold	Fettschrift
	bold-italic	Fettschrift und Schrägstellung
Größe	8,10,11,12,14,17, 18,20,24,25,34	Feste Größen, die *Tgif* aus den vorhandenen Schriftgrößen des X11-Servers festlegt. Die Angaben beziehen sich auf Punkte (pt). Zum Beispiel entspricht 11pt der in diesem Dokument verwendeten Schriftgröße.

Tabelle 3.9: Mögliche Werte für die Schriftattribute auf der Benutzerseite

Mindestens die in der Tabelle aufgeführten Werte sollten dem Benutzer über Menüs angeboten werden. Weitere Familien und Formate können zu der Liste hinzugefügt werden, müssen aber sowohl der Oberfläche als auch der dem Fontgenerator bekannt gegeben werden. Letzterer wird dazu Konfigurationdateien zur Verfügung stellen[7]. Die Bezeichnung der Schriftarten ist unter PostScript leicht verschieden zu denen in der Tabelle aufgeführten. In PostScript werden der Name und das Format zu einem Namen verbunden (z.B. `Times-Roman`).

Für die Speicherung der Zuordnung werden Konfigurationdateien vorgeschlagen, die für jedes Ausgabeformat existieren müssen, und die vorhanden Schriften, deren Größen und der Bezeichner im Ausgabeformat. Abbildung 3.24 zeigt eine solche Datei.

```
Times                 roman          Times-Roman                 8,10,11,12,14,17,18,20,24,25,34
Times                 italic         Times-Italic                8,10,11,12,14,17,18,20,24,25,34
Times                 bold           Times-Bold                  8,10,11,12,14,17,18,20,24,25,34
Times                 bold-italic    Times-BoldItalic            8,10,11,12,14,17,18,20,24,25,34
Helvetica             roman          Helvetica                   8,10,11,12,14,17,18,20,24,25,34
Helvetica             italic         Helvetica-Italic            8,10,11,12,14,17,18,20,24,25,34
Helvetica             bold           Helvetica-Bold              8,10,11,12,14,17,18,20,24,25,34
Helvetica             bold-italic    Helvetica-BoldItalic        8,10,11,12,14,17,18,20,24,25,34
Courier               roman          Courier                     8,10,11,12,14,17,18,20,24,25,34
Courier               italic         Courier-Italic              8,10,11,12,14,17,18,20,24,25,34
Courier               bold           Courier-Bold                8,10,11,12,14,17,18,20,24,25,34
Courier               bold-italic    Courier-BoldItalic
NewCenturySchlbk      roman          NewCenturySchlbk            8,10,11,12,14,17,18,20,24,25,34
NewCenturySchlbk      italic         NewCenturySchlbk-Italic     8,10,11,12,14,17,18,20,24,25,34
NewCenturySchlbk      bold           NewCenturySchlbk-Bold       8,10,11,12,14,17,18,20,24,25,34
NewCenturySchlbk      bold-italic    NewCenturySchlbk-BoldItalic 8,10,11,12,14,17,18,20,24,25,34
```

Abbildung 3.24: Mapping-Datei für die Attribut-Zuordnung von `Fontgen` beim Ausgabeformat *Tgif*. Die ersten beiden Spalten bezeichnen die vom Benutzer wählbaren Attribute Name und Format. Die dritte Spalte enthält die Fontbezeichner des Ausgabeformats und in der vierten Spalte stehen die für diesen erlaubten Größen. Fehlt diese vierte Angabe, sind alle Fontgrößen erlaubt (z.B. bei PostScript).

Steht eine Schrift im Ausgabeformat nicht mit den gewünschten Attributen zur Verfügung, müssen Ersatz-Schriften gefunden werden. Dafür werden folgende Regeln aufgestellt:

- Wird die gewünschte Schriftart (Name + Format) nicht gefunden, wird als Ersatzfont immer der erste der Liste genommen. Es wird keine einzelne Betrachtung von Name oder Format vorgenommen.

[7] Die Oberfläche und der Graphik-Editors bieten derzeit keine direkte Möglichkeit dafür. Die direkte Angabe anderer Attribute über die Source-Kommentare ist aber möglich. Auch die Oberfläche könnte eine Konfigurationdatei erhalten, die die möglichen Werte definiert.

- Ist die Fontgröße nicht vorhanden, wird immer die nächst kleinere Fontgröße gewählt. Ausnahme sind hier Fontgrößen, die kleiner als alle aufgeführten sind. Hier wird die erste Angabe der Größenliste gewählt.

Durch diese Regel kann eine Default-Schriftart definiert werden, die in der Liste als erstes aufgeführt wird. Die Schriftgrößen müssen nicht aufgelistet werden.

3.9.3 Dimensionen der einzelnen Zeichen

Der Generator braucht für die Formatierung der Texte die Größe der einzelnen Buchstaben in der aktuellen Schriftart. Die Breite eines Zeichen bestimmt die Position des nachfolgenden Zeichens. Die Höhe bestimmt den Zeilenabstand. Abbildung 3.25 zeigt die Dimensionen eines Zeichens. Neben Breite und Höhe gibt es auch eine Tiefe, die Zeichen mit Unterlängen vorhanden ist. Bei der Berechnung des Zeilenabstands muß die Tiefe mit einfließen.

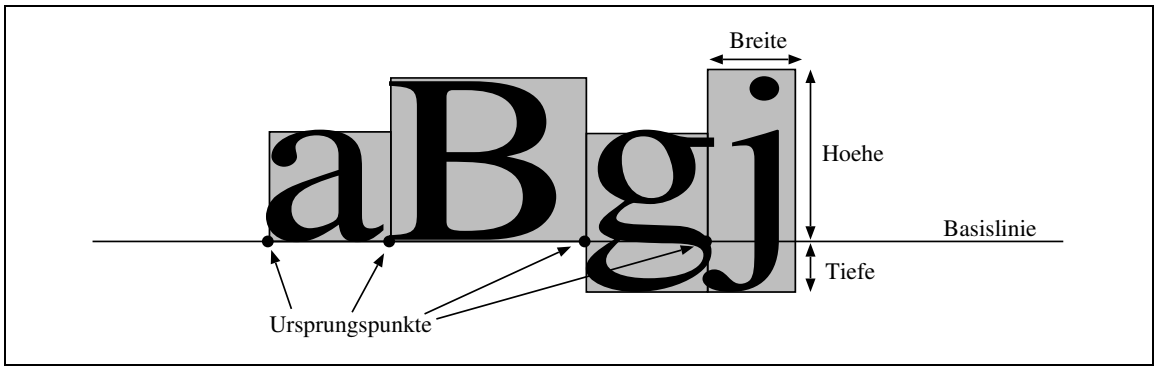


Abbildung 3.25: Dimensionen eines Zeichens im Ausgabeformat: Die BoundingBox des Zeichen beschreibt dessen Ausmaße und ist über Breite, Höhe und Tiefe bestimmt.

In PostScript besteht ein Zeichensatz aus einer Menge von Zeichen, die mit Hilfe einer Codetabelle den Positionen innerhalb der ASCII-Tabelle zugeordnet werden. Eine Ausnahme spielen hier die Umlaute, die im Normalfall nicht in der Codetabelle zugeordnet sind. In PostScript muß die Codetabelle entsprechend erweitert werden. Im *Tgif*- und *Xfig*-Format sind die Zeichen vorhanden.

Für jeder Schriftart müssen also für alle 256 Zeichen der Codetabelle die Dimensionen gespeichert werden. Da bei allen Ausgabeformaten die Schriften auf PostScript basieren und diese dort frei skalierbar sind, müssen die Dimensionen nur für eine Referenzgröße (z.B. 10pt) gespeichert werden. Der Generator ist damit selbst in der Lage die Dimensionen mit einem Größenfaktor zu multiplizieren.

Berechnung der Dimensionen

Die Dimensionen der Schriftzeichen sind zwar in speziellen Dateien verschiedener Textverarbeitungssystemen vorhanden, die Interpretation der Dateien ist aber zum Teil schwierig. Eine einfache Möglichkeit bietet der PostScript-Interpreter *Ghostscript* der unter Unix und Windows PostScript-Dateien am Bildschirm anzeigen kann. PostScript bietet zudem Befehle an, die von einem Text die Bounding-Box berechnen:

```
charpath flattenpath pathbbox
```

Normalerweise dient diese Bounding-Box als Grundlage für weitere graphische Aktionen. Mit dem Befehl `print` können die berechneten Koordinaten der umgebenen Box auch auf der Standardausgabe ausgegeben werden. Damit ist es möglich eine PostScript-Sequenz zu erstellen, die mit dem PostScript-Interpreter ausgeführt wird, und auf der Standardausgabe eine Liste der Dimensionen der einzelnen Schriftzeichen ausgibt. Diese Liste kann in eine Datei gespeichert und vom Fontgenerator wieder eingelesen werden.

Die oben beschriebene Methode liefert nicht die drei Attribute Breite, Höhe und Tiefe, sondern die Koordinaten der linken unten und rechten oberen Ecke der Bounding-Box bezogen auf einen

Ursprungspunkt, der links vom Zeichen auf der Basislinie liegt. Der Fontgenerator muß die Koordinaten beim Einlesen umrechnen.

```
>Times-Roman
0:2.5 -0.0 2.5 -0.0
...
33:1.30004883 -0.08984375 2.38037109 6.76001
34:0.770507812 4.31005859 3.31005859 6.76001
35:0.0500488281 -0.0 4.95996094 6.62011719
...
255:2.5 -0.0 2.5 -0.0
>Times-Italic
0:2.5 -0.0 2.5 -0.0
...
```

Abbildung 3.26: Mit einem PostScript-Interpreter erstellte Dimensionsdatei: Die Dimensionen der einzelnen Zeichen könnten in diesem Format abgespeichert werden. Die Information zu einer Schriftart werden mit deren Namen eingeleitet.

3.9.4 Effiziente Speicherung der Fontinformationen

Das Modul Fontgenerator muß eine große Menge von Informationen vorhalten. Zusätzlich werden diese Informationen während der Formatierung vom Generator oft gebraucht. Für die Speicherung der Daten wird daher eine Erweiterung der NXS-Struktur vorgenommen. Bei jedem Text-Objekt in der Teilstruktur `graph` ein Verweis auf eine `Realfont`-Struktur gespeichert. Diese enthält zur gewählten Schriftart alle nötigen Informationen. Abbildung 3.27 zeigt den Aufbau der Struktur.

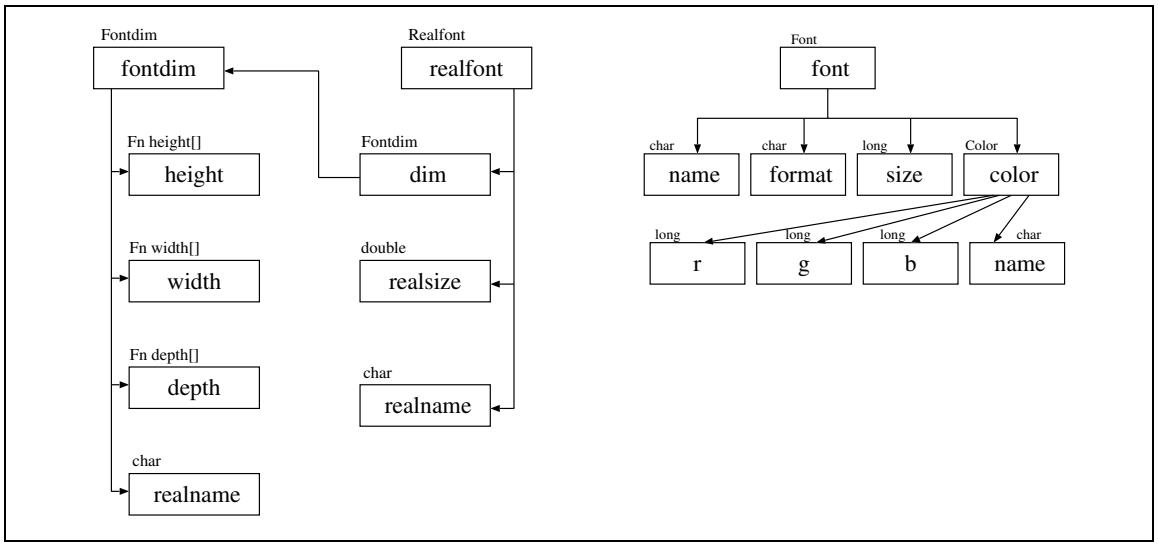


Abbildung 3.27: Nach außen sichtbare Datenstrukturen vom Modul Fontgenerator: Eine Schriftart, die über die rechte Struktur beschrieben wird, wird auf eine im Ausgabeformat vorhandene Schriftart abgebildet.

Der Fontgenerator bietet als Schnittstelle eine Funktion `fontmap()` an, die eine gewünschte Schriftart auf eine realen Schriftart abbildet. Die Funktion liefert nur einen Verweis auf diese Struktur zurück. Die Speicherverwaltung der Struktur bleibt beim Fontgenerator. Das ermöglicht ihm eine effiziente und Redundanz-freie Speicherung der Daten. Die Struktur `Realfont` enthält neben dem Namen und der tatsächlichen Größe eine weiteren Verweis auf die Dimensionen der einzelnen Zeichen. Diese zweigestufte Verknüpfung ist auch durch die Redundanz-freie Speicherung begründet.

Zur internen Speicherung müssen drei Listen verwaltet werden (siehe Abb. 3.28 auf der nächsten Seite). In der ersten Liste `allfonts` werden die Dimensionen aller verfügbaren Schriften gespeichert. Die Liste wird durch das Einlesen der Fontdim-Datei gefüllt und bleibt bis zum Programmende oder einem Wechsel des Ausgabeformats unverändert. Die Dimensionen der Schriftzeichen beziehen sich auf die Größe 10pt. Eine Umrechnung auf die gewünschte Schriftgröße wird vom Fontgenerator nicht vorgenommen, muß also bei der Formatierung durchgeführt werden.

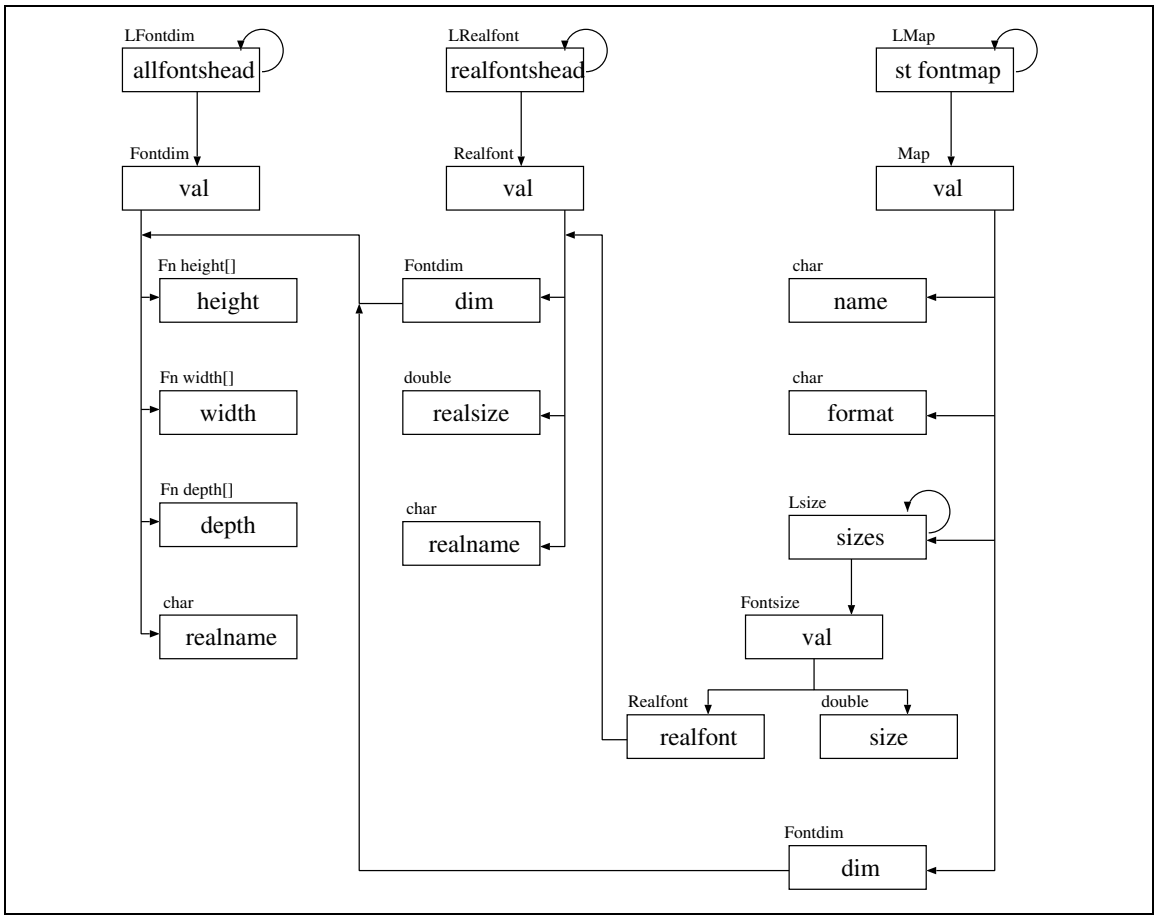


Abbildung 3.28: Interne Verwaltung der Fontinformationen: Die drei Listen sind nur für den Fontgenerator sichtbar und ermöglichen ihm eine effiziente Speicherung der Daten.

Die zweite Liste `Realfonts` enthält die alle bisher generierten `Realfont`-Strukturen. Da die Dimensionen nur für eine Schriftgröße gespeichert werden, können mehrere `Realfont`-Strukturen auf einen Eintrag der ersten Liste verweisen. Daher ist eine Speicherung in getrennten Listen sinnvoll. Diese zweite Liste ermöglicht auch, das die `Realfont`-Strukturen nur einmal pro Schriftart/größe existieren. Die Liste wird während des Programmablaufs vom Fontgenerator aufgebaut.

Die dritte Liste speichert die in der Fontmap-Datei beschriebene Abbildung der Schriftarten. Für jede Schriftart wird dort eine Liste von Größen angelegt. Zu jeder Größe kann in dieser Liste ein Verweis auf eine `Realfont`-Struktur gespeichert werden. Dieser Verweis wird nur angelegt, wenn eine Anfrage zu dieser Schriftart/größe vorhanden ist und dadurch ein `Realfont` angelegt wurde. Die Liste wird bei Schriftarten mit einer festen Anzahl von Schriftgrößen durch das Einlesen der Fontmap-Datei definiert. Ist eine freie Wahl der Schriftgröße möglich (z.B. bei PostScript) wird die Liste erst beim Programmablauf aufgebaut. Da bei einer Schriftart mehrere Größen auf eine Größe im Ausgabeformat abgebildet werden können, mußte diese Information getrennt von der Realfonts gespeichert werden.

Wechsel des Ausgabeformats

Die Informationen zu den Schriftarten ist zu groß, um sie für alle Ausgabeformate im Speicher zu halten. Daher ist führt ein Wechsel des Ausgabeformats führt beim Fontgenerator zu einer neuen Initialisierung der Datenstrukturen. Die Listen `Allfonts` und `Fontmap` werden durch das Einlesen der neuen Fontdim- bzw. Fontmap-Dateien erneuert. Die Liste `Realfonts` wird gelöscht. Die in der NXS gespeicherten Verweise auf die `Realfont`-Strukturen werden damit ungültig. Es muß von der Oberfläche und dem Fontgenerator sichergestellt werden, daß nach einem Wechsel des Ausgabeformats nicht mehr auf diese Verweise zugegriffen wird.

Kapitel 4

Implementierung

Wie schon in der Einleitung beschrieben, ist diese neue Entwicklung von Nassi durch die Ablösung der Großrechner angestoßen worden. Die bis dahin eingesetzte Version von Nassi war bedingt durch die Wahl der Programmiersprache VS Pascal und dem Graphiksystem GKS an das Großrechner-Betriebssystem VM/CMS eng gekoppelt. Für die neue Entwicklung war durch die Migration von VM/CMS zu Unix die Zielplattform schon vorgegeben. Auch bei der Wahl der Programmiersprache und des Graphiksystems müssen die Randbedingungen berücksichtigt werden.

Dieses Kapitel stellt in den ersten Abschnitten die Festlegung der für alle Module geltenden Implementationbedingungen, wie Programmiersprache, Entwicklungsumgebung und Plattformen vor.

Im Abschnitt Datenstrukturen wird die Implementation von abstrakten Datenstrukturen, wie Listen und Hashes beschrieben, die zuerst entwickelt werden und damit bei der Implementation der einzelnen Module zur Verfügung stehen.

Die nächsten Abschnitte beschreiben die Implementation der einzelnen Module. Dies geschieht anhand von ausgewählten Beispielen aus dem Source-Code. Eine komplette Beschreibung der Implementation wäre an dieser Stelle zu umfangreich.

4.1 Plattformen

Nassi soll vorwiegend auf Unix-Maschinen mit den Betriebsystemen AIX, Solaris, Irix und Digital Unix laufen. Eine Portierung zwischen diesen Unix-Derivaten sollte keine größeren Schwierigkeit darstellen. Einzige Besonderheit sind Irix (SGI) und Digital Unix, die mit einer 64bit Speicheradressierung arbeiten. Als Entwicklungsplattform dient eine Maschine IBM-RS/6000 (AIX). Als zweite Plattform steht eine Sun-Workstation zur Verfügung, auf der auch das Tool *purify* läuft. Die beiden anderen Plattformen werden erst in den letzten Schritten der Entwicklung mit berücksichtigt.

Für den Bereich der PC-Plattformen wird das Betriebsystem Linux hinzugenommen. Eine Entwicklung von Nassi unter Windows 95/NT ist derzeit nicht möglich.

4.2 Programmiersprache und Entwicklungsumgebung

Als Implementationssprache wurde für *Nassi* C (ANSI) gewählt. Für diese Programmiersprache stehen auf allen Plattformen Compiler zur Verfügung. Durch die Benutzung der GNU C-Compilers `gcc` ist eine leichte Portierung innerhalb der Unix-Plattformen möglich. Dieser Compiler steht in der gleichen Version auf allen Plattformen zur Verfügung. Es sollen bei der Implementation nur Sprach-Konstrukte des ANSI-Standards benutzt werden.

Als Libraries und Tools werden für den Parser `lex` und `yacc` [6] und für die Oberfläche `xforms` [5] und ggf. die X11-API benutzt.

Da die Entwicklung auf mehreren Maschinen und von mehreren Autoren vorgenommen wird, ist eine Version-Verwaltung des Source-Codes nötig. Dazu wird die Versionskontrolle CVS (Concurrent Version System, [7]) benutzt, die eine zentrale Repräsentation des Source-Codes (Repository) speichert. Jeder Entwickler arbeitet auf einer eigenen Kopie des Source-Codes und muß die Erweiterungen jeweils in das Repository einspeisen. Die anderen Entwickler werden davon per Mail informiert und können mit CVS die eigene lokale Kopie aktualisieren. Da beim Entwurf ein modulares Prinzip verwirklicht worden ist, werden Konflikte bei gleichzeitiger Änderung einer Stelle von zwei Entwicklern selten vorkommen.

Als Hilfmittel bei der Analyse von Fehlern stehen die Debugger `xldb` (IBM) und `ddd` zur Verfügung. Ein weiteres sehr mächtiges Hilfsmittel ist `purify` [4], das die Überprüfung von Speicherzugriffen (Heap) unterstützt. Damit können die bei C schwer zu findenden Speicherzugriffsfehler (Zugriff auf schon freigegeben Speicherplatz, doppeltes Freigeben von Speicherplatz, ...) auch behoben werden.

4.3 Datenstrukturen

Der Aufbau der Datenstruktur NXS ist im Kapitel Entwurf sehr genau beschrieben. In diesem Abschnitt wird daher auf die Darstellung der NXS verzichtet und stattdessen die Bereitstellung einer typsicheren Verwaltung von Listen- und Hashstrukturen erläutert.

Zum Austausch von Daten zwischen den einzelnen Modulen dient die C–Datenstruktur `NXS` (Nassi Xchange Structure). Sie ist zusammen mit weiteren Datentypen in `NXS.h` definiert. Von einigen dieser Typen werden Listen oder Hashes benötigt. Anstelle generischer Listen bzw. Hashes (über `void *`) wird für jeden Typ ein eigener Listen- oder Hash-Typ definiert. Der Vorteil dieses Ansatzes ist eine ”typensichere” Verwendung der Listen: der Compiler erkennt bereits Fehler durch Verwendung falscher Pointer-Typen. Die Implementierung erfolgt mit Hilfe der Präprozessors. Man definiert den Macro `VTYPE` auf den Typ, von dem eine Liste benötigt wird, und benutzt spezielle Include–Dateien zur Deklaration und Definition des Listentyps und der Funktionen zur Verwaltung der Listen. Hier stehen zwei Varianten zur Auswahl. Für einfache Datentypen (z.B. `int`, `double`, ...) `SLIST.[ch]` und für Zeiger–Typen `LIST.[ch]`. Die beiden Varianten unterscheiden sich in zwei Punkten. Zum einen werden bei den ”einfachen Listen” bei Parametern oder Rückgabewerten immer der Wert übergeben, bei dem ”Zeiger–Listen” nur ein Pointer auf die Variable. Der andere Unterschied ist, daß beim Löschen einer Zeiger–Liste die einzelnen Listenelemente explizit mitgelöscht werden (`free_type()` s.u.).

```
#define VTYPE Position
#include "LIST.h"
#undef VTYPE
```

definiert eine Zeiger–Liste vom Typ `Position` und deklariert Funktionen zum Zugriff auf diese Liste. Die Konstruktion

```
#define VTYPE Position
#include "LIST.h"
#include "LIST.c"
#undef VTYPE
```

definiert zusätzlich die Funktionen. Für einfache Listen ist jeweils `SLIST` anstelle von `LIST` zu verwenden.

Bei der Einführung einer neuen Zeiger–Liste (für den Basistyp `type`) müssen folgende Funktionen zur Verfügung gestellt werden:

```
type *new_type( ... , long fatal_mask)
```

Diese Funktion erzeugt ein Objekt des betreffenden Typs und belegt es mit Werten. Die Parameter sind typischerweise die einzelnen Strukturelemente. Mit dem letzten Parameter `fatal_mask` können temporär Flags der „fatal"–Option (vgl. die Beschreibung der Fehlerroutinen) gesetzt werden. Im Fehlerfall wird ein `NULL`–Pointer zurückgegeben, sonst ein Pointer auf das neue Objekt. Diese Funktionen sind in `nxs.c` definiert.

```
void free_type(type *obj)
```

Diese Funktion gibt den Speicherplatz des Objekts wieder frei. Insbesondere die Listentypen benutzen diese Funktion ihres Basistyps, um die Listenelemente zu löschen. Diese Funktionen sind in `nxs.c` definiert. Für einfache Listen sind solche Funktionen nicht erforderlich.

Wenn ein Listentyp zu einem Basistyp definiert wurde, so stehen folgende Funktionen für diesen Typ zur Verfügung (`[*]` meint hier, daß für Zeiger–Listen `*` für einfache Listen nichts einzusetzen ist):

`void free_Ltype(Ltype *lst)`	löscht eine Liste mitsamt aller Elemente
`Ltype *insert_Ltype(type [*]obj, Ltype *lst)`	fügt `obj` vor `lst` in eine Liste ein,
`Ltype *append_Ltype(type [*]obj, Ltype *lst)`	fügt `obj` nach `lst` in eine Liste ein,
`Ltype *push_Ltype(type [*]obj, Ltype *lst)`	hängt `obj` an das Ende der Liste `lst` an,
`Ltype *first_Ltype(Ltype *lst)`	gibt das erste Listenelement zurück
`Ltype *last_Ltype(Ltype *lst)`	gibt das letzte Listenelement zurück
`Ltype *cut_Ltype(Ltype *elem)`	schneidet ein Element aus einer Liste aus und löscht es. Rückgabewert ist jeweils das neue Listenelement
`long nelem_Ltype(Ltype *lst)`	gibt die Anzahl der Elemente der Liste zurück

Ein Hash besteht hier aus einer Zuordnung eines Objekts eines beliebigen Typs (`HTYPE`) zu einem Text. Es können Hashes einfacher Datentypen (`SHASH`) oder von Zeiger–Typen (`HASH`) durch die Konstruktion

```
#define HTYPE Position
#include "HASH.h"
#undef VTYPE
```

deklariert. Mit

```
#define HTYPE Position
#include "HASH.c"
#undef VTYPE
```

werden die zugehörigen Funktionen definiert. Aufgrund der erhöhten Komplexität (jeder Hash enthält Listen eines Symbol–Typs) ist es erforderlich, für den Definitionsteil jeweils eine eigene Datei anzulegen. Für den Typ `type` existieren anschließend folgender Datentypen:

```
typedef struct stype Stype;
struct stype {
  char   *name;      type   [*]val;
};
typedef struct htype Htype;
struct htype {
  long   size;       LStype **table;
};
```

Der erste stellt einen Eintrag in der Symboltabelle dar, auf den über `name` zugegriffen werden kann.

Der zweite ist das gesamte Hash. Folgende Funktionen zum Zugriff werden definiert:

`Htype *new_Htype(long size, long mask);`	erzeugt ein neues Hash (zu `mask` vgl. Listen)
`find_Stype(char *name, Htype *hash);`	sucht ein Element eines Hashes
`new_Stype(char *name, type [*]val, long mask);`	erzeugt ein neues Hashelement (nur für interne Zwecke)
`add_Stype(char *name, type [*]val, Htype *hash);`	hängt ein neues Element in den Hash ein
`del_Stype(Stype *sym, Htype *hash);`	löscht ein Element aus dem Hash
`get_Stype_name(Stype *sym, Htype *hash);`	übergibt den Namen (`sym->name`)
`get_Stype_val(Stype *sym, Htype *hash);`	übergibt den Wert (`sym->val`)
`free_Htype(Htype *hash)`	löscht ein Hash (und alle Element–Werte, falls kein `SHASH`)

4.4 Fehlerbehandlung

Um eine einheitliche Behandlung von Fehlerfällen zu gewährleisten wird für die Implementierung von Nassi dazu eine Bibliothek bereitgestellt.

Funktionen und Macros zur Fehlerbehandlung sind in `../common/ne.[ch]` deklariert bzw. definiert. Alle Funktionsnamen beginnen mit `ne_` alle Macros mit `NE_` .

Ein Fehler besteht aus einem Fehlercode (`errno`) und einem Text, der den Fehler beschreibt (implementiert ist der Text als Liste von Strings, was die Manipulation erleichtert). Funktionen erlauben das Setzen und Abfragen des Fehlercodes. Der Fehlertext kann gesetzt, gelöscht und erweitert werden. Optional kann der Fehlertext den Fehlercode und/oder den Namen der Quelldatei und die Zeilennummer, in dem der Fehler aufgetreten ist, enthalten.

Weitere Optionen erlauben es, auftretende Fehler explizit in fatale oder nicht fatale umzuwandeln oder die Abarbeitung des Fehlers zu verhindern – typischerweise, um ihn zu einem späteren Zeitpunkt zu behandeln.

Zur Einbindung in C-Programme werden folgende Header benötigt:

```
#include <stdarg.h>   /* wegen der variablen Argumentlisten */
#include "ne.h"       /* Funktionsdeklarationen und Macros */
```

4.4.1 Fehlercodes

Je nach Fehlercode fallen die Fehler in 4 Klassen. Der Fehlercode `0` bedeutet keinen Fehler. Dann gibt es Systemfehler, fatale Fehler, nicht_fatale Fehler und Warnungen. Fatale Fehler führen zum Beenden des Programms. Bei nicht_fatalen Fehlern und Warnungen wird das Programm nach Ausgabe der Fehlermeldung fortgesetzt. Systemfehler werden nur vom System behandelt. Zur Berechnung der Fehlercodes führt folgende Formel: `fehlercode = klasse + modul + fehler` Dabei können die Werte für die Klassen und Module der Abbildung 4.1 entnommen werden.

4.4.2 Funktionen

```
void ne_seterrno(long errno)
```

Mit dieser Funktion wird der Fehlercode gesetzt. Der Wert wird durch die ”fatal”–Option (vgl. `ne_setfatal()`) beeinflußt:

`NE_NOFATAL`	ist dies gesetzt, so werden fatale Fehler in nicht fatale gewandelt
`NE_FATAL`	ist dies gesetzt (und `NE_NOFATAL` nicht), so werden nicht fatale Fehler in fatale gewandelt (bei `errno > 2*NE_MAXFATAL` wird der Fehlercode auf `NE_MAXFATAL` gesetzt).

NE_SYS=0	Systemfehler
NE_FATAL=1000	fataler Fehler
NE_ERROR=2000	nicht fatale Fehler
NE_WARN=3000	Warnungen
NE_PARSER=0	Fehler des Parsers
NE_UPDATE=100	Fehler des update-Moduls
NE_FONTGEN=200	Fehler des Fontgenerators
NE_GENER=300	Fehler des Generators
NE_OUTPUT=400	Fehler des Output-Moduls
NE_GREDIT=500	Fehler des graphischen Editors
NE_SRCEDIT=600	Fehler des Source-Editors
NE_GUI=700	Fehler der Oberfläche

Abbildung 4.1: Fehlerbehandlung: Festlegung der Fehlercodes

```
void ne_settext(char *fmt, ...)
void ne_vsettext(char *fmt, va_list ap)
```

Mit dieser Funktion wird eine Fehlermeldung erzeugt. Die Parameter sind wie bei den C–Library Funktionen `printf()` bzw. `vprintf()`. Falls bereits ein Fehlermeldungstext existiert, wird der neue Text **vor** diesem eingefügt. Der erzeugte Text wird durch die ”debug”–Option (vgl. `ne_setdebug()`) beinflußt:

NE_ERRNO	Fehlermeldung wird erweitert um: `Error <errno>:`
NE_POS	Fehlermeldung wird erweitert um: `Error in file "<file>" line <line>:` die Werte von `<file>` und `<line>` beziehen sich auf den letzten Aufruf einer Funktion, die den Fehlercode setzt (also `ne_seterrno, ne_seterror, ne_error`)

```
void ne_pushtext(char *fmt, ...)
```

Diese Funktion erzeugt wie `ne_settext()` einen Fehlertext, der durch die ”debug”–Option beeinflußt werden kann. Für `<file>` und `<line>` wird aber die Position des `ne_pushtext()`–Aufrufs in der C–Quelle und nicht die des `ne_seterrno()`–Aufrufs benutzt.

```
void ne_seterror(long errno, char *fmt, ...)
void ne_vseterror(long code, char *fmt, va_list ap) ***
```

Nur eine Kombination von `ne_seterrno()` und `ne_settext()` bzw. `ne_vsettext()`, d.h. es werden Fehlercode und Fehlermeldung gesetzt.

```
void ne_appendtext(char *fmt, ...)
void ne_vappendtext(char *fmt, va_list ap)
```

Diese Funktionen (Parameter wie `printf()` bzw. `vprintf()` hängen einen Text an die bestehende Fehlermeldung an. Im Gegensatz zu `ne_settext()` wird die ”debug”–Option nicht ausgewertet.

```
void ne_cleartext()
```

Diese Funktion löscht die bestehende Fehlermeldung (nur den Text, nicht den Fehlercode).

```
void ne_reset()
```

Diese Funktion löscht den Fehlertext und setzt den Fehlercode auf `0` zurück. Ein Fehler kann so zurückgesetzt werden, ohne daß Fehlermeldungen ausgegeben werden.

```
void ne_dispatch()
```

Diese Funktion gibt die Fehlermeldung aus, löscht anschließend die Fehlermeldung und setzt den Fehlercode auf `0` zurück. Ist der Fehler fatal, wird das Programm mit exit-Code `1` beendet. Ist die

”fatal”–Option NE_DELAY gesetzt, so passiert gar nichts. (Diese Option erlaubt es, die Fehlerbehandlung auf einen späteren Zeitpunkt zu verschieben).

```
void ne_error(long errno, char *fmt, ...)
```

Diese Funktion erzeugt einen Fehler mit Nummer und Text und führt anschließend die Fehlerbehandlung durch (Der Ablauf ist mit ne_seterror(); ne_dispatch() identisch).

```
long ne_setdebug(long debug)
long ne_adddebug(long mask)
long ne_cleardebug(long mask)
```

Diese Funktionen manipulieren die ”debug”–Option:

ne_setdebug	setzt die Option auf einen Wert
ne_adddebug	setzt einzelne Flags (Oder-Verknüpfung mit dem alten Wert)
ne_cleardebug	löscht einzelne Flags

Folgende Flags sind zur Zeit definiert:

NE_ERRNO	ergänzt Fehlermeldungstext um Fehlercode
NE_POS	ergänzt Fehlermeldungstext um Dateiname und Zeilennummer beim Auftreten des Fehlers

Der Rückgabewert ist jeweils der neue Wert der ”debug”–Option.

```
long ne_setfatal(long fatal)
long ne_addfatal(long mask)
long ne_clearfatal(long mask)
```

Diese Funktionen manipulieren die ”fatal”–Option in gleicher Weise wie die entsprechenden Funktionen für die ”debug”–Option. Folgende Flags sind zur Zeit definiert:

NE_DEFAULT	kein Flag (0)
NE_NOFATAL	wandelt fatale in nicht-fatale Fehlercodes
NE_FATAL	wandelt nicht-fatale in fatale Fehlercodes
NE_DELAY	verhindert die Fehlerbehandlung (vgl. ne_dispatch())

```
long ne_geterrno()
long ne_getdebug()
long ne_getfatal()
```

Mit diesen Funktionen können Fehlercode und die Werte der ”debug”– und ”fatal”– Optionen abgefragt werden.

4.4.3 Beispiele

Minimal-Beispiel

```
#include <stdarg.h>  /* f"ur variable Argumentlisten ben"otigt */
#include "ne.h"

void main() {
  /* Fehlermeldungen um Extra-Infos erg"anzen */

  ne_setdebug( NE_ERRNO | NE_POS );

  /* Fehler erzeugen ung sofort bearbeiten
     NE_STSYNTAX ist ein in "ne.h" definiert */
  ne_error(NE_STSYNTAX, "Und Tschuess !!\n");

  printf("hier kommen wir nie hin, weil NE_STSYNTAX < NE_MAXFATAL\n");
}
```

Ausgabe:

```
Error 600 in file "demo.c" line 9:
Und Tschuess !!
```

Verzögerte Fehlerbehandlung

```
/* Diese Funktion zum "Offnen einer Datei ist in nu.c definiert */

FILE *nu_fopen(char *fname, char *mode, long mask) {
  FILE *p;
  long ofatal;

  /* neue "fatal"-Optionen setzen */
  ofatal = ne_addfatal(mask);

  if((p=fopen(fname, mode)) == NULL) {
    /* Dieser Fehler enth"alt den vom System zur Verf"ugung gestellten
       Text, und etwas mehr. */
    ne_error(errno, "Error opening \"%s\" with mode \"%s\": %s\n",
                fname, mode,strerror(errno));
  }
  /* die urspr"ungliche "fatal"-Option restaurieren */
  ne_setfatal(ofatal);

  return p;
}

#include <stdio.h>
#include <stdarg.h>
#include "ne.h"
#include "nu.h"

void main() {
  FILE *out;

  ne_setdebug( NE_ERRNO | NE_POS );
  ne_setfatal( NE_NOFATAL );

  /* Hier tritt ein fataler Fehler auf, wegen des NE_DELAY Flags
     wird er aber noch nicht bearbeitet */
  if((out=nu_fopen("/verboten", "w", NE_DELAY)) == NULL) {
    /* deshalb k"onnen wir wir die Fehlermeldung noch erg"anzen,
       hier wollen wir keinen eigenen Text hinzuf"ugen, sondern nur
       die Position des Fehlers in der Quell-Datei ausgeben */
    ne_pushtext("");
    ne_dispatch();
  }
  printf("Hier kommen wir hin, weil der Fehler nicht fatal war\n");
}
```

Ausgabe:

```
Error 1013 in file "demo.c" line 17:
Error 1013 in file "nu.c" line 49:
Error opening "/verboten" with mode "w": Permission denied
Hier kommen wir hin, weil der Fehler nicht fatal war
```

4.5 Modul Oberfläche

Das Modul Oberfläche bildet die Benutzerschnittstelle. Als graphisches Toolkit wurde die Forms Library gewählt.

Das Hauptfenster hat ein festes Layout und kann deshalb mit Hilfe des Gui-Generators *fdesign* erzeugt werden. Die Gestaltung der modalen Optionen-Dialoge hingegen ist abhängig von den anderen Modulen und muß deshalb variabel gestaltet werden. Da die Forms-Library keine positionsunabhängigen Layoutmanager bereitstellt, muß die Position der einzelnen Widgets in einem zweispaltigen Layout berechnet werden.

```
  num=nelem_LGeneral_opt(list);

  /* get size of the new form */
  get_metrics(list,&left_label,&right_label,&form_height);

  /* create dialog */
  local_form = fl_bgn_form(FL_UP_BOX, left_label+right_label+2*OFF_X,
                           form_height+3*OFF_Y+BUT_WIDTH);
  ...

  /* array of all objects */
  obj=nu_malloc(num*sizeof(Obj),NE_DEFAULT);
  p=list;
  for (i=0; i<num; i++) /* Create each object */
   {
     obj[i]=create_object(p,&y_value,left_label,right_label);
     p=p->next;
   }
```

Abbildung 4.2: create_options: Die in `list` übergebenen Optionen werden in einem Dialog dargestellt. Die Höhe des Dialogs wird durch die Anzahl der Elemente bestimmt. Die Breite ist die Summe des breitesten Beschreibungstextes in der linken Spalte und der breitesten Komponente in der rechten Spalte. Diese Geometriedaten werden in `get_metrics` berechnet.

Die einzelnen Gui-Komponenten werden in `create_object` erzeugt und in den modalen Dialog eingefügt.

```
Obj create_object(LGeneral_opt *list, FL_Coord *y_value
                  , int left_label, int right_label)
{
 FL_IOPT cntl;

  fl_get_defaults(&cntl);
  switch(list->val->show){
  case DOUBLESLIDER:
    obj.o= fl_add_valslider(FL_HOR_SLIDER,OFF_X+left_label,*y_value,
                            right_label,OBJ_HEIGHT,list->val->description);
    fl_set_slider_precision(obj.o,MAX(
                         (int)(-1.*log10(list->val->option.double_slider.step)+.5),0));
    fl_set_slider_bounds(obj.o,list->val->option.double_slider.min
                         ,list->val->option.double_slider.max);
    fl_set_slider_step(obj.o,list->val->option.double_slider.step);
    fl_set_slider_value(obj.o,list->val->value.double_value);
    fl_set_object_lalign(obj.o,FL_ALIGN_LEFT);
    fl_get_defaults(&cntl);
    fl_set_object_lsize(obj.o,cntl.labelFontSize);
   break;
    ...
  *y_value+=OBJ_HEIGHT+get_space(list->val->show);
```

Abbildung 4.3: create_object: abhängig vom Typ in `list->val->show` wird die entsprechende Gui-Komponente erzeugt und positioniert.

Jeder Dialog enthält die Buttons *Ok, Apply* und *Cancel*. Mit Ok werden die Werte übernommen und der Dialog beendet. Cancel verläßt den Dialog und ignoriert ggf. vorgenommene Änderungen. Apply bietet eine Vorschau der Änderungen. Der Dialog wird nicht beendet und die Änderungen können ggf. mit Cancel verworfen werden.

Abbildung 4.4 auf der nächsten Seite zeigt die Schleife, in der die auftretenden Events behandelt werden. Mit `get_options` werden die vom Benutzer eingegebenen Werte aus dem Dialog ausgelesen. Mit `apply_options` wird die externe Modul-Funktion `apply_opt_`*`Modul`* aufgerufen, die die Änderungen aktiviert. Die Änderungen werden mit Aufruf von `redraw_canvas` im Graphikeditor angezeigt.

Die Datenstruktur `LGeneral_opt` wird von den Modulen bereitgestellt. Sie enthält die Informationen, die zur Erstellung der Dialoge notwendig sind.

```
  /* handle form events */
  while (((objs = fl_do_forms())!=ok) && (objs!=cancel))
    if (objs == apply) { /* apply button pressed */
      get_options(list,local_form,obj);
      apply_options(list)
    }

  if (objs == cancel) { /* cancel button pressed */
    retrieve_General_opt(list,copy);  /* retrieve values from copy */
    apply_options(list);
  }
  else { /* ok button pressed */
    get_options(list,local_form,obj); /* get values from the dialog */
    apply_options(list);
  }
```

Abbildung 4.4: Eventloop in create_options: Nur das Aktivieren eines der Buttons *Ok, Apply* oder *Cancel* wird bearbeitet. Nach Aktivierung von *Ok* oder *Cancel* wird der Dialog beendet.

```
typedef struct general_opt General_opt;

/* define List of general_opt */
#define VTYPE General_opt
#include "LIST.h"
#undef VTYPE

struct general_opt{
  char            *description; /* label in the GUI */
  char            *id;          /* unique id of this element */
  Flag            dirty;        /* true, if value has changed */
  Flag            need_init;    /* true, if element changes other elements */
  Flag            locked;       /* true, if element isn't active */
  Opt_show_type   show;
  Opt_type        option;
  Value           value;
};

/* choices to show Option */
typedef enum { COUNTER, DOUBLESLIDER,  ON_OFF,
 TEXT,  SUBMENU, RADIOBUT, CHOICE, HYPHEN_INPUT,
 OPT_SLIDER, OPT_COUNTER, POPUP                 } Opt_show_type;

union value_of_option {
  long     long_value;
  double   double_value;
  char     *text;
  Flag     on_off;
};
typedef union value_of_option Value;


/* this may hold any form of Option */
union opt_type{
  Counter         long_counter;
  Doubleslider    double_slider;
  Opt_slider      opt_slider;
  Opt_counter     opt_counter;
  char            *text;
  LGeneral_opt    *submenu;
  LChoice         *choice;
  Popup           popup;
};
typedef union opt_type Opt_type;
```

Abbildung 4.5: Datenstruktur `LGeneral_opt`

4.6 Modul Parser

Hauptaufgabe des Parsers ist der Aufbau des Syntaxbaumes und das Füllen der Datenstruktur NXS mit den Ptext-Listen. Hier wird der Weg dahin exemplarisch an Code-Beispielen des C Pseudocode-Parsers gezeigt. Der erste Schritt besteht darin, den Quelltext in Token zu zerlegen. Der C-Code dafür wird von dem Werkzeug flex aus einer Beschreibung generiert, die jeweils regulären Ausdrücken, die die Token beschreiben, eine Aktion (ein C-Code-Fragment) zuordnet. Der so generierte „Lexer“ gibt bei jedem Aufruf ein Token zurück, muß also wiederholt aufgerufen werden, bis die gesamte Eingabedatei verarbeitet ist. Das folgende Beispiel zeigt, wie Identifier (also Variablen- und Funktionsnamen) erkannt und behandelt werden.

```
L               [A-Za-zäöüÄÖÜß_]
D               [0-9]
ID              (L|\\\"L)(L|D|\\\"L|::~|::)*

<INITIAL>ID    act_Ptext(T_ID); return TOK_ID;
```

Die ersten drei Zeilen definieren Abkürzungen für reguläre Ausdrücke. Ein Identifier (`ID`) beginnt mit einem Buchstaben oder Unterstrich (`L`), dem optional weitere Buchstaben oder Ziffern folgen. Im Pseudocode gelten auch deutsche Umlaute als Buchstaben, die auch in der LaTeX-Notation (`\"u` für ü) eingeben werden dürfen. Eine weitere Besonderheit ist, daß auch „`::`“ und „`::~`“ erlaubt sind. Dies ermöglicht es, einfache C++ Programme mit dem C Parser zu verarbeiten.

Der im zweiten Teil stehende C-Code wird ausgeführt, wenn ein Identifier erkannt wird während der Lexer sich in dem Zustand `INITIAL` befindet. Dieser Zustand ist stets aktiv, wenn C Statements oder Deklarationen verarbeitet werden. In andere Zustände wird z.B. umgeschaltet, wenn Kommentare, Präprozessordirektiven, Strings oder NSCs verarbeitet werden. Der Aufruf von `act_Ptext` generiert ein Ptext-Fragment vom Ttype `T_ID` (für Identifier), das den gesamten Quelltext vom Ende des letzten gefunden Tokens bis zum Ende des Identifiers enthält. Insbesondere werden Kommentare und Leerzeichen als „hidden“ Text diesem Token zugeordnet. Der Rückgabewert `TOK_ID` ist eine Zahl, anhand derer der Parser das Token als Identifier erkennt.

Der von *bison* generierte Parser ist auf die Zusammenarbeit mit dem Lexer abgestimmt. Er ruft diesen im Verlauf seiner Arbeit auf, wann immer er ein neues Token benötigt. Der Parser wird aus einer rekursiven Beschreibung der Syntax der Programmiersprache erzeugt. Diese definiert sogenannte Symbole als eine Abfolge anderer Symbole oder einzelner Token (sog. Terminalsymbole). Einer solchen Symboldefinition kann ein C-Code zugeordnet werden, der ausgeführt wird, wenn das Symbol erkannt wurde. Das folgende Beispiel zeigt die Definition einer Do-While Schleife, wie unter 3.4.1 beschrieben.

```
repeat_statement
        : TOK_DO statement TOK_WHILE '(' any_stuff ')' ';'
                {
                  DPRINTF(("until "));
                  nl_hide($1,YES);
                  nl_hide($3,YES);
                  nl_hide($<ptext>4,YES);
                  nl_hide($<ptext>6,YES);
                  nl_hide($<ptext>7,YES);
                  $$ = nl_mkNXS(st_count++, N_REPEAT,
                                join_LPtext(7,$1,new_Lslot,$3,$<ptext>4,
                                $5,$<ptext>6,$<ptext>7));
                  $$->body.nxs = $2;
                }
        ;
```

Die Schleife besteht aus dem Schlüsselwort `do`, gefolgt von einem Statement (ein an anderer Stelle definiertes Symbol, das auch Blöcke enthält). Es folgt das Schlüsselwort `while` und eine in Klammern gesetzte Abbruchbedingung und zuletzt ein Semikolon. Der zugeordnete Code erzeugt (mit `nl_mkNXS`) einen neuen NXS-Knoten im Syntaxbaum vom Nodetype `N_REPEAT`. Der NXS, der das `statement` repräsentiert (der zugehörige Code ist bei der Definition des Symbols

statement angegeben) wird als „Sohn“ an den neuen Knoten angehängt. Ein Parameter von nl_mkNXS ist die Liste der Ptexte, die die Schleife beschreiben. Diese enthält genau die in 3.4.1 beschriebenen Komponenten, insbesondere auch einen „Slot“, in den der Ptext des Sohn-NXS einzufügen ist. Mit nl_hide werden zuvor die Ptexte, die vom Generator ignoriert werden sollen, als „hidden“ markiert.

Die Verarbeitung der Quelltexte zu einem NXS-Baum verläuft weitgehend wie in den obigen Beispielen dargestellt. Ein zusätzlicher Aufwand entsteht lediglich dadurch, daß der Parser auch Funktionsaufrufe als solche erkennen soll. Beim eigentlichen Parser-Lauf wird eine Liste von Funktionen erstellt, über die von der Oberfläche aus die Generierung der Diagramme zu diesen Funktionen angestoßen wird. Wenn diese Liste und der NXS-Baum vollständig vorliegen, wird der Baum noch einmal durchlaufen und alle Identifier daraufhin geprüft, ob sie in der Liste der Funktionen stehen. Alle Statements, die einen solchen Identifier enthalten, werden als Funktionsaufruf markiert.

4.7 Modul Generator

Der Generator steht bei der Erstellung der Nassi-Shneiderman-Diagramme in einer zentralen Rolle. Er muß den Programmtext des Eingabeprogramms, der hierarchisch angeordnet in der NXS gespeichert ist, in die graphische Darstellung eines Nassi-Shneiderman-Diagramms überführen.

Im wesentlichen sind dazu rekursive Durchläufe durch die NXS-Struktur nötigt, die in den einzelnen Knoten der NXS die Komponente graph mit Inhalt füllen. Für jeden NXS-Typ (While, ...) steht im Generator eine einzelne Routine zur Verfügung, die die entsprechende Graphikobjekte generiert (work_on_*type*). Dazu gibt es eine zentrale Routine work_on_, die als Argument einen Verweis auf einen bisher noch nicht weiter untersuchten NXS-Knoten erhält. Diese bestimmt den Typ des Knoten und ruft die dafür spezifizierte Routine auf. Diese Routinen rufen ggf. wieder die allgemeine Routine work_on_ auf, um die in dieser Struktur enthaltenen Statements zu bearbeiten (siehe Abb. 4.6).

```
switch (element->type) {
 case N_FUNCTION  : { error = work_on_function(element,poslu,dim,factor,
                                               deep+1,font_info,
                                               key_font_info,colors,old_fcall);
                      break; }
 case N_STATEMENT : { error = work_on_simple_statement(element,poslu,dim,factor,
                                                       deep+1,font_info,
                                                       key_font_info,colors,old_fcall);
                      break; }
 case N_IF        : { error = work_on_if(element,poslu,dim,factor,deep+1,
                                         font_info,key_font_info,colors,old_fcall);
                      break; }
 case N_WHILE     : { error = work_on_while(element,poslu,dim,factor,
                                            deep+1,font_info,key_font_info
                                            ,colors,old_fcall);
                      break; }
 case N_REPEAT    : { error = work_on_repeat(element,poslu,dim,factor,
                                             deep+1,font_info,
                                             key_font_info,colors,old_fcall);
                      break; }

  ...

 default          : { printf("unknown: %d\n",element->type);
                      break; }
}
```

Abbildung 4.6: generator: Indirekte Rekursion bei der Bearbeitung des NXS-Baums: Die in dieser Routine aufgerufenen Routinen work_on_*type* bearbeiten den entsprechenden Knoten im NXS-Baum und rufen dabei ggf. die Routine work_on_ wieder auf (Indirekte Rekursion).

Die Routinen, die die einzelnen Typen der NXS-Knoten bearbeiten (work_on_*type*), sind alle in einer ähnlichen Weise aufgebaut. Dieser Aufbau soll an Hand der Routine work_on_repeat

vorgestellt werden. Die einzelnen Routinen unterscheiden sich durch die unterschiedliche Formatierung der Elemente im Nassi-Shneiderman-Diagramme innerhalb der einzelnen Schritte so sehr, daß eine Zusammenfassung zu einer Routine zu kompliziert wäre.

```
long work_on_repeat( ... )
{
  /*********************************************************/
  /* 1) create new structure graph in element              */
  /*********************************************************/
  if (element->graph) {
      free_NXGraph(element->graph);
      element->graph = NULL;
  }
  help_dim = new_Position(dim->x,0.0,0);
  pos          = new_Position(poslu->x,poslu->y,0);
  element->graph = new_NXGraph(help_dim,pos,NULL,NULL,0);

  /*********************************************************/
  /* 2) read hints from element-gen                        */
  /*********************************************************/
  ...

  /*********************************************************/
  /* 3) work on body of repeat                             */
  /*********************************************************/
  body_dim   = new_Position(dim->x - margin,0.0,0);
  pos        = new_Position(poslu->x + margin,poslu->y,0);
  error = work_on_(element->body.nxs,pos,body_dim,factor,deep,actual_outfont,
                   actual_key_outfont,actual_colors,new_fcall);
  free_Position(pos);

  /*********************************************************/
  /* 4) add dimension of body to reach beginning of condition*/
  /*********************************************************/
  element->graph->dim->y = body_dim->y;

  /*********************************************************/
  /* 5) generate condition                                 */
  /*********************************************************/
  help_dim = new_Position(dim->x,0.0,0);
  pos = new_Position(poslu->x,poslu->y + element->graph->dim->y,0);
  error += work_on_statement(element,pos,help_dim,factor,deep,actual_outfont,
                             actual_key_outfont,actual_colors,new_fcall);

  /*********************************************************/
  /* 6) make polygon                                       */
  /*********************************************************/
  lpoint = make_polyline(12,poslu->x,poslu->y,
                         poslu->x+margin,poslu->y,
                         poslu->x+margin,poslu->y+body_dim->y,
                         poslu->x+margin+body_dim->x,poslu->y+body_dim->y,
                         poslu->x+margin+body_dim->x,poslu->y+element->graph->dim->y,
                         poslu->x,poslu->y+element->graph->dim->y);
  pline = new_Gpolyline(lpoint,YES,actual_colors[LINE],actual_colors[FILL],0);
  if (element->graph->lobj)
    append_LGobject(new_Gobject(G_POLYLINE,pline,0),element->graph->lobj);
  else
    element->graph->lobj = append_LGobject(new_Gobject(G_POLYLINE,pline,0),
                                           NULL);
  *dim = *element->graph->dim;

  ...

  return(error);
}
```

Abbildung 4.7: generator: Bearbeitung einer Repeat-Anweisung

Abbildung 4.7 zeigt die wichtigsten Teile der Routine `work_on_repeat`. Folgende Schritte werden von dieser Routine durchgeführt:

1. Da in der Teilstruktur `graph` noch Graphikobjekte aus vorherigen Generator-Aufrufen enthalten kann, wird diese gelöscht und wieder neu angelegt. Dazu steht für jede Komponente der NXS-Struktur eine Routine `free_...` zur Vefügung (siehe Kapitel 4.3 auf Seite 64).

2. Danach werden aus dem Knoten die lokal veränderten Hints ausgelesen. Diese legt der Parser in der Teilstruktur `gen` ab. Die lokalen Änderungen müssen mit denjenigen Werten vermischt werden, die vom übergeordneten Knoten an `work_on_repeat` weitergegeben worden sind. Die so angepaßten Werte werden bei der Formatierung dieses Statements und den ggf. im NXS-Baum darunter liegenden Statements genutzt.
3. Da in der Repeat-Schleife zuerst der etwas nach rechts eingerückte Inhalt der Schleife gezeichnet wird, muß dieser Inhalt zuerst vom der Routine gezeichnet werden. Dazu wird wieder die Routine `work_on_` mit dem Body-NXS-Konten aufgerufen (Indirekte Rekursion). Die Routine erhält die geänderten Koordinaten (`pos` und `dim`) und gibt die neue linke untere Ecke über die gleichen Variablen zurück.
4. Als nächstes muß die Bedingung der Repeat-Schleife gezeichnet werden. Da deren Formatierung der eines normalen Statements entspricht, wird dazu auch diese Routine (`work_on_statement`) benutzt. Danach sind in der `graph`-Struktur die formatierten Texte abgespeichert.
5. Zum Schluß müssen noch die verbliebenen Linien der Repeat-Schleife gezeichnet werden. Dies geschieht mit Hilfe einer Polyline.

Als ein zweites Beispiel für die Implementierung des Generators wird die zentrale Routine für den Textumbruch aufgeführt (Abb. 4.8 auf der nächsten Seite). Im Entwurf wurde dafür ein Zustandsübergangsgraph vorgeschlagen. Dieser läßt sehr leicht mit einer Schleife und einer – etwas längeren – Switch-Anweisung implementieren.

Die Routine erhält eine Liste von GTexten, deren einzelnen Elemente die zu formatierenden Textstücke enthalten. Die Positions- und Dimensionsvariablen der Elemente sind noch unbesetzt. Die Berechnung der Position und auch die Größe der Textelemente ist die Aufgabe der Routine `fits_text`.

Für die Berechnung des Zeilenvorschubs wird als erstes die maximale Texthöhe berechnet. In der nachfolgenden Schleife wird für jedes Textstück untersucht, ob Trennvorschläge vorliegen. Ist dies der Fall wird das Textstück aufgeteilt. Da nach dem Aufruf der Routine `insert_hyphen_in_text` die Liste verändert sein kann, muß schon vorher ein Zeiger auf das nächste Element zwischengespeichert werden.

In der folgenden Schleife wird für jedes Textstück dessen Länge berechnet, indem die Längen der einzelnen Zeichen aufeinander addiert werden.

Da die Routine sowohl für den Textbruch im Blocksatz als auch für den Textsatz in der If-Bedingung zuständig ist, muß die Anfangsposition eine jeden Zeile berechnet werden (`search_line`).

In der nun folgenden Schleife beginnt der Durchlauf durch den Zustandsübergangsgraphen. Die Variable enthält den aktuellen Zustand. Erst wenn dieses Variable den Wert `F_END` enthält ist der Durchlauf beendet und der Text formatiert.

4.8 Modul Update

Wie bereits im Kapitel Entwurf beschrieben, beschränkt sich die Aufgabe des Moduls Update darauf, den NXS-Baum zu durchlaufen und alle Ptexte auszugeben. Diese triviale Aufgabe soll hier nicht detailierter beschrieben werden. Zusätzlich übernimmt das Modul noch die Aufgabe, die „hints“ im Quelltext als NSCs zu aktualisieren. Dies ist in der Funktion `hints_to_ptext` gekapselt. Vor der Ausgabe wird der NXS-Baum einmal durchlaufen und diese Funktion für jeden NXS einmal aufgerufen. Unproblematisch sind die Fälle, in denen bereits vorhandene hints ersetzt oder gelöscht werden müssen. Dazu wird im Ptext des Statements nach dem NSC-Token gesucht. Dessen Text wird entweder ersetzt oder gelöscht. Soll ein Statement, das zuvor keine hints besaß, mit solchen versehen werden, muß der eingefügte NSC geeignet formatiert werden. Dabei wird folgende Strategie verfolgt. Ist – was häufig der Fall ist – das Statement das erste oder einzige in einer Zeile, so wird der NSC in einer eigenen Zeile darüber eingefügt, wobei es genauso wie das Statement eingerückt wird. Anderenfalls wird der NSC direkt vor den Text des Statements plaziert.

```
Flag fits_text(Position *pos1,Position *dim1,Position *middle,LGtext *gtext,
               double factor,char *hyphen)
{
  Fits_status status=F_BEGIN;
  Flag        rc=YES;
  ...

  /* determine max fontsize (loop over list gtext) */
  maxsize= ...

  /* insert local and global hyphen points */
  while(lpt) {
    next_lpt = lpt->next;
    insert_hyphen_in_text(hyphen,lpt);
    lpt=next_lpt;
  }

  /* determine for each text the width and store it in dim */
  for(lpt=gtext; lpt; lpt=lpt->next) {
      font_fak = lpt->val->realfont->realsize / 10.0;
      if (lpt->val->dim->x > -100) {
          lpt->val->dim->x = 0.0;
          for (i=0;i<strlen(lpt->val->text);i++)
            lpt->val->dim->x += lpt->val->realfont->dim->width[(int)lpt->val->text[i]]
      * font_fak;
      }
  }
  dim1->y += options.line_fak*maxsize*F_TO_L*factor;
  pos = new_Position(0.0,options.line_fak*maxsize*(F_TO_L-BOT_FAK)*factor+pos1->y,0);

  if (middle)      /* its a triangle */
    /* sets pos to begin of line */
    line_width = search_line(pos1,dim1,middle,pos);
  else  {          /* its a box */
      line_width = dim1->x;
      pos->x = pos1->x;
  }

  lpt = gtext;
  font_fak = lpt->val->realfont->realsize / 10.0;
  do {
      switch (status)   {
        case F_BEGIN :
          status = F_FITS;
          width = line_width;
          break;
        case F_FITS :
          if (lpt)  {
              if (strcmp(lpt->val->text,"\\\\")==0) { /* carriage return from user */
                  lpt->val->dim->x = -100.0; /* don't print it */
                  lpt = lpt->next;
                  if (lpt) font_fak = lpt->val->realfont->realsize / 10.0;
                  status = F_CRLF;
              } else
                if (lpt->val->dim->x > -100) /* not tilde */
                  {
                    if (lpt->val->dim->x <= width)
                      {
                        lpt->val->pos->x = pos->x;
                        lpt->val->pos->y = pos->y;
                        DPRINTFF(("   pos = %f %f\n",lpt->val->pos->x,lpt->val->pos->y));
                        pos->x += lpt->val->dim->x;
                        width -= lpt->val->dim->x;
                        lpt = lpt->next;
                        if (lpt) font_fak = lpt->val->realfont->realsize / 10.0;
                        status = F_FITS;
                      }
                    else status = F_HARD;
                  } else status = F_TILDE;
            } else {
              rc = YES;
              status = F_END;
            }
          break;
        case F_TILDE:
        ...
    } while (status != F_END);

  return(rc);
}
```

Abbildung 4.8: generator: Routine für den Textumbruch

4.9 Modul Graphik-Editor

Der Graphik-Editor ist für die Verwaltung der in der Oberfläche integrierten Zeichenfläche zuständig. Neben der reinen Darstellung der Diagramme in dieser Zeichenfläche muß der Graphik-Editor dort auch die Interaktion mit dem Benutzer unterstützen.

Für das Zeichnen der Graphikobjekte bietet X11 eine Reihe von Funktionen an. Problematisch ist die Ausgabe von Texten, da dabei auf Ersatzfonts zurückgegriffen werden muß. Da die Anzeige aber in erster Linie für die Bearbeitung der Diagramme dienen soll, müssen hier keine aufwendige Verfahren implementiert werden.

Die Richtungen der Koordinatenachsen stimmen mit denen von Nassi überein. Der Ursprung des Koordinatensystems liegt links oben. Die Koordinaten werden in Punkten angeben, die der Auflösung des Bildschirms entsprechen. Die in *nassi* mit 1/72dpi veranschlagte Auflösung kann direkt übernommen werden.

```
double scale_factor[SCALESTEPS]
       = { 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.5, 3.0, 3.5, 4};

/* nassi -> X11 */
inline int  ytoxy (double y) {
  return (int) ( (int) y * draw_status.draw_scale)+YMOVE;
}
inline int  xtoxx (double x) {
  return (int) ((((int) x)+XMOVE)*draw_status.draw_scale);
}

/* X11 -> nassi */
inline double xxtox (int xx) {
  return (double) (((double) xx)/draw_status.draw_scale) - (double) XMOVE;
}
inline double xytoy (int xy) {
  return (double) ( (double) (xy-YMOVE))/draw_status.draw_scale;
}
```

Abbildung 4.9: draw: Funktionen zur Umrechnung der Koordinaten: Die ersten beiden Routinen konvertieren *nassi*-Koordinaten in X11-Koordinaten. Die nächsten beiden Routinen sind für die umgekehrte Richtung zuständig und werden bei der Auswertung von Maus-Event benötigt. Die beiden Makros XMOVE und YMOVE sorgen dafür, daß das Diagramm nicht direkt am Rand der Zeichenfläche erscheint. Die Variable draw_scale kann eine der in scale_factors definierte Vergrößerungsstufen enthalten. Die Rundung der Koordinaten, die für X11 notwendig ist, wird so spät wie möglich durchgeführt.

Für die Anzeige der Diagramme werden zwei X11-Bitmaps angelegt. Die erste beinhaltet den in in der Oberfläche sichtbaren Teil des Diagramms (Canvas), die zweite dient nicht zur Anzeige, sondern nur zur Speicherung der gesamten Graphik. Beide Bitmaps werden im Speicher des X-Servers angelegt. Jede Aktion, die die Größe des Diagramm verändert, muß auch die Größe der zweiten Bitmap anpassen. Dazu muß ggf. die vorherige gelöscht und die Bitmap mit der neuen Größe angelegt werden. Die wichtigsten Abschnitte der Routine `canvas_draw`, die sowohl beim ersten Zeichnen als auch bei solchen Größenänderungen aufgerufen wird, ist in Abbildung 4.10 auf der nächsten Seite dargestellt.

4.9.1 Interaktion mit dem Benutzer

Für die Interaktion mit dem Benutzer müssen Callback-Routinen definiert werden, die bei einer Aktion des Benutzers aufgerufen werden. Dazu gehören das Drücken einer Taste bzw. Maustaste und das Bewegen der Maus. Auch das Auswählen eines Menüpunktes in den Optionen-Menüs führt zur Ausführung einer Callback-Routine. Bei einigen dieser Routinen ist der aktuelle Zustand des Graphik-Editors wichtig. Dieser Zustand wird mit Hilfe von statischen Zustandvariablen gespeichert, die auch über mehrere Funktionsaufrufe ihren Wert behalten. Zur besseren Übersicht werden die Variablen in einer Struktur `draw_status` gespeichert (siehe Abbildung 4.11 auf der nächsten Seite).

```
    /* 1. get size of diagram  */
    draw_status.draw_pos1_diagram.x=xtoxx(graph->pos->x);
    draw_status.draw_pos1_diagram.y=ytoxy(graph->pos->y);
    draw_status.draw_pos2_diagram.x=xtoxx(graph->pos->x+graph->dim->x);
    draw_status.draw_pos2_diagram.y=ytoxy(graph->pos->y+graph->dim->y);

    /* 2. free old pixmap */
    XFreePixmap(fl_get_display(),area);

    /* 3. get size of canvas */
    fl_get_object_geometry(fmain->canvas,&x_win,&y_win,&w_win,&h_win);
    asize->w=MAX(draw_status.draw_pos2_diagram.x+RIGHTMARGIN,w_win);
    asize->h=MAX(draw_status.draw_pos2_diagram.y+BOTTOMMARGIN,h_win);

    /* 4. allocate new pixmap */
    area = XCreatePixmap (fl_get_display(), FL_ObjWin(fmain->canvas),
                          asize->w, asize->h, fl_get_visual_depth());

    /* 5. clear pixmap  */
    XSetForeground(fl_get_display(), canvasGC, fl_get_flcolor(BACKGROUND));
    XFillRectangle(fl_display, area, canvasGC, 0, 0, asize->w, asize->h);
    XSetForeground(fl_get_display(), canvasGC, fl_get_flcolor(BACKGROUND));

    /* 6. initialize sliders */
    fl_set_slider_bounds(fmain->vslider, 0, MAX((double) asize->h-h_win,0));
    fl_set_slider_size(fmain->vslider,MIN((double)h_win/asize->h,1.));
    fl_set_slider_bounds(fmain->hslider, 0, MAX((double) asize->w-w_win,0));
    fl_set_slider_size(fmain->hslider,MIN((double)w_win/asize->w,1.));

    /* 7. draw diagram */
    traverse_canvas(nxs,0);

    /* 8. copy visible part of diagram to canvas */
    yoff=(int) fl_get_slider_value(fmain->vslider);
    xoff=(int) fl_get_slider_value(fmain->hslider);
    XCopyArea (fl_get_display(), area, FL_ObjWin(fmain->canvas), canvasGC,
               xoff, yoff, w_win, h_win, 0, 0);
```

Abbildung 4.10: draw: Anlegen der Bitmap und Zeichnen des Diagramms: Da der Graphik-Editor in die Oberfläche eingebettet und diese mit *Forms* realisiert ist, werden einige Ressourcen über Routinen der Forms-Library bereitgestellt. Die Routinen beginnen mit `fl_`. Die Funktion `traverse_canvas` durchläuft die NXS-Struktur rekursiv und zeichnet dabei alle darin gespeicherte Graphikobjekte. Der letzte Schritt kopiert einen Teil der nicht sichtbaren Bitmap in das sichtbare Canvas. Diese Operation geschieht alleine im X-Server und ist daher sehr effizient.

```
struct draw_status_st {
  NXS      *actual_nxs_on_canvas;  NXS      *nxs_under_mouse;
  LNXS     *selected_nxs;          State_io stateinout;
  Flag     statefocus;
  double   last_button2_pos_x;   double   last_button2_pos_y;
  long     shift_pressed;        long     control_pressed;
  long     button2_pressed;      long     shiftbutton1_pressed;
  Position draw_pos1_diagram;    Position draw_pos2_diagram;
  double   focus_act_x1;         double   focus_act_y1;
  double   focus_act_x2;         double   focus_act_y2;
  double   focus_mark_x1;        double   focus_mark_y1;
  double   focus_mark_x2;        double   focus_mark_y2;
  long     colspec_if_active;    long     colspec_if_snapped;
  double   colspec_if_x;         double   colspec_if_y;
  double   colspec_if_boxsize;   double   colspec_if_mouse_x;
  double   colspec_if_mouse_y;   Position colspec_if_p1;
  Position  colspec_if_p2;       Position colspec_if_m;
  Flag set_lock_all_draw_state;  double   draw_scale;
  long     draw_scale_step;      LXfont_store  *lxfont_store;
  LFlcolor *flcolorshead;        FL_COLOR next_free_flcolor;
};
typedef struct draw_status_st Draw_status;
```

Abbildung 4.11: Statische Zustandsvariablen des Graphik-Editors: Die Variablen enthalten z.B. Informationen über das im Graphik-Editor dargestellte Diagramm, welche Strukturen daraus selektiert sind, über welcher Struktur sich die Maus momentan befindet und verschiedene Koordinaten, die für die Behandlung spezieller Aktionen benötigt werden.

Die Callback-Routinen werden über Befehle der Forms-Library, die auch für die Breitstellung des Canvas zuständig ist, an die Events gebunden (siehe Abb. 4.12). Die Events des Canvas werden an insgesamt fünf Callback-Routinen angebunden.

```
   fl_add_canvas_handler(fmain->canvas, Expose,        canvas_expose, asize);
   fl_add_canvas_handler(fmain->canvas, MotionNotify,  canvas_motion, NULL);
   fl_add_canvas_handler(fmain->canvas, KeyPress,      canvas_key,    asize);
   fl_add_canvas_handler(fmain->canvas, KeyRelease,    canvas_key,    asize);
   fl_add_canvas_handler(fmain->canvas, ButtonPress,   canvas_mouse,  asize);
   fl_add_canvas_handler(fmain->canvas, ButtonRelease, canvas_mouse,  asize);
```

Abbildung 4.12: Anbindung der Callback-Routinen an die Events des Graphik-Editors

Als Beispiel für eine einfach aufgebaute Callback-Routine wird hier diejenige aufgezeigt, die bei einer Veränderung der zum Canvas gehörenden Scrollbars aufgerufen wird (Abb. 4.13). Deren Anbindung geschieht bei der Definition der Scrollbars. Die neue Position der Scrollbars kann über die Routine `fl_get_slider_value` erfragt und direkt bei der Kopieren des neuen Ausschnitts in den sichtbaren Canvas-Bereich genutzt werden.

```
void canvas_scroll(FL_OBJECT *obj, long dummy)
{
  extern FD_nassi *fmain;  extern GC canvasGC;
  extern Pixmap   area;
  Window win;            /* Canvas                                          */
  FL_Coord x,y,w,h;      /* coordinates of the canvas-window                */
  int xoff,yoff;         /* left upper corner of visible part of the pixmap */

  win=FL_ObjWin(fmain->canvas);
  fl_get_object_geometry(fmain->canvas,&x,&y,&w,&h);
  yoff=(int) fl_get_slider_value(fmain->vslider);
  xoff=(int) fl_get_slider_value(fmain->hslider);
  XCopyArea (fl_get_display(), area, win, canvasGC, xoff, yoff, w, h, 0, 0);
}
```

Abbildung 4.13: Callback-Routine zu den Scrollbar-Events des Canvas

Eine etwas aufwendigeren Aufbau haben diejenigen Callback-Routinen, die an die Tasten der Maus oder Tastatur gebunden sind. Als Beispiel soll dazu die Callback-Routine zum Event `canvas_key` aufgeführt werden, die im wesentlichen die Bewegung des Diagramm innerhalb des Canvas über Tasten der Tastatur steuert (Abb. 4.14).

```
int canvas_key(FL_OBJECT *canvas, Window win, long w_win, long h_win,
                  XEvent *ev, void *a)
{   ...

    XLookupString(&(ev->xkey),buffer,bufsize,&keysym,&compose);

    if ((ev->type == KeyPress) && ( (keysym==XK_Shift_L) || (keysym==XK_Shift_R) ))
      draw_status.shift_pressed=YES;
    else draw_status.shift_pressed=NO;

    if ((ev->type == KeyPress) && ( (keysym==XK_Control_L) || (keysym==XK_Control_R) ))
      draw_status.control_pressed=YES;
    else draw_status.control_pressed=NO;

    if ((ev->type == KeyPress) &&  (keysym==XK_Prior) ) {
        xoff=(int) fl_get_slider_value(fmain->hslider);
        yoff=(int) fl_get_slider_value(fmain->vslider);
        yoff=MAX(0,yoff-(h_win/2));
        fl_set_slider_value(fmain->vslider,(double) yoff);
        XCopyArea (fl_get_display(), area, win, canvasGC, xoff, yoff, w_win, h_win, 0, 0);
        draw_status.statefocus=NO; /* last focus is destroyed */
        determine_focus(ev->xkey.x+xoff,ev->xkey.y+yoff);
    } else
    if ...
}
```

Abbildung 4.14: Callback-Routine zu den Tastatur-Events

Da die Routine beim Drücken einer beliebigen Taste angesprochen wird, muß dort als erstes die verursachende Taste gefunden werden. Dazu steht der X11-Befehl `XLookupString` zur Verfügung, die zu einem Event die eine Spezifikation der Taste liefert. Mit den ersten beiden Abfragen wird

untersucht, ob eine Modifier-Taste (Shift, Control) gedrückt ist. Die entsprechenden Statusvariablen werden aktualisiert und können auch von anderen Callback-Routinen ausgelesen werden.

Anschließend werden in einer Kaskade von If-Abfragen die belegten Tasten abgefragt. Als Beispiel ist hier die Taste `PageUp` aufgelistet, die eine ähnliche Wirkung wie die Bewegung des vertikalen Scrollbar hat. Das Diagramm wird um die halbe Fensterbreite nach oben geschoben. Zusätzlich zum Kopieren der Bitmap müssen hier auch Scrollbars angepaßt werden. Die letzten beiden Befehle setzten den Fokus (Markierung des aktuellen Statements) neu.

4.9.2 Markierung und Selektion von Strukturen

Als Beispiel für die Markierung und Selektion von Strukturen innerhalb des Diagramm wird die Routine `mark_select_rect` aufgeführt, die von den Callback-Routinen aufgerufen wird, wenn eine Teilstruktur selektiert wird. Der Parameter `select` bestimmt, ob die Struktur selektiert oder eine bestehende Selektion aufgehoben werden soll. Bei der Selektion werden in die vier Ecken des Nassi-Shneiderman-Elements kleine Boxen gezeichnet, deren Größe von dem Skalierungsfaktor des Diagramms abhängen. Der restliche Code wird benötigt, um den Sonderfall eines If-Statements abzuhandeln. Ein solches Statement erhält einen weiteren Markierungspunkt, der eine Verschiebung der Spaltenbreiten erlaubt. Die Position wird dazu in Statusvariablen gespeichert. Für das Löschen der Selektion wird der Teil des Diagramms einfach neu gezeichnet. Dies geschieht mit dem Aufruf der Routine `traverse_canvas`. Handelte sich bei dem Statement um eine If-Statement müssen auch noch die entsprechenden Statusvariablen zurückgesetzt werden. Die Aktionen dieser Routine beziehen sich auf die komplette Bitmap. Selektionen bleiben also auch beim Verschieben des Diagramms erhalten. Als letzte Aktion der Routine muß der sichbare Anteil des Diagramms wieder in das Canvas kopiert werden.

```
void mark_select_rectangle(double x1, double y1, double x2, double y2, OnOff select,
                           Window win, long w_win, long h_win, long mouse_x, long mouse_y,
                           NXS *nxs) {
  if(select==ON) { /* mark the nxs */
      bs=(int) (SELECTBS*sqrt(draw_status.draw_scale)); /* boxsize */
        XSetFunction(fl_get_display(),canvasGC,GXcopy);
        XSetForeground(fl_get_display(), canvasGC, fl_get_flcolor(SELECTCOL));
        XSetBackground(fl_get_display(), canvasGC, fl_get_flcolor(BACKGROUND));
        XFillRectangle(fl_get_display(),area,canvasGC,x1+1, y1+1, bs, bs);
        XFillRectangle(fl_get_display(),area,canvasGC,x1+1, y2-bs, bs, bs);
        XFillRectangle(fl_get_display(),area,canvasGC,x2-bs, y1, bs, bs);
        XFillRectangle(fl_get_display(),area,canvasGC,x2-bs, y2-bs, bs, bs);

        if ( (nxs->type==N_IF) &&    ...  )  {
          midx=xtoxx(nxs->body.sif->selse->graph->pos->x);
          midy=ytoxy(nxs->body.sif->selse->graph->pos->y);
          XSetForeground(fl_get_display(), canvasGC, fl_get_flcolor(COLSPECCOL));
          XSetBackground(fl_get_display(), canvasGC, fl_get_flcolor(BACKGROUND));
          XFillRectangle(fl_get_display(),area,canvasGC, midx-(bs/2), midy-bs, bs, bs);
          draw_status.colspec_if_active=YES;
          draw_status.colspec_if_x=midx-(bs/2);
          draw_status.colspec_if_y=midy-bs;
          draw_status.colspec_if_bs=bs;
        }
  } else {  /* unmark the nxs */
    traverse_canvas(nxs,100);
    if ((nxs->type==N_IF) && (draw_status.colspec_if_active==YES) ) {
      draw_status.colspec_if_active=NO; draw_status.colspec_if_x=0;
      draw_status.colspec_if_y=0;
    }
  }
  xoff=(int) fl_get_slider_value(fmain->hslider);
  yoff=(int) fl_get_slider_value(fmain->vslider);
  XCopyArea (fl_get_display(), area, win, canvasGC, xoff, yoff, w_win, h_win, 0, 0);
  draw_status.statefocus=NO; determine_focus(mouse_x,mouse_y);
}
```

Abbildung 4.15: Markierung einer selektierten Struktur

4.10 Modul Fontgenerator

Die Implementation des Moduls Fontgenerator beschränkt sich auf das Auslesen und Verwalten der im Entwurf beschriebenen Datenstrukturen. Listing 4.16 zeigt die nach außen hin sichtbare Schnittstelle zu den anderen Modulen.

```
#ifndef _FONTGEN_H
#define _FONTGEN_H

/* possible output drivers */
typedef enum {DRV_TGIF,DRV_EPS,DRV_PS,DRV_XFIG,
      DRV_JAVA, DRV_NONE} Output_driver;

/* warning in fontgen functions */
#define NE_FONTGENWARN (NE_FONTGEN + NE_WARN + 1)

/* reads font dimensions and mapping table from files and generate a
   structure pos_fonts for the Output_driver */
long fontgen(Output_driver);

/* Search a driver font for fontname, fontformat, fontsize and returns
   a pointer to an existing font structure.  */
Realfont *fontmap(char *fontname,char *fontformat, double fontsize);

/* read the font dimensions and mapping informations from ASCII files
   and writes this informations to binary files */
void fontcompile();
#endif
```

Abbildung 4.16: fontgen.h: Schnittstelle zum Fontgenerator

Die Funktion `fontmap()` ist die zentrale Routine des Fontgenerators. Sie durchsucht die aktuelle Zuordnungstabelle nach einer Schriftart, die zu den Parametern paßt und gibt einen Zeiger auf die entsprechende Realfont-Struktur zurück.

Das Einlesen der Daten dauert auf einer durchschnittlichen Maschine mehrere Sekunden. Die Dateien für die Dimensionen und der Zuordnungstabelle enthalten die Information in Klartext. Das Einlesen kann erheblich beschleunigt werden, wenn die Dateien im Binärform eingelesen werden können. Da das Binärformat aber je nach Plattform unterschiedlich sein kann, können die Binärdateien erst bei der Installation von `Nassi` generiert werden. Dazu bietet der Fontgenerator die Funktion `fontcompile()` an, die die ASCII-Dateien einliest und die Informationen im Binärformat wieder abspeichert. Über eine Kommandozeile-Option (`-fontcompile`) kann bei der Installation von Nassi die Aktion durchgeführt werden.

Die Strukturen zur Speicherung der Dimensionen sind in zwei Stufen gegliedert (Listing 4.17 auf der nächsten Seite). Es sollte auf jeden Fall sichergestellt sein, daß pro Schriftart die drei Vektoren `Fn_Height`, `Fn_Width` und `Fn_Depth` nur einmal im Speicher vorhanden sind. Daher sind die Vektoren in eine eigene Struktur verpackt. Die Struktur Realfont enthält nur einen Verweis auf diese. Bei der Implementation von *Nassi* wird davon ausgegangen, daß alle Schriftarten 256 Zeichen umfassen.

Als Schnittstelle sind die in in Listing 4.16 aufgezeigten Routinen und die in Listing 4.17 auf der nächsten Seite aufgeführten Datenstrukturen den anderen Modulen bekannt. Insbesondere wird die Realfont-Struktur zur Bildung der NXS benötigt.

```
#ifndef _REALFONT_H
#define _REALFONT_H

#define MAX_CHAR_NR 256

typedef double Fn_height[MAX_CHAR_NR];
typedef double Fn_width[MAX_CHAR_NR];
typedef double Fn_depth[MAX_CHAR_NR];

/* structure fontdim contains dimensions of all characters of font */
struct fontdim {
    Fn_height height;
    Fn_width  width;
    Fn_depth  depth;
    char      *realname;
};
typedef struct fontdim Fontdim;

/* structure realfont contains description of a existing font in the
   output format */
struct realfont {
    Fontdim   *dim;
    double    realsize;
    char      *realname;
    char      *mapname;
    char      *mapformat;
    void      *xfont;
};
typedef struct realfont Realfont;

#endif
```

Abbildung 4.17: realfont.h: Defintion der Struktur `Realfont`, die eine im Ausgabeformat vorhandene Schriftart beschreibt. In dieser Struktur werden auch die Dimensionen der einzelnen Zeichen der Schriftart gespeichert.

4.11 Modul Ausgabe

Das Ausgabe-Modul ist für die Generierung der Ausgabedateien verantwortlich. Dem Benutzer stehen dabei mehrere Formate zur Verfügung, die zum Teil eine unterschiedliche Behandlung erfordern. Neben PostScript als reines Druckformat und Encapsulated PostScript als Format für die Einbindung in Textverarbeitungssysteme sollten auch die internen Speicherformate von den Graphikeditoren Tgif und Xfig unterstützt werden.

Wie im Entwurf beschrieben handelt es sich bei allen drei Formaten (PostScript, Tgif und Xfig) um reine ASCII-Dateien. Die Dateien können also mit mit `fprintf`-Befehlen beschrieben werden.

Die in den Dateien zu speichernden Informationen sind komplett in der NXS enthalten und daraus in einem rekursivem Durchlauf entnommen und ausgegeben werden. Bei den Formaten der Graphikeditoren ist dabei die Reihenfolge wichtig. Diese legt auch die Reihenfolge der Elemente bezüglich Hinter- und Vordergrund fest.

Allen drei Formaten ist gemein, daß die Dateien aus einem Vorspann und einem Hauptteil bestehen. Im Vorspann werden Informationen über die Graphik abgelegt und auch Befehle definiert, die im Hauptteil benutzt werden.

Da die Abarbeitung der NXS in bei allen Formaten ähnlich erfolgt, sollte sie gekapselt werden. Dies erspart zum einen redundanten Code, ermöglicht aber auch eine einfache Hinzunahme eines neuen Formates.

Die Kapselung erfolgt über die in Listing 4.18 auf der nächsten Seite aufgeführte Struktur. Für jedes Ausgabeformat müssen die darin enthaltenen Variablen mit Zeigern auf Funktionen belegt werden, die dann die eigentliche Ausgabe der Diagramme durchführen. Die Funktionen erhalten

```
/* contains pointers to drawing routines for the choosen driver */
struct draw_func_t {
    void (*polygon)( int anz, double x[], double y[],Color *col, Color *fcol);
    void (*polyline)( int anz, double x[], double y[],Color *col, Color *fcol);
    void (*box)(Position *pos, Position *dim, Color *col, Color *fcol);
    void (*text)(Position *pos, char *text, Realfont *font, Color *col);
    void (*line)(Position *p1, Position *p2, Color *col);

    void (*collecttext)(Position *pos, char *text, Realfont *font, Color *col);
    void (*collectfunctionname)(char *name, double wy);
    void (*preamble)();

    int  (*start)(char* fnam, double x, double y, double wx, double wy);
    void (*end)();
    void (*start_diagram)(char* fnam, double x, double y, double wx, double wy);
    Flag (*end_diagram)();
};
typedef struct draw_func_t Draw_func;
```

Abbildung 4.18: Ausgabe: Interne Struktur des Ausgabe-Moduls, die Zeiger auf Funktionen enthält, die während des Durchlaufs durch die NXS ausgeführt werden.

die benötigten Informationen als Parameter. Bei der Initialisierung oder bei einer Änderung des Ausgabeformats muß nur noch die Struktur mit den entsprechenden Zeigern belegt werden.

Bei einen Aufruf der `output`-Routine werden die dort aufgeführten Funktionen in der folgenden Art ausgeführt:

1. Zu Beginn der Ausgabe wird die Routine `start()` aufgerufen. Diese sollte die Ausgabedatei öffnen und den statischen Vorspann der Ausgabe, wie z.B. neue Befehle im PostScript und Einbettung der Umlaute ausgeben.
2. Um Informationen über benötigte Fonts und Graphikobjekte erfassen zu können, wird der Teilbaum der NXS-Struktur erstmalig zur Datenerfassung durchlaufen. Dabei werden die Routinen `collecttext()` und `collectfunctionname()` aufgerufen.
3. Erst dann wird die Funktion `preamble()` aufgerufen, die einem vom Inhalt der Graphik abhängigen Vorspann erzeugen kann.
4. Bei dem Ausgabeformat PostScript, das mehrere Diagramme auf getrennten Seiten enthalten kann, wird vor jedem Diagramm die Routine `start_diagram` und nach dem Diagramm `end_diagram` aufgerufen. Mit diesen Routinen kann die Seitensteuerung vorgenommen werden.
5. Zur eigentlichen Ausgabe des Diagramms wird der Teilbaums der NXS wird zweites Mal durchlaufen und dabei für jedes Graphikobjekt eine der ersten fünf Routinen aus Listing 4.18 aufgerufen.
6. Zum Schluß der Bearbeitung wird die Routine `end()` aufgerufen, die z.B. die Ausgabedatei schließt.

Mit dieser Zwischenebene wird die interne Behandlung der NXS-Struktur von der eigentlichen Ausgabe getrennt. Bei der Hinzunahme eines weiteren Ausgabeformats müssen nur die oben aufgeführten Funktionen definiert werden. Dazu muß die Beschaffenheit der NXS-Struktur nicht bekannt sein.

Ausgabeformat Tgif

Für die Generierung von Tgif-Dateien steht eine C-Library zur Verfügung (`tgifctrl` [8]), die dazu eine Sammlung von Funktion bereitstellt. Die in Listing 4.18 aufgeführten Routinen konnten direkt mit den Routinen aus der `tgifctrl`-Library realisiert werden. [1]

[1] Da die Library noch nicht an die aktuelle Version 3.1 von Tgif angepaßt wurde, mußte dies bei die Implementation von Nassi durchgeführt werden. Dies gestaltete sich problematisch, da keine Beschreibung des Tgif-Format existiert. Die

```
int draw_start_tgif(char* fnam, double x, double y, double wx, double wy) {
  f = TRopen(fnam);
  if (f==NULL) {
    ne_reset();
    ne_error(NE_OUTPUTERROR,"Cannot open output file '%s'\n",fnam);
    return NE_OUTPUTERROR;
  }
  TRset_htrans(f,0,210, 0,MAXXPOS_PS);
  TRset_vtrans(f,10,307, 0,MAXYPOS_PS);
  TRset_closed_fill(f,FIC_BLACK,FIP_NONE,FIP_SOLID);
  TRset_unlock(f);
  return 0;
}
```

Abbildung 4.19: Ausgabe: Start-Routine für die Ausgabe im Tgif-Format: Da sogar Befehle für die Transformation der Koordinaten zur Verfügung stehen, muß während des Zeichnen keine Umrechnung der Koordinaten vorgenommen werden. Einen Verweis auf die geöffnete Ausgabedatei muß bei allen Funktionsaufrufen mit angegeben werden.

Die Diagramme sind in ihrer Länge nicht beschränkt. Auch bei Tgif ist dafür keine Einschränkung vorhanden. Daher ist eine Kontrolle der Ausgabegröße nicht notwendig. Trotzdem muß bei der Definition des Zeichenfeldes eine Maximalgröße angegeben werden. Dies ist wichtig, um bei der Ausgabe mehrerer Diagramme die gleichen Größenverhältnisse zu erhalten.

Ausgabeformat Xfig

Für den Graphikeditor Xfig existiert zwar keine Library zur Erzeugung der Dateien. Gegenüber Tgif wird mit dem Editor Xfig eine komplette und aktuelle Beschreibung des Dateiformats geliefert. Mit dieser konnten die Graphikelemente von Nassi direkt auf die Graphikelemente des Graphikeditors abgebildet werden.

Die Auflösung im Xfig-Format ist 1200 dpi, so daß eine Umrechnung der Koordinaten (72 dpi in Nassi) bei der Ausgabe nötig ist. Diese wird durch die beiden Routine `xtofx()` und `ytofy()` erledigt.

```
void draw_line_xfig(Position *p1, Position *p2, Color *color) {
  fprintf(out_xfig_file,
  "2 1 0 1 %ld -1 100 0 20 0 0 0 0 0 0 2 \n",
  xfig_colormap(color->name));
  fprintf(out_xfig_file,
  "\t %ld %ld %ld %ld \n",
  xtoxfx(p1->x), ytoxfy(p1->y),
  xtoxfx(p2->x), ytoxfy(p2->y));
}
```

Abbildung 4.20: Ausgabe: Zeichenroutine (Linie) für die Ausgabe im Xfig-Format.

Ausgabeformat PostScript

Für die beiden Ausgabeformate PostScript und Encapsulated PostScript können für die Ausgabe der Graphikobjekte die gleichen Routinen benutzt werden. Die Formate unterscheiden sich im wesentlichen nur beim Vor- und Nachspann. Beim Format EPS müssen die Graphiken mit den EPSF-Kommentaren (z.B. Bounding-Box) versehen werden. Bei der PostScript-Ausgabe muß zusätzlich die Formatierung der PostScript-Seite eingebaut werden.

Standardmäßig sind in den Fonts von PostScript keine Umlaute enthalten. Für eine solche Erweiterung der Fonts müssen diese im Preamble des PostScript-Codes neu definiert werden. Die Zeichen selbst sind definiert, nur die Code-Page muß an den entsprechenden Stellen mit der Zuordnung

einzelnen Parameter mußten durch Testgraphiken oder durch Untersuchungen des Source-Codes bestimmt werden.

der Umlaute versehen werden. Dazu dient der in Listing 4.21 aufgeführte PostScript-Code. Bei der Ausgabe müssen dazu in einem ersten Durchlauf alle benötigten PostScript-Fonts gesammelt und in Vorspann des PostScript-Codes definiert werden. Dazu dient die Funktion `collecttext()`.

```
nassireencsmalldict 12 dict def
nassiReEncodeSmall
{ nassireencsmalldict begin
     /newcodesandnames exch def
     /newfontname exch def
     /basefontname exch def
     /basefontdict basefontname findfont def
     /newfont basefontdict maxlength dict def
     basefontdict
     { exch dup /FID ne
        { dup /Encoding eq
           { exch dup length array copy newfont 3 1 roll put }
           { exch newfont 3 1 roll put } ifelse
        } { pop pop } ifelse
     } forall
     newfont /FontName newfontname put
     newcodesandnames aload pop
     newcodesandnames length 2 idiv
     { newfont /Encoding get 3 1 roll put} repeat
     newfontname newfont definefont pop
  end
} def
nassi-font-vec [ 8#304 /Adieresis  8#326 /Odieresis 8#334 /Udieresis
         8#337 /germandbls 8#344 /adieresis 8#366 /odieresis
                  8#374 /udieresis ] def
% Beispiel
/Times-BoldItalic /Times-BoldItalic-8 nassi-font-vec nassiReEncodeSmall
/Times-BoldItalic-8 findfont 6 scalefont setfont
```

Abbildung 4.21: Ausgabe: Definition der Umlaute in den PostScript-Fonts: Der obere Teil definiert eine PostScript-Funktion, die einen Font kopiert und in der Kopie die Umlaute einfügt. Der neue Font kann erhält den Namen des Originals, der mit der Endung `-8` versehen wird. Die unteren beiden Zeile zeigen wie ein neuer Font definiert wird.

Bei der Druckausgabe mit PostScript ist die Ausgabefläche durch die Papiergröße beschränkt. Bei größeren Diagrammen stehen zwei Möglichkeiten zur Verfügung. Bei der ersten wird das Diagramm solange in der Größe verkleinert, bis das es auf die Seite paßt. Bei der zweiten wird das Diagramm auf mehrere Seiten verteilt, wobei das Diagramm abgeschnitten wird (siehe 4.22).

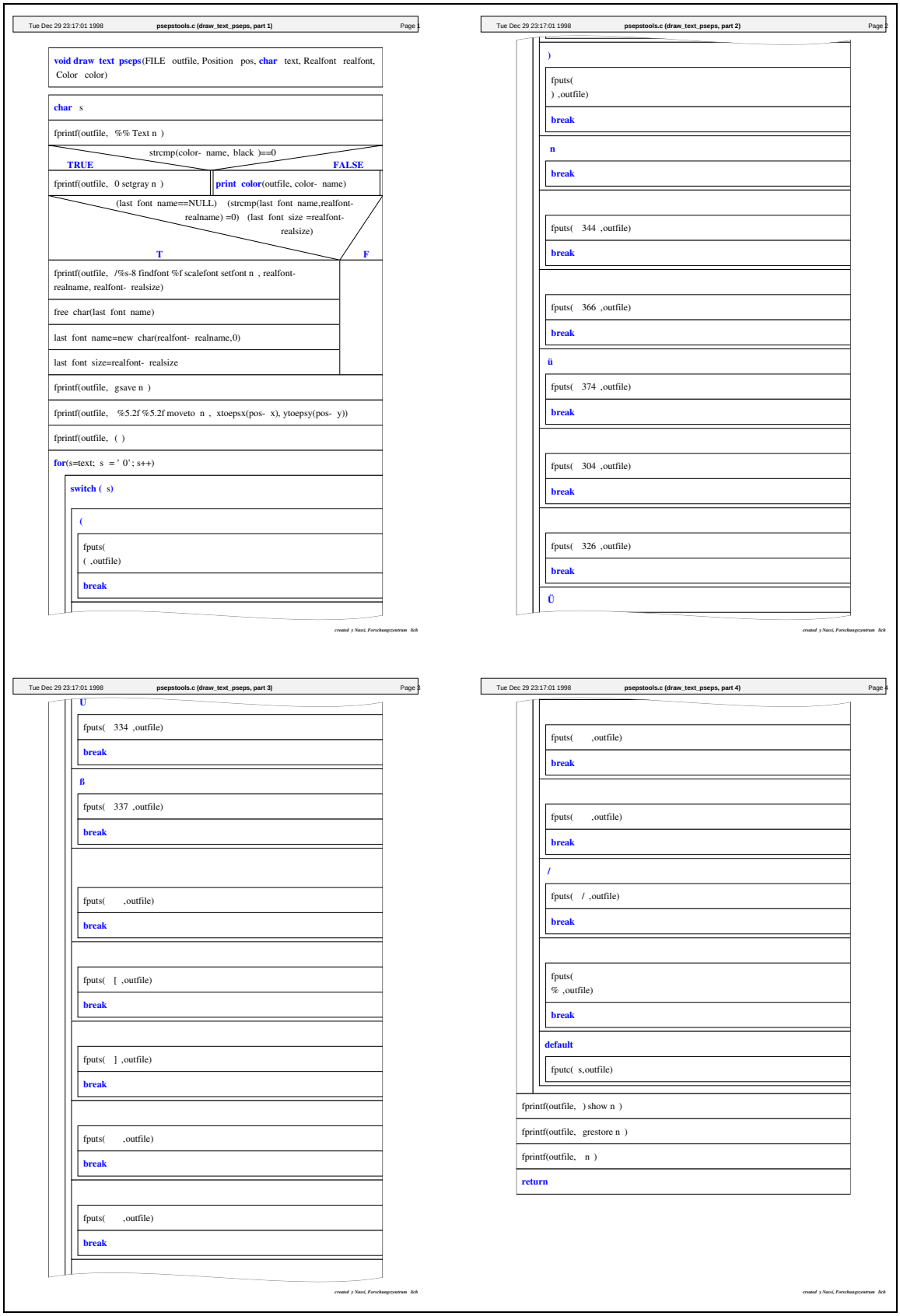


Abbildung 4.22: Beispiel für die Ausgabe eines langen Diagramms in PostScript: Die vier Seiten der Ausgabe sind hier in einem Diagramm zusammengefaßt. Die Teildiagramme werden mit Hilfe der in PostScript vorhandenen Clip-Linienzüge am oberen bzw. unteren Ende abgeschnitten. In der Header-Zeile sind Informationen zum Diagramm und die laufende Seitennummer enthalten.

Kapitel 5

Benutzung

Generation of Nassi-Shneiderman Diagrams under Unix with *nassi*

Thomas Eickermann, Wolfgang Frings, Anke Häming, Birgit Reuter
14 May 1997, TKI-0305

5.1 General

Nassi-Shneiderman diagrams serve to synoptically represent algorithms and program sequences by creating a graph of the elements of structured programming (loops and multi-way decisions) which can give a better overview in the case of complicated algorithms. The representation of algorithms using Nassi-Shneiderman diagrams is rather needed for small-scale programming, where it is a good alternative to flow diagrams.

The *nassi* program developed at ZAM serves to generate such Nassi-Shneiderman diagrams under Unix. *nassi* can be used to convert programs and algorithms available in C pseudocode or in C or PASCAL language into a graph. In this first version, *nassi* only supports a subset of the C programming language. A restriction is that the C parser of *nassi* does not recognize any definitions of variables within functions and is thus unable to exclude these from the diagram. However, since in most cases the diagrams are used in documentations where C pseudocode should be used, the restriction to the analysis of C pseudocode is not particularly relevant. In generating the graph, each function in the input is converted into a separate diagram.

For representation and postprocessing *nassi* provides a convenient interface with which single diagrams can be selected, represented, issued in different output formats and printed. The interface permits all menu-variable global options to be stored in profiles or directly in the source.

A graphics editor allows layout changes of the whole diagram or of single statements or control structures. If diagrams are too crowded, statements and whole structures can be hidden or shifted as a block to a separate diagram. Such changes to the layout and structure of diagrams can be stored again in the source code via special comments and are then directly available for further treatment of the source code.

For output purposes, *nassi* also provides the option of generating source data of the Tgif and Xfig graphics editors in addition to screen output, Encapsulated PostScript graphics and printable PostScript files. This makes it possible to also change and extend diagrams far beyond the functionality of the built-in graphics editor. For the rapid generation of diagrams *nassi* provides a batch option which generates diagrams in the desired output format independent of the interface.

5.2 Interface

5.2.1 Call

The program is called with
`nassi [options] [source file]`
In the current version, C and PASCAL source files can be specified. The options are described under 5.2.7 on the facing page.

5.2.2 Usage in batch mode

The call
`nassi -batch [options] source file`
causes *nassi* to be executed in batch mode. The diagrams are generated and issued without any further user inputs. The desired output format can be specified in a profile (5.7.1 on page 100).

5.2.3 Usage under X

After calling nassi, a window appears showing several menus, a subroutine browser displaying the subroutines of a source file, a graphics editor and a status browser.

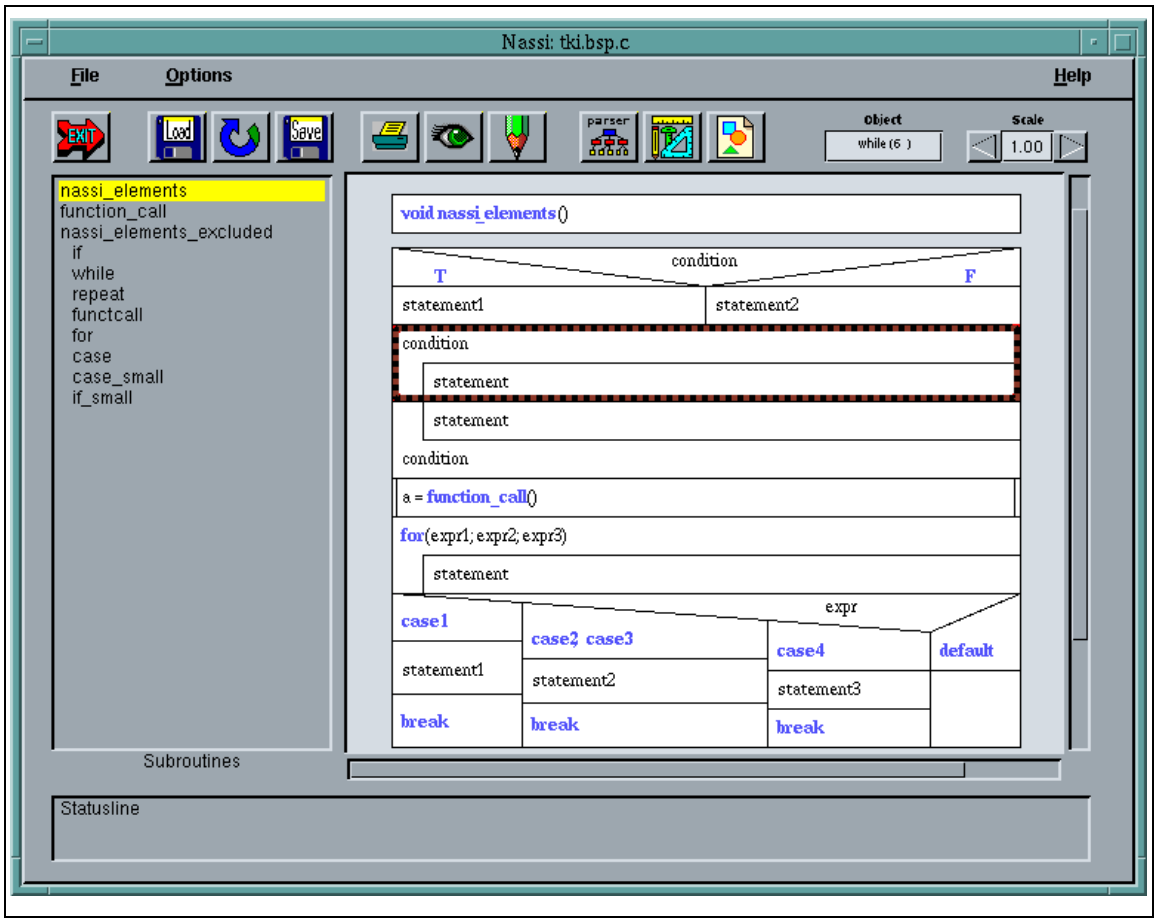


Figure 5.1: Main window of *nassi*

File menu

Load Source	loads a source file that can be specified in a file browser.
Reload Source	loads the current source code again.

Save Source	saves the source code with the options set by the user, which are filed in the form of comments in the source code. Optionally, all options or only those set for single statements can be stored.
Save Profile	optionally saves the options changed by the user in `$HOME/.nassirc`, `./nassirc` or in a file to be specified. Options that reside in `./nassirc` or `$HOME/.nassirc` are automatically loaded upon call, searching first in the current directory.
Load Profile	loads a profile that can be specified in a file browser.
Print	enables output of the graph on a PostScript printer or to a file.
Preview	displays all marked subroutines with ghostview.
Generate Output	generates an output file corresponding to the output options or, if necessary, several files.
Exit	terminates nassi.

Options menu

This menu serves to specify options for the parser, generator and output modules. The menu items are described in section 5.3 on the following page.

Help menu

This menu contains the following help texts: *About*, *Introduction* (TKI-0305), *Changes* and *Problems*.

5.2.4 Button bar

The most common menu items are provided underneath the menu bar in the form of icons that can be activated by clicking on them with any of the mouse buttons. The button row is divided into four groups:

1. *Exit* terminates nassi without any further question
2. *Load, Reload, Save* for loading and saving source code
3. *Print, Preview, Generate* for printing, viewing and generating output files
4. *Parser, Generator, Output options* for setting the global options

5.2.5 Subroutine browser

After loading a source file, all subroutines are listed in the subroutine browser. Subroutines must be selected for displaying a diagram in the graphics editor and for generating output. A subroutine is selected by clicking with the left mouse button (**M1**) on the corresponding line in the subroutine browser. In order to select several subroutines, keep the shift key pressed (**S-M1**) while clicking on the subroutines. In this case, the graphics editor only displays the subroutine selected first.

5.2.6 Status browser

The status browser serves to record all actions, warnings and error messages. The number of lines of the browser can be increased through the option `-statuslines`.

5.2.7 Options and X resources

The following options can be specified when calling the program:

-batch	Program is executed in batch mode
-statuslines *n*	Number of lines of the status browser
-rec_file *file*	If this option was specified, the position and size of the windows are stored in the specified file for the next call. The file name specified is preceded by the path of the user's HOME directory.
-profile *string*	Name of the profile to be loaded
-fontsize *n*	Character size in pixels
-width *n*	Width of the window
-height *n*	Height of the window
-geometry *string*	Size and position of the window

Options can also be specified in the form of X resources in addition to the command line. The corresponding lines of the resources file are built up as follows:

```
nassi*option:  value
```

Logic values as in the `-batch` option are specified as *true* or *false*.

The following additional information may be given in the X resources for storing the geometry data or changing the colors:

Background	background color
BorderColor1	border color (left and upper border)
BorderColor2	border color (right and lower border)
InputColor	color of the input fields
CanvasColor	color of the canvas

An example of the specification of X resources can be found under
`/usr/local/nassi/KFAappdefaults/Nassi.ad`.

5.3 Options menus

5.3.1 Options of the parser

The menu for the parser options is activated either via the options menu or directly by the parser button. At first, a window appears in which a parser can be selected. In the current version, parsers are available for C pseudocode and PASCAL. In addition, it is possible to have the parser automatically selected as a function of the file name. This is done by the option "Autoselect Language" which is active by default. A submenu exists for each parser, in which options can be set that are specific to the respective parser. It is only possible to branch into the menu of the currently active parser. For each parser, a search pattern ("File-pattern") can be specified in this menu, which is used for automatic parser selection. The PASCAL parser additionally has the option "hide empty statements". It defines whether empty statements at the end of a block are to appear in the diagram. Such empty statements are the consequence of a ';' after the last statement in a block. This option is active by default.

5.3.2 Generator options

When the button for the options of the generator is pressed, a window appears in which the type of generator can be selected. In the current version, however, there are only Nassi- Shneiderman diagrams available for selection. Further options can be changed in the next window that appears when pressing the button with the text "Nassi-Shneiderman diagram" (see Fig. 5.2).

The first three options concern the distance of the text from the border of the respective structure element. The values for these distances are specified in points, 1 point corresponding to approx.

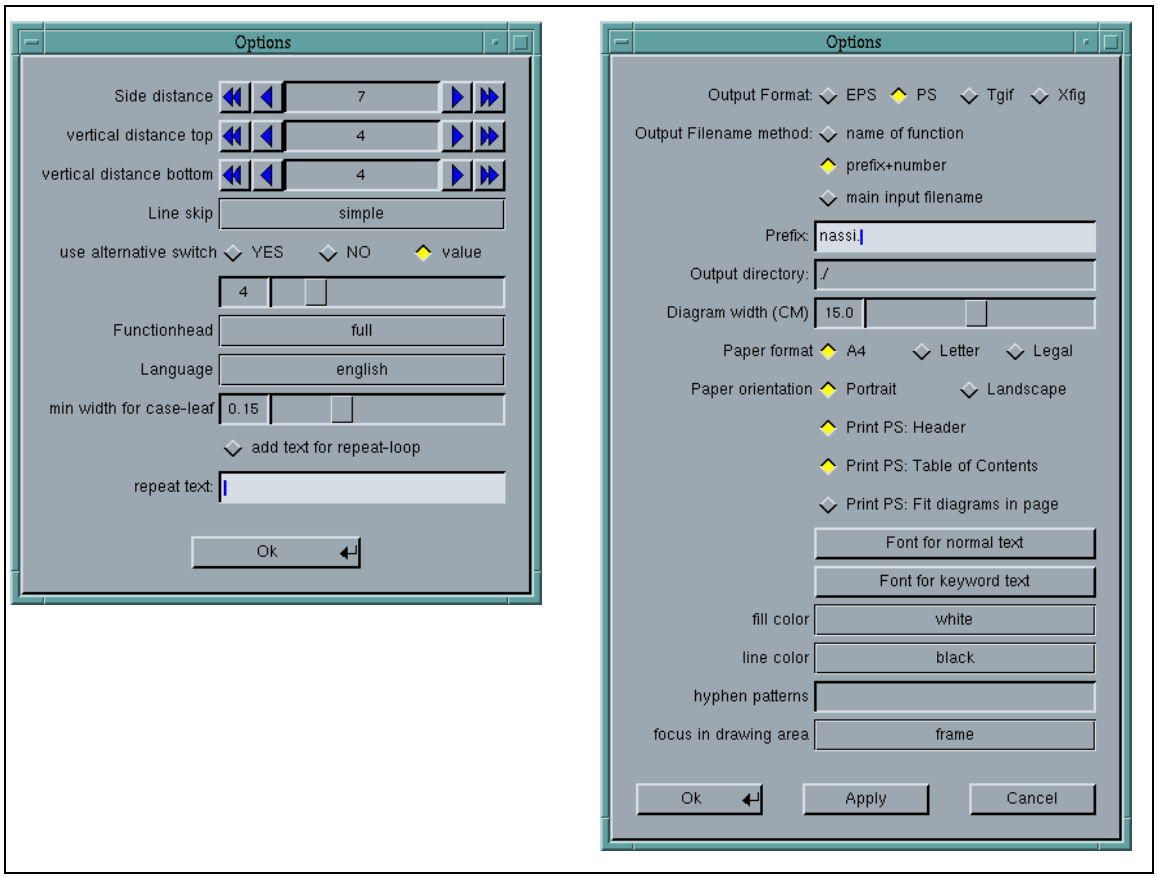


Figure 5.2: Options for the generator (left) and output (right)

0.3 mm. The values can be increased or lowered by 3 / 1 point using the double / single arrows, respectively.

The option *Line skip* switches between single and double line spacing.

The next option *use alternative switch* defines the limit at which the normal representation of a multi-way decision switches to the alternative representation.

The *function head* can either not be represented at all (*none*), or in a simplified form (*small*) or completely (*full*). In the simplified version, only the function name is plotted, whereas in the complete version all the parameters of the function also appear.

Language selection only relates to the words `TRUE` and `FALSE` in the ”if” head.

The next option relates to the minimum width of a case leaf and specifies a percentage value for the total width of the respective case structure to be plotted. The last two options provide the possibility of replacing the keyword `WHILE` by any arbitrary text in repeat loops.

The default values are as follows:

Side Distance	7	Functionhead	full
vertical distance top	7	Language	english
vertical distance bottom	7	min width for case leaf	0.15
Line skip	simple		

5.3.3 Options for the output

The third menu contains options which concern the output of the diagrams (see Fig. 5.2).

Four output formats are currently supported and can be selected in the top menu item.

EPS For each diagram, a separate file is produced containing Encapsulated PostScript code. These output files can be used for incorporating the diagrams into other word processing systems.

PS A PostScript file of all diagrams selected is produced and can be directly printed. In this case, one diagram is generated per page and can be reduced in size, if necessary, so that it fits on one DIN A4 page. The pages are provided with a current header featuring the page number, name of the input file, date of generation and name of the current function.

Tgif A separate *obj* file is produced for each diagram and can be loaded and further processed with the *Tgif* graphics editor. The representation of the diagram within *Tgif* varies slightly from the print version (PS) since alternate fonts are used again in *Tgif*.

Xfig A separate *fig* file is produced for each diagram and can be loaded and further processed with the *Xfig* graphics editor.

There are again three possibilities for generating the output file name:

name of function With the *EPS* and *Tgif* output formats, the name of the plotted function or of the exclude block is used for the individual files. Blanks are replaced by an underscore (_).

prefix+number The prefix specified in the next input field with an appended serial number is used for the name of the output file. The serial number is re-initialized with 1 for each output call.

main input filename The name of the input file (without extension), to which a serial number is appended again, is used for the name of the output file. The serial number is re-initialized with 1 for each output call.

The input field *Prefix* is given the name which, provided with a serial number, is used for generating the file name using the method *prefix+number*.

The next input field *Output directory* describes the directory in which the output files are stored. In this field, both a relative (e.g. `./out`) and an absolute path (e.g. `/tmp/out`) can be specified.

The option *Diagram width (CM)* describes the desired width of the diagrams in centimetres. The height of the diagrams depends on the contents of the functions.

The next option *Paper format* allows to specify a paper format for output. The selection of the german DIN format *A4* and the formats *letter* and *legal* is possible. The following option *Paper orientation* switches between the upright *portrait* and horizontal *landscape* orientation.

The next three options are only valid, if the output format *PS* is selected.

The first one (*Print PS: Header*) controls the printing of a header on each page which includes the input file name, the function name, the page number, and the current date.

The second one *Print PS: Table of contents* causes *nassi* to insert a table of contents at the beginning of the output.

The third option *Print PS: Fit diagram in page* switches between the scaling and the splitting of a diagram. The first choice scales the diagram down unless it fits on the output page. In the other case the diagram will be split to more pages without any scaling.

The next two buttons (*Font for normal/keyword text*) open new submenus in which the font name, font format, font size and font color can be set.

The attributes *fill color* and *line color* adjust the background color and the line color of the diagrams.

The input field *hyphen patterns* allows the specification of hyphenation proposals to be used for all the texts in the diagrams. The words can be set either directly or via the local options menu of the graphics editor using the switch *save hyphen in global hyphens*.

The last option *focus in drawing area* does not describe an option for the output, but the way of marking the structure element currently focused with the mouse in the graphics editor.

All the attributes of this menu have a global effect, i.e. they apply to all diagrams and can only be overwritten by the local attributes for single statements.

The default values for the individual attributes are as follows:

Output Format	EPS	width of nassi	15.0 cm
Output Filename method	prefix	max height of nassi	25.0 cm
Prefix	`./nassi`	Print PS header	yes
Output directory	`./`		
normal font name	Times	normal font size	11
normal font format	roman	normal font color	black
keyword font name	Times	keyword font size	11
keyword font format	bold	keyword font color	black
fill color	white	line color	black
hyphen patterns		focus in drawing area	frame

5.4 Graphics Editor

In the right-hand part of the *nassi* window there is an area serving to display and process the diagrams generated. This area always shows the diagram selected in the list of functions. If several diagrams are selected there, only the first diagram is shown.

5.4.1 Display

Since not all of the fonts are available under X11, alternate fonts may be used in displaying the diagrams, which can have a different size and form. If a font is not available in the desired size, the next smallest font of the same type is displayed. If the character font is not available, Courier is taken as an alternative. The editor only shows the coarse structure of the diagram with respect to fonts, a diagram with the proper fonts and font sizes is obtained using the preview button.

The scroll bars at the right and bottom of the canvas can be used to displace the diagram. Moreover, the keys **PageUp** and **PageDown** and the **Cursor** keys are assigned for displacing the diagram.

The diagram can be directly displaced in the canvas by pressing the middle mouse button (**M2**) and keeping it pressed; every movement of the mouse then shifts the diagram accordingly.

The fields **Object** and **Scale** at the right above the canvas show the type of structure on which the mouse is pointing and the current scaling factor for representation.

Scaling can be varied in a range from 0.25 to 4 using the two buttons beside the field and also by the key combinations **C-M1** and **C-M3** (mouse buttons). This scaling has only an effect on the display in the canvas and no significance for the output.

5.4.2 Local changes to structures

A local options menu is available for changing the layout and structure for statements or whole control structures of the diagram.

In order to change one or more statements, these must first be selected.

Structures can be selected using the left mouse button (**M1**), and the selected structures are then provided with a (red) mark at the corner points. The selection of the structure is cancelled by pressing this button again. The left mouse button can only be used to select one structure. The selection of another structure cancels all previous selections.

In contrast, additional structures can be selected with the key combination Shift left mouse button (**S-M1**). If the current structure has not been selected up to now, it is added to the set of selected structures, otherwise it is removed again from the set.

The right mouse button can now be used to call various submenus via a small PopUp menu. If only one structure was selected, these submenus show its attributes. Attributes which so far have not been changed in this structure are initialized with *inherit* in the menus.

The submenus do not have to be closed after each change. If a change is acknowledged with the **Apply** button instead of the **ok** button, the window remains open. The open windows are updated when selecting other structures.

The values of the individual inputs can either be chosen from a selection list using the left mouse button (**M1**) or be increased or reduced using the middle (**M2**) or right mouse button (**M3**).

The menus *structure*, *font / color* and *layout / misc* are available.

structure menu

This menu (Fig. 5.3) contains options which change the structural representation of the diagrams. This makes it possible to exclude structures from individual diagrams (exclude) or hide them (hide).

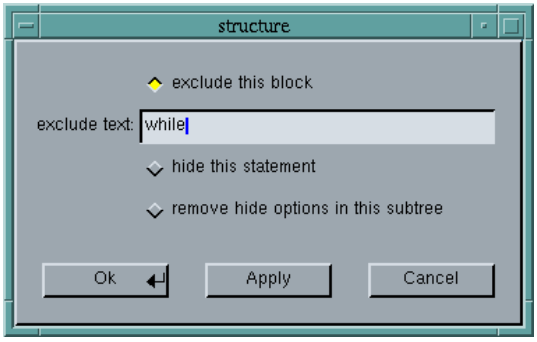


Figure 5.3: Local *structure* options menu in the graphics editor

Exclude blocks

The upper two options serve to administer Exclude blocks which are helpful in the case of diagrams being too long or too crowded. It is thus possible to remove individual statements or structures from this diagram and put them in a new separate diagram. The main diagram will then only show the scheme of a simple statement with the text `Block: exclude text`. The excluded block then appears as an additional input in the list of functions and can be further treated like a normal function. The block will be given the same heading in the lower-level diagram as in the higher-level one (`Block: exclude text`).

Such an exclusion can only be cancelled again in the local options menu of the higher-level diagram, since the attributes concerning Exclude and Hide are blocked for the function or block statement.

Hiding statements

Individual statements or structures can be hidden with the switch *hide this statement*. They will then not appear any more in the visible display of the diagram. However, since they are then no longer visible in the graphics editor either, this attribute cannot be cancelled from the editor any more. The hiding of structures can only be cancelled in the higher-level structures using the switch *remove hide options in this subtree*.

fonts / colors menu

This menu (Fig. 5.4 on the next page) can be used to change the fonts and colors of texts.

A distinction is made between fonts and colors for normal text and those for keywords. The options of font name, font format, font size and color are available for both of them.

The last two options determine the fill and frame color of the structure.

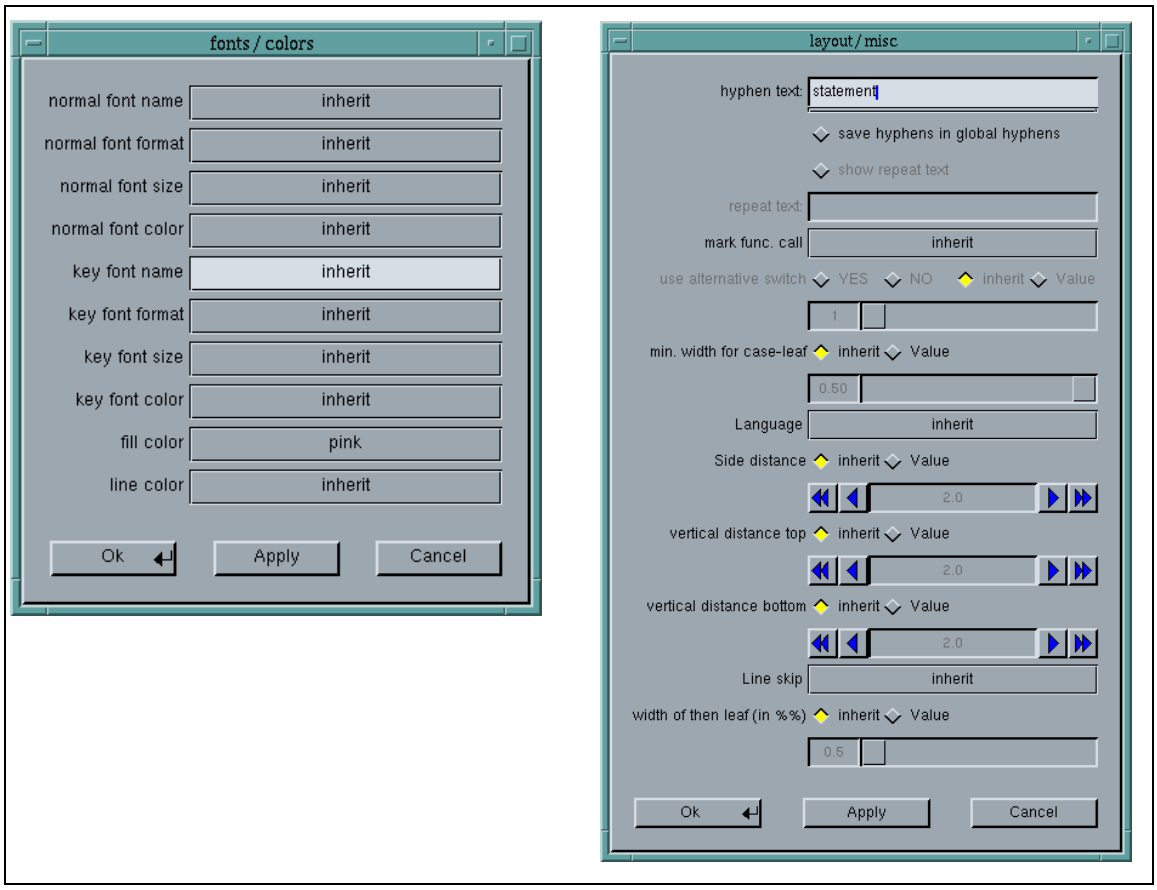


Figure 5.4: Local *layout / misc (fonts / colors)* options menu in the graphics editor

layout / misc menu

This menu (Fig. 5.4) provides options determining the layout and display of the individual structures.

The first two options can be used to insert breaks in the existing text of a statement (see also chapter 5.5 on the next page).

The next two options determine the representation of the keyword `WHILE` for repeat loops. The option *mark func. call* can decide on whether the structure is to be marked with lateral double lines as a function call. The value *NO_inherit* specifies that neither this nor the structures contained therein are to be marked as a function. The values *auto* and *auto_inherit* turn on the automatic recognition of a function call for this structure (where applicable, with the structures contained therein).

The next option *use alternative switch* defines the limit at which the normal representation of a multi-way decision switches to the alternative representation.

The option *min. width for case-leaf* serves to set a minimum column width for a case leaf.

The option *Language* describes the representation of the keywords `TRUE` and `FALSE` in an *if* structure. The next three options describe the distances of text from the border of the structure. The last option can be used to change line spacing.

Changing the layout of an if-statement

If an if-statement is selected, *nassi* displays an additional marker (in green) at the top of the vertical line between the *then* and *else* leaf of the if-statement. This button can be dragged to a new position. While dragging you see the new triangle for the if-condition. After releasing the mouse button the

new diagram layout will be computed and displayed. The hint `COLSPEC_IF` describes the layout of an if-statement in the source code. The value of this hint is the portion of the horizontal width reserved for the *then* leaf.

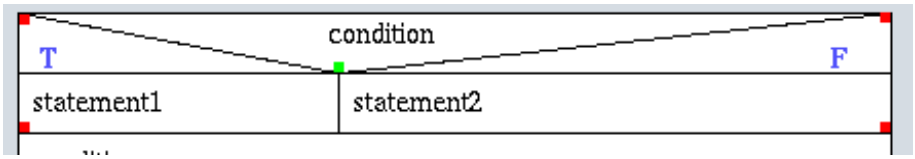


Figure 5.5: Changing the layout of an if-statement: *the green button rearranges the layout of the if-statement*

Navigation through the call tree of a program

Statements marked as function call with a double line are selectable with a double click of the left mouse button (**Double-M1**). After selecting *nassi* switches to the diagram of the function called in this statement. Additionally you can select block exclude statements in the same way.

The function displayed before this migration is stored in an internal stack. You can go back to the last one with a double click on the whole diagram (e.g. on the head of the diagram). So you can move up and down through the call tree of the input program. A selection in the subroutine browser flushes the internal stack.

In the current version of *nassi* the same functionality is assigned to the **Return** (Enter) key.

Key assignment within the graphics editor

Key	Description
PageUp	moves diagram downwards by half a page
PageDown	moves diagram upwards by half a page
Cursor $\leftarrow, \rightarrow, \uparrow, \downarrow$	moves diagram by one unit in the corresponding direction
left mouse button (M1)	selects / deselects structure
Shift-M1	selects further structure and deselects previously selected structure
middle mouse button (M2)	shifts diagram according to mouse movement
right mouse button (M3)	calls local options menu
Control-M1	reduces diagram size by one increment
Control-M3	enlarges diagram by one increment
Double-M1, Return	goes to the function/block called in this statement

5.5 Hyphenation of Text

The text of a statement often does not fit in one line so that proper hyphenation must be performed. In principle, lines are only broken between two tokens of the programming language or in the case of blanks within strings.

The user can insert additional optional breaks for texts, which are only observed as required. So-called local breaks can be entered, which only apply to the next structure, and so-called global breaks applying to the entire source code.

A distinction is made between breaks with and without hyphen. In order to obtain a unique identification for the input of breaks, two different symbols are used (`#` without break symbol, $\sim$ with break symbol).

5.5.1 Input of breaks

The user has several possibilities for the input of breaks.

Local breaks can be inserted in the source code via comments (`HYPHEN`) or using a special menu in the graphics editor. In this menu, the user can only insert breaks, but not change text, since this is the task of a source-code editor.

Global breaks can be specified in the output options, using also a comment (`G_HYPHEN`) at the beginning of the source code or by a specification in the profile (`G_HYPHEN`).

5.5.2 Internal treatment of breaks

The graphics editor provides a menu which displays the text pieces belonging to the current structure. Breaks can be inserted in this menu.

The generator only treats these breaks when a line must be broken. It then checks whether a hyphen should and can be inserted. If this hyphen does not fit in the line, the last complete piece of text is written to the next line. If a character string does not have any breaks and does not fit into the line, a hard division is made leaving as many characters as possible (at least 2) in the old line. A hyphen is inserted when a break was made between two letters.

5.6 Parser

The current version is able to analyse C and PASCAL programs. The aim was to process both ”real” programs and pseudocode. For PASCAL this aim was achieved by a ”pseudocode” syntax, which is an extension of the standard PASCAL syntax. C does not allow this due to a few particularities of the syntax. For this reason, a pseudocode parser was implemented and an ANSI-C parser is planned for a future version of *nassi*.

5.6.1 C–pseudocode

The preprocessor appertaining to C and the special features of the syntax require a very detailed analysis of the source code if an interpretation is desired which is correct in all points. As a rule, however, such an analysis will not accept ”pseudocode”. In order to be able to analyse both pseudocode and at least simple C programs with a parser, a syntax with the following features was defined:

A largely ANSI-C-conforming text is expected outside function definitions (i.e. for variables, type and function declarations). Within functions, on the other hand, a relatively free input is permissible.

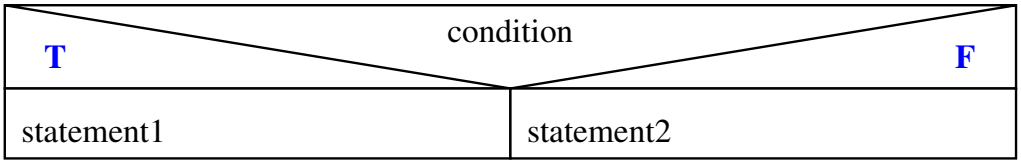
The restrictions and particular features involved in this compromise will be described in the following.

Mapping the C-structure elements on the structogram elements

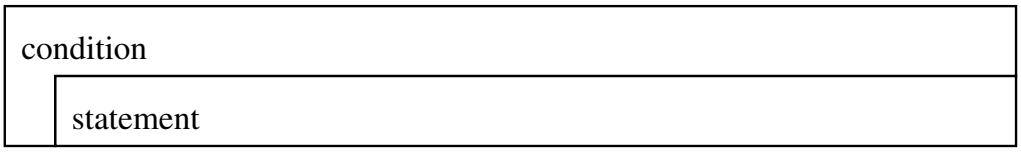
The programming language C contains types of statements which do not directly correspond to the Nassi-Shneiderman structure elements. Although a correct conversion is basically possible, it may lead to extremely unclear diagrams in some cases (especially with the `goto` statement), so that a different approach was selected.

The following types of C statements essentially correspond to the Nassi-Shneiderman structure elements:

- `if( condition ) statement1 else statement2`



- `while( condition ) statement`

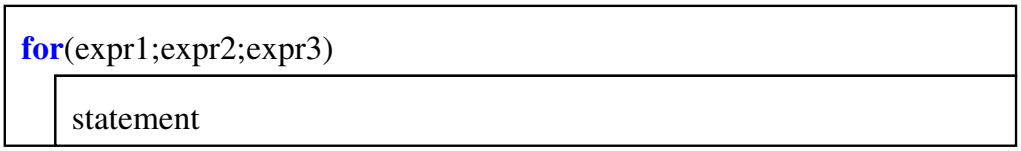


- `do statement while( condition )`
  It should be noted here that the Nassi-Shneiderman structure element corresponds to a "`do ... until`". The condition can be preceded by an arbitrary text using the generator option "add text for repeat-loop", e.g. by a negating "`!`".

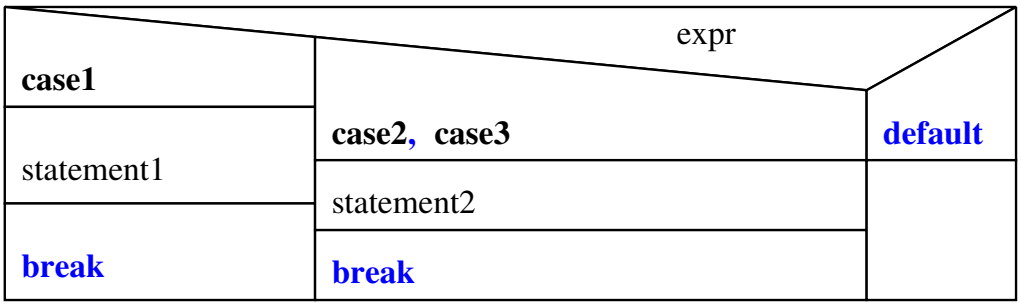
- Calls for functions defined within the program are marked accordingly.

In order to correctly represent the following types of statements, they must be generally decomposed into several statements. This was omitted for the sake of clarity.

- The `for` loop: `for( expr1; expr2; expr3)` statement is represented like the `while` loop with a starting condition.



- The `switch` statement in C does not correspond to the multiple choice structure element according to Nassi-Shneiderman , but is rather related to the "computed-goto" of FORTRAN. However, since it is generally used like a multiple choice by inserting the `break` statement, it is always represented in the diagram as such.



There is no direct equivalent of the statements `break continue` and `goto`. The conversion into Nassi-Shneiderman elements depends on the context in which they are used. These statements are represented as simple instructions in the diagrams.

Restrictions and special features

The following restrictions apply to the analysis of "real" C programs:

- As in ANSI-C, the type of a function must be explicitly specified in declarations/definitions (no implicit *int* as in K&R-C).
- Preprocessor directives are disregarded, i.e.:
  - macros are not expanded
  - include-files are ignored
  - conditional compilation is not performed
- Declarations of variables within functions cannot be distinguished from statements. They are therefore represented as statements, but can be suppressed by the graphics editor (see "hide" in 5.4.2 on page 94).
- If a pointer is used on a function in a statement, this statement appears as a function call in the diagram. The same applies to variables whose name is the same as that of a function. If necessary, errors can be corrected in the graphics editor using the option *mark.func.call* (see 5.4.2 on page 95).

The following should be noted when generating pseudocode:

- Keywords, braces and semicolons retain their normal meaning.
- Brackets (`() []`) must be used in pairs and nested correctly.
- Double and single quotes (`" '`) enclose strings that are not further analysed.
- The format of C statements is free — except for the above three items.
- Keywords and semicolons preceded by a `\` can be used in the text.
- German umlauts can be entered directly or generated in a similar way as in LaTeX, that is e.g. `\"A` for "Ä" oder `\"s` for ß.
- The sequence `\.`*X* generates the character *X*. In this way, for example, brackets not occurring in pairs can be generated or blanks forced in a diagram.
- The sequence `\\` is replaced by a line break in the diagram.

5.6.2 PASCAL

PASCAL also contains types of statements which do not directly correspond to the Nassi- Shneiderman structure elements. They are treated similarly to the corresponding C statements.

The following types of PASCAL statements correspond exactly to the Nassi-Shneiderman structure elements(cf. Figures in 5.6.1 on page 97):

- `IF condition THEN statement1 ELSE statement2`
- `WHILE condition DO statement`
- `REPEAT statement_list UNTIL condition`
- `CASE expression OF ... END`
  Although not defined by the standard, the specification of a default leaf (with the keyword `OTHERWISE` or `ELSE`) allowed in many compiler implementations is also accepted here.
- Calls for functions and procedures defined within the program are marked accordingly.

In order to correctly represent the following types of statements, they would in general have to be decomposed into several statements. This was omitted for the sake of clarity.

- The `for` loop: `FOR var := val1 TO/DOWNTO val2 DO statement` is represented like the `WHILE` loop.
- The `WITH var_list DO statement` instruction is not a structure element, since it merely declares abbreviating notations for `statement`. Nevertheless, such constructs are represented here like a `WHILE` loop.

There is no direct equivalent of the `goto` instruction. The conversion into Nassi-Shneiderman elements depends on the context in which it is used. This statement is represented like a simple instruction in the diagrams.

Restrictions and special features

The following restrictions apply to the analysis of "real" PASCAL programs:

- If a variable whose name is the same as that of a function is used in a statement, the statement appears as a function call in the diagram (unless it is an assignment of the return value of the function itself). If necessary, errors can be corrected in the graphics editor using the option *mark func.call* (see 5.4.2 on page 95).

The following should be noted when generating pseudocode:

- Keywords, colons and semicolons retain their normal meaning.
- Brackets (`()` `[]`) must be used in pairs and nested correctly.
- Single quotes (`'`) enclose strings that are not further analysed.
- The format of PASCAL statements is free — except for the above three items.
- Keywords, colons and semicolons preceded by a `\` can be used in the text.
- German umlauts can be entered directly or generated in a similar way as in LaTeX, that is e.g. `\"A` for "Ä" or `\"s` for "ß".
- The sequence `\.`*X* generates the character *X*. In this way, for example, brackets not occurring in pairs can be generated or blanks forced in a diagram.
- The sequence `\\` is replaced by a line break in the diagram.

5.7 Options in the Profile and Source Codes

5.7.1 Profile

A possibly existing profile is loaded when calling *nassi*. The options contained therein apply to the entire session unless they are overwritten by other options. In Profile the options have the following appearance:

```
Keyword = value
Keyword = value
...
```

Each option is written in a separate line. If the value of an option contains blanks, it must be provided with "...".

The keywords with their values are described in 5.8 on page 102. If the Profile has the name `./nassirc` or `$HOME/.nassirc`, it is automatically loaded upon call, searching first in the current directory. Otherwise, it must be specified with the option *-profile* or via the menu item *File→Load Profile*.

5.7.2 Comments in the Source code

The user can change the appearance of individual statements by means of special comments in the Source code. These comments contain e.g. information about fonts and colors. The parser stores

these comments in a list of hints, which is used by the generator.

However, the user must himself make sure that the values are correct.

Appearance of the comments

The comments begin and end like normal comments of the programming language. They are reconized as *Nassi special comments* (`NSC`) if the keyword `/* NSC:` or `/* NSC_GLOBAL:` is the first token inside the comment.

One or several pairs of keywords with corresponding values and separated by blanks must follow. The value may be enclosed in `"`.

A list of pairs is defined in 5.8.

Position and validity of comments

The NSC comments are always placed in front of the statement structure to which they are to apply. If they are placed within a statement, the parser will produce a syntax error message.

The comments cannot be nested.

If a comment is placed before a function name, it applies to the function head and the block below the function head. If it is placed between function name and block ("{" in C and "`BEGIN`" in PASCAL) it only applies to the block.

Comments preceded by `NSC_GLOBAL:` set options that are applicable to the entire Source code. Only one comment of this type is allowed. It must be placed before the first function in the Source file.

5.7.3 Hierarchy of the options

The different possibilities of specifying options require a defined hierarchy of execution and validity. The following list shows the order in which the options are read. If a keyword occurs several times, the value read last is valid.

1. Profile
2. starting area of the Source code
3. comments in the Source code

5.8 Annex: List of Keywords and Values

The following keywords are so far defined in order to set options:

Keyword	Possible values	Explanation
LINE_SKIP	SIMPLE / DOUBLE	line spacing
FUNCTIONHEAD	NONE / SMALL / FULL	type of function head
MINWIDTH	0.0,..,1.0	minimum width of a case or "if" leaf; percentage of total width
FONT_NAME	Helvetica, NewCenturySchlbk, Times,Courier	character set for normal text
FONT_FORMAT	roman,bold,italic, bold-italic	character set format for normal text
FONT_SIZE	8,..,34	character set size for normal text
FONT_COLOR	red, black,yellow,...	color for normal text
KEY_FONT_NAME	Helvetica,NewCenturySchlbk, Times,Courier	character set for keywords
KEY_FONT_FORMAT	roman,bold,italic, bold-italic	character set format for keywords
KEY_FONT_SIZE	8,..,34	character set size for keywords
KEY_FONT_COLOR	red, black,yellow,...	color for keywords
SIDE_DISTANCE	2,..,20	distance of text from lateral border (in points)
VERTICAL_DIST_BOT	2,..,20	distance of text from bottom border (in points)
VERTICAL_DIST_TOP	2,..,20	distance of text from top border (in points)
FILL_COLOR	red, black,yellow,...	color for background filling
LINE_COLOR	red, black,yellow,...	line color
HIDE	YES / NO	hiding the structure
EXCLUDE	YES / NO	structure is excluded
EXCLUDETEXT	any string	text for excluded structure
LANGUAGE	ENGLISH / GERMAN	language for keywords TRUE/FALSE or WAHR/FALSCH in an "if" structure
G_HYPHEN	any string	specification of global hyphenation proposals
HYPHEN	any string	specification of local hyphenation proposals
OUT_WIDTH	1.0,..,18.0	width of a diagram (in cm)
OUT_MAXHEIGTH	1.0,..,29.7	maximum height of a diagram, has no meaning in this version (in cm)
OUT_FORMAT	PS, EPS, TGIF	type of output
OUT_METHOD	"name of function", "prefix+number", "name of main input file"	type of output file name
OUT_PREFIX	"nassi."	prefix preceding the name of the output files
OUT_DIR	"./"	directory in which the output files are stored
PS_HEADER	YES / NO	adding a header for PostScript output
FOCUS	frame, invert, line, block, debug	way in which the statement under the mouse should be marked
COLSPEC_IF	0.0, ..., 100.0	percentage of the horizontal width reserved for the *then* leaf on an if-statement

Literaturverzeichnis

[1] WWW-Seiten zu Nassi
`http://www.kfa-juelich.de/zam/nassi`
Beschreibung, QuickTour und Download

[2] G. Egerer, R. Knecht, W. E. Nagel, Algorithmen und Strukturen in C,
Berichte des Forschungszentrum Jülich; 3587, ISSN 0944-2952

[3] B. W. Kernighan, , Ritchie, D. M., Programmieren in C,
Hanser , 1982, ISBN 3-446-15497-3

[4] Purify, Rational Software
`http://www.rational.com/products/purify_unix/index.jtmpl`

[5] Forms Library, A Graphical User Interface Toolkit for X,
`http://bragg.phys.uwm.edu/xforms`

[6] John R. Levine, Tony Mason and Doug Brown, Lex & Yacc,
O'Reilly & Associates, Inc., 1992, ISBN 1-56592-000-7

[7] CVS, Concurrent Versions System
`http://www.cyclic.com`
`http://www.loria.fr/ molli/cvs-index.html`

[8] `Tgifctrl`, Run Time Library in C to create TGIF object files
Author: Mats Bergstöm, University of Lund, Sweden. (Mats.Bergstrom@kosufy.lu.se)