

**Rogowski-Institut für Elektrotechnik der RWTH Aachen
Lehrstuhl für Allgemeine Elektrotechnik und
Datenverarbeitungssysteme
Prof. Dr.-Ing. Walter Ameling**

Entwicklung eines
syntaxgesteuerten
graphischen Petrinetzeditors

Diplomarbeit im Fach Informatik
Jürgen Abel
Matrikelnummer: 145259

Ausgegeben und betreut von:
Prof. Dr.-Ing. Walter Ameling
Dr.-Ing. Wolfgang Kubalski
Dipl.-Ing. Dieter Rosenthal

Eidesstattliche Erklärung:

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe.

Die verwendeten Literaturquellen sind vollständig im Literaturverzeichnis angegeben.

Aachen, im Juli 1990

Inhaltsverzeichnis

1. Einleitung.....	1
1.1. Bedeutung, Anwendung und Eigenschaften von Petrinetzen ...	1
1.2. Ziel der Arbeit	2
1.3. Aufbau der Arbeit	3
2. Petrinetze	4
2.1. Prädikat-/Transitionsnetze	4
2.2. NSL-Netze	7
2.3. Petrinetze und ihr Schaltverhalten	8
2.4. Petrinetze und Automaten	9
2.5. Graphische Darstellung von Petrinetzen	11
3. Syntaxsteuerung durch Parsergeneratoren	13
3.1. Wirkungsweise und Vorteile von Parsersteuerungen	13
3.2. Der Compilergenerator YACC	16
3.3. Erzeugung des Parsers durch YACC und sein Einsatz	19
4. Aufbau des Editors	20
4.1. Programmaufbau insgesamt	20
4.2. Struktur der Konfigurations- und Petrinetzdateien	23
4.3. Globale Variablen und Typen	26
4.4. Beschreibung der einzelnen Funktionsgruppen	32
4.4.1. Scanner	32
4.4.2. Parser	36
4.4.3. Bildschirmprozeduren.....	40
4.4.4. Objektprozeduren	42
4.4.5. Datensicherungs- und -ladeprozeduren.....	44
4.4.6. Druckprozedur	46
4.5. Beispiel eines Programmablaufs.....	47
5. Anwendung des Editors	49
5.1. Programmeinführung	49
5.2. Bedienung den Texteditors	53
5.3. Erläuterung der einzelnen Menüfunktionen	54
5.3.1. Funktionen des Generalmenüs	54
5.3.1.1. "Undelete"	55
5.3.1.2. "Load"	55
5.3.1.3. "Save"	56
5.3.1.4. "Hex"	56
5.3.1.5. "Print"	56
5.3.1.6. "Refresh window"	57
5.3.1.7. "Grid on/off"	57
5.3.1.8. "Display/Hide grid"	57

5.3.1.9. "Dataflows right/normal angled"	57
5.3.1.10. "Smooth/Unsmooth data flow"	58
5.3.1.11. "Symbolname bottom/center"	58
5.3.1.12. "Defaults"	58
5.3.1.13. "Quit"	59
5.3.2. Funktionen den Hintergrundmenüs	59
5.3.2.1. "Add place"	59
5.3.2.2. "Add transition"	59
5.3.2.3. "Add text"	60
5.3.2.4. "Setup"	60
5.3.3. Funktionen des Stellenmenüs	61
5.3.3.1. "Source"	62
5.3.3.2. "Nova"	62
5.3.3.3. "Resize"	62
5.3.3.4. "Default-resize"	62
5.3.3.5. "Take as default"	62
5.3.3.6. "Rename"	63
5.3.3.7. "Delete"	63
5.3.3.8. "Type"	63
5.3.4. Funktionen des Transitionenmenüs	64
5.3.4.1. "Source"	65
5.3.4.2. "Move"	65
5.3.4.3. "Resize"	65
5.3.4.4. "Default-resize"	65
5.3.4.5. "Take an default"	65
5.3.4.6. "Rename"	65
5.3.4.7. "Delete"	66
5.3.4.8. "Condition"	66
5.3.4.9. "Delay"	66
5.3.4.10. "Procedure"	66
5.3.5. Funktionen des Kantenmenüs	66
5.3.5.1. "Destination"	67
5.3.5.2. "Add connect point"	68
5.3.5.3. "Delete"	68
5.3.5.4. "Bend"	68
5.3.5.5. "Move connect point"	68
5.3.5.6. "Delete connect point"	68
5.3.6. Funktionen des Hintergrundtextmenüs	69
5.3.6.1. "Change"	70
5.3.6.2. "Move"	70
5.3.6.3. "Delete"	70

6. Programmiererweiterungen	71
6.1. Erweiterung des Menüs	71
6.2. Erweiterung des Parsers	74
6.3. Erweiterung der Objektprozeduren	75
6.4. Erstellung des ausführbaren Programms	76
7. Zusammenfassung und Ausblick	78
8. Literatur	81

1. Einleitung

1.1. Bedeutung, Anwendung und Eigenschaften von Petrinetzen

Petrinetze finden heutzutage immer mehr Beachtung. Das zeigt sich in der Literatur ebenso wie auf Tagungen. Die Anzahl der Fachgruppen und Magazine zu diesem Thema steigt laufend.

Der Grund hierfür liegt in der wachsenden Bedeutung von Strukturierungs- und Beschreibungsmöglichkeiten von Systemen aller Art, da die System-Komplexität dauernd zunimmt. Eine wichtige Beschreibungsform derartiger Systeme stellen Petrinetze dar.

Zur Anwendung kommen Petrinetze bei der Organisation und der Analyse von umfangreichen Vorgängen und Systemen wie Prozeß-Steuerungen und Rechnersystemen. Insbesondere bei der Strukturierung können sie von Vorteil eingesetzt werden. Die Systeme werden hierbei durch ein Netz unter Beachtung bestimmter Regeln graphisch dargestellt. So können mit Petrinetzen, von der Systemplanung angefangen bis zur Realisierung, Ablaufstrukturen erfaßt und untersucht werden. Auch das Umfeld von Systemen kann in die Beschreibung mit einbezogen und ausgewertet werden. Petrinetze beschränken sich hier nicht nur auf statische Zusammenhänge, sondern ermöglichen auch die Repräsentation dynamischen Verhaltens.

Viele Eigenschaften von Petrinetzen sind dabei von Bedeutung. Zum einen vermeiden sie einen unnötigen Formalismus, da Systeme bildlich anschaulich dargestellt werden können, zum anderen tragen sie durch die ihnen zugrundeliegenden Regeln zur Strukturierung bei.

Der Informationsaustausch und die Diskussion von Problemen werden durch Illustrierung erleichtert, da ein kurzes Diagramm oftmals aussagekräftiger und eindeutiger ist als eine lange schriftliche Spezifikation. Dennoch gestattet der Aufbau der Petrinetze eine genaue Systemdarstellung und Problemanalyse.

Bereits bei dem Systementwurf ermöglicht die Modellierung durch Petrinetze, Fehler zu erkennen und somit zu vermeiden. Die Gliederung eines Gesamtsystemes in Teilsysteme wird durch Aufteilung in einzelne Petrinetze, welche dann getrennt behandelt und untersucht werden können, unterstützt. Dadurch können auch nebenläufige Pro-

zesse modelliert werden, eine Eigenschaft, die grundlegend für die Theorie der Petrinetze ist.

Zuletzt sei noch auf die Möglichkeit der Systemverifikation hingewiesen. Systemaussagen können mit Petrinetzen auf Grund ihrer genauen Syntax und Semantik überprüft werden, wodurch sie sich von vielen anderen Systemmodellen unterscheiden.

1.2. Ziel der Arbeit

Die vorliegende Arbeit hat zur Aufgabe, einen syntaxgesteuerten graphischen Editor zur Erstellung von Petrinetzen zu realisieren. Syntaxgesteuert bedeutet hierbei, daß der Ablauf des Programmes von vorgegebenen Syntaxregeln abhängt, an Hand welcher Programmeingaben geprüft werden. Graphisch heißt, daß das Petrinetz auf einem Bildschirm gezeichnet und bearbeitet werden kann. Die Benutzereingaben werden vom Programm laufend ausgewertet, und syntaktische Fehler, die sich auf die Erstellung eines Petrinetzes beziehen, werden von vornherein vermieden oder direkt angezeigt. Ein solcher Editor wird im folgenden beschrieben.

Um Programmeingaben aller Art auf syntaktische Richtigkeit zu überprüfen, wird in der Informatik im Rahmen des Compilerbaus ein Parser eingesetzt. Er analysiert nacheinander die Eingaben entsprechend der ihm vorgegebenen Regeln, seiner Grammatik. Um einen derartigen Parser zu erstellen, dient der Parsegenerator YACC. Der Generator erzeugt aus einer Menge von BNF-ähnlichen Regeln einen zugehörigen Parser. Dieser verwaltet nun den gesamten Editor, indem er die Eingaben des Benutzers mit den jeweiligen Regeln vergleicht und entsprechende Programmaktionen auslöst. Er stellt damit einen Hauptteil der Arbeit dar.

Der Einsatz eines Parsers zur Programmsteuerung bietet viele Vorteile. Er gewährleistet einen strukturierten Programmaufbau, welcher übersichtlich und gut zu lesen ist. Programmänderungen können leicht an der Grammatik vorgenommen werden, ohne daß umfangreiche Änderungen an den Programmteilen anfallen; diese Arbeit übernimmt der Parsegenerator. Die Portierbarkeit des Programmes wird erhöht, da lediglich implementationsabhängige Prozeduren geändert werden müssen, während die Grammatik unverändert bleibt.

Dem Parser zur Seite steht eine Reihe von Funktionsblöcken. Sie dienen dazu, die Benutzeraktionen in einer bestimmten Form an den Parser weiterzuleiten und nach Auswertung auf dem Bildschirm darzu-

stellen. Des weiteren übernehmen sie Verwaltungsaufgaben wie das Laden, Speichern und Drucken von Petrinetzen.

Sowohl die Arbeit mit dem Programm als auch dessen Wartung erhalten durch die Parsersteuerung ein hohes Maß an Komfort, was der Erstellung der Petrinetze zugute kommt.

1.3. Aufbau der Arbeit

Die Arbeit ist in ihrer Gesamtheit in acht Kapitel gegliedert, deren erstes durch diese Einleitung gebildet wird. Im zweiten Kapitel werden Petrinetze erläutert. Dazu gehören die Beschreibung mehrerer Netzklassen, der Vergleich von Petrinetzen und Automaten, die Beschreibung des Schaltverhaltens von Petrinetzen sowie deren graphische Darstellung. Kapitel drei beschäftigt sich mit der Syntaxsteuerung durch Parsergeneratoren, dem Einsatz des Compilergenerators YACC und dem erzeugten Parser. Der Aufbau des Editors mit seinen Variablen, Typen und Funktionsgruppen befindet sich im vierten Kapitel. Auch ein typischer Programmablauf ist dort beschrieben. Im Anschluß daran wird im Kapitel fünf die Anwendung des Editors erklärt. Hier werden alle Menüs mit sämtlichen Funktionen und auch der implementierte Texteditor dargelegt. Programmerweiterungen behandelt das sechste Kapitel, und das siebte befaßt sich mit einer Zusammenfassung und einem Ausblick. Die Arbeit schließt mit den Literaturhinweisen im achten Kapitel.

2. Petrinetze

2.1. Prädikat-/Transitionsnetze

Im Rahmen der Netztheorie entwickelten Petri und seine Mitarbeiter in den sechziger Jahren die nach ihm benannten Petrinetze. Diese Netze dienen der Systemorganisation und behandeln vorwiegend parallele und nebenläufige Prozesse (/REI S85/, /REI S82/).

Die Netztheorie, welche sich mit Netzen aller Art beschäftigt, teilt Netze in eine Reihe von verschiedenen Netzklassen ein. Die Netzklassen unterscheiden sich in der Notation der Netzkomponenten und in der Interpretation des entsprechenden Netzmodelles.

Grundsätzlich kann man ein Netz als ein Tripel

$$N = (S; T; F)$$

auffassen, wobei

- S und T zwei disjunkte nichtleere Mengen sind, und
- F die sogenannte Flußrelation zwischen S und T ist.

Die Elemente von S werden dabei als Stellen, die Elemente von T als Transitionen bezeichnet.

In bezug auf die Themenstellung dieser Arbeit stellt die Graphentheorie eine treffende Definition zur Verfügung (/ZUSE80/):

- Ein Netz ist ein gerichteter Graph.
- Der Graph enthält zwei Arten von Knoten: Stellen und Transitionen.
- Die Kanten des Graphen laufen entweder von Stellen nach Transitionen oder umgekehrt.
- Der Graph enthält keine isolierten Knoten oder mehrfache Kanten.

Die graphische Darstellung solcher Graphen ist in Kapitel 2.5 näher erläutert.

Für die Klasse der Prädikat-/Transitionsnetze läßt sich nun folgende Definition aufstellen:

Ein Prädikat-/Transitionsnetz besteht aus:

- Einem Netz $N = (P; T; F)$.
Die Elemente von P sind n -stellige Prädikate ($n \geq 0$).
 T ist die Menge der Transitionen.
- Einer algebraischen Struktur S .
- Einer Pfeilbeschriftungsfunktion D .
Besitzt das Prädikat, von welchem ein Pfeil ausgeht beziehungsweise zu welchem ein Pfeil hinläuft, die Stelligkeit n , so erhält der Pfeil als Beschriftung ein n -Tupel, bestehend aus Elementen von S .
- Schaltbedingungen für die Transitionen.
Eine Schaltbedingung wird angegeben als eine logische Formel über der Struktur S .
- Einer Anfangsmarkierung M für jedes Prädikat.
- Einer Kapazitätsfunktion K .
Die Kapazitätsfunktion legt fest, wieviel Objekte gleicher Art jedes Prädikat aufnehmen kann.

Als Beispiel für ein Prädikat-/Transitionsnetz seien hier die denkenden Philosophen aufgeführt (/REI S82/):

"Drei Philosophen sitzen an einem Tisch. Jeder von ihnen hat einen gefüllten Teller vor sich sowie links und rechts eine Gabel. Zum Essen benötigt jeder Philosoph zwei Gabeln. Wenn ein Philosoph nicht ißt, dann denkt er."

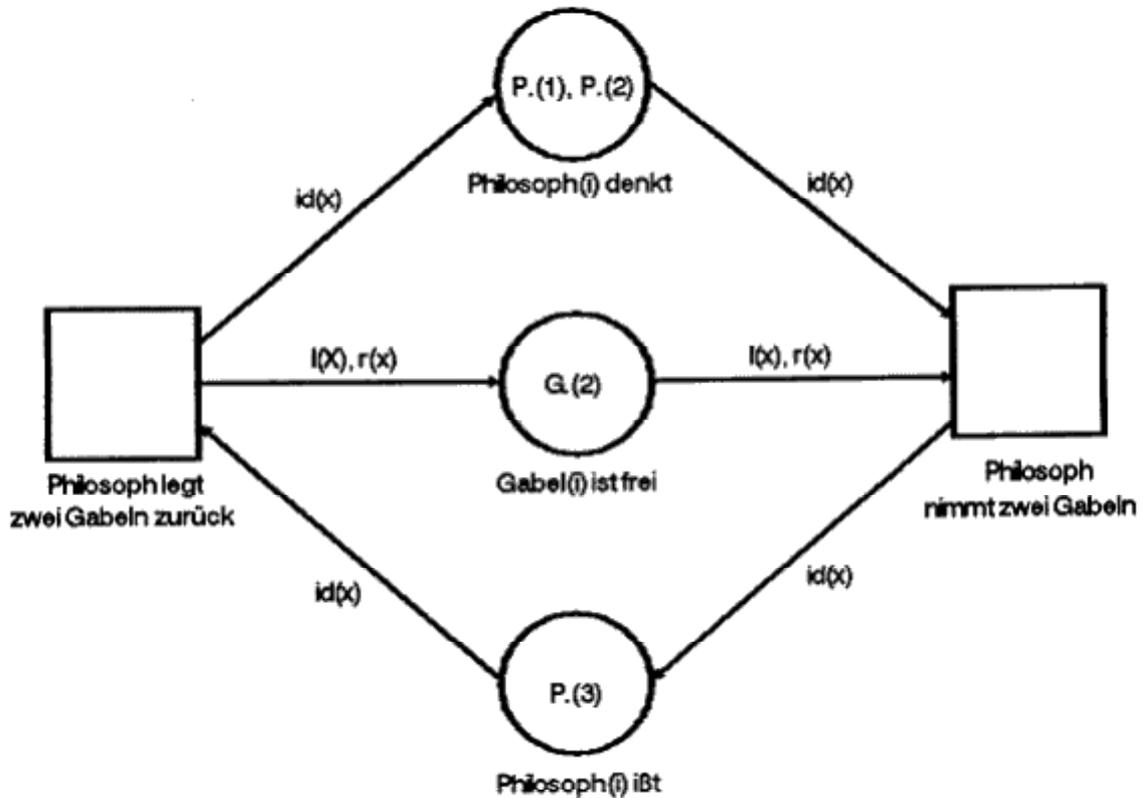


Bild 2.1: "Die denkenden Philosophen"

Dieses Beispiel läßt sich nun als Prädikat-/Transitionsnetz interpretieren:

Es existieren drei Prädikate:

- P1 = "Philosoph(P) denkt"
- P2 = "Philosoph(p) ißt"
- P3 = "Gabel (g) ist frei"

Es gibt zwei Transitionen:

- T1 = "Ein Philosoph nimmt zwei Gabeln"
- T2 = "Ein Philosoph legt zwei Gabeln zurück"

Die Menge, für welche die Prädikate zutreffen, lautet:

- P = {Philosoph(1), Philosoph(2), Philosoph(3)}
- G = {Gabel (1), Gabel (2), Gabel (3)}

Die Funktionen haben folgende Form:

$$\begin{aligned} l : P &\rightarrow G, \text{ mit } l(\text{Philosoph}(i)) = \text{Gabel}(i), & \text{für } i \in (1, 2, 3) \\ r : P &\rightarrow G, \text{ mit } r(\text{Philosoph}(i)) = \text{Gabel}(i+1), & \text{für } i \in (1, 2), \\ & r(\text{Philosoph}(3)) = \text{Gabel}(1) \\ id : P &\rightarrow P, \text{ mit } id(\text{Philosoph}(i)) = \text{Philosoph}(i), & \text{für } i \in (1, 2, 3) \end{aligned}$$

Für die Anfangsmarkierung gilt, daß dem Prädikat "Philosoph(p) denkt" die Elemente Philosoph(1) und Philosoph(2), dem Prädikat "Philosoph(p) ißt" das Element Philosoph(3) und dem Prädikat "Gabel(g) ist frei" das Element Gabel(2) zugeordnet wird.

Durch diese Anfangsbelegung entsteht das in Bild 2.1 wiedergegebene Netz.

2.2. NSL-Netze

Die NSL-Netze sind eine Erweiterung der Prädikat-/Transitionsnetze. Die Klasse der Prädikat-/Transitionsnetze wird hierbei um das Zeit- und Inspektionskonzept erweitert.

Das Inspektionskonzept dient einem nur lesenden Zugriff der Transitionen auf die Marken der Prädikate, wobei die Marken nicht bewegt werden. Hierfür werden die sogenannten Inspektionskanten eingeführt, welche als gestrichelte Linien dargestellt werden.

Das Zeitkonzept dient der Beschreibung zeitlicher Abhängigkeiten innerhalb des Netzes. Hierzu kann jeder Transition neben der Schaltbedingung eine Verzögerungszeit zugeordnet werden. Diese Verzögerungszeit beschreibt, wie lange eine Marke im Vorbereitungsbereich der Transition gelegen haben muß, bis sie an den Schaltvorgängen der Transition teilnehmen kann.

NSL-Netze können im Rahmen einer Simulation getestet werden, indem sie durch einen entsprechenden NSL-Compiler in ein ausführbares Programm übersetzt werden. Die hierbei benutzte Grammatik wird im folgenden als NSL-Sprache bezeichnet. In bezug auf diesen Compiler können mit Hilfe des beschriebenen Petrineteditors Textstücke in der NSL-Sprache erzeugt, syntaktisch geprüft und den entsprechenden Objekten zugeordnet werden.

2.3. Petrinetze und ihr Schaltverhalten

Petrinetze bestehen ganz allgemein aus gerichteten Graphen mit zwei Arten von Knoten, den Stellen und den Transitionen. Zu Beginn werden den Stellen sogenannte Marken zugewiesen, welche später über Transitionen zu anderen Stellen bewegt werden.

Der Ablauf innerhalb eines Petrinetzes kann nun als ein Markenspiel aufgefaßt werden, wobei das Netz als ein Spielbrett dargestellt wird (/ZUSE80/). Dabei sind zwei verschiedene Arten von Spielregeln zu unterscheiden:

- Die spezielle Spielregel setzt voraus, das jede Stelle mit maximal einer Marke belegt wird (dies kommt der Automatentheorie sehr entgegen).
- Die allgemeine Spielregel erlaubt die Belegung der Stellen mit mehr als einer Marke.

Marken werden nun ausgetauscht, indem Transitionen schalten. Darunter versteht man den Übergang der Marken von einer zur anderen Stelle (/ZUSE80/), wie es in Bild 2.3 abgebildet ist. Hierbei wird die Anzahl der Marken bei den einlaufenden Stellen um eins vermindert und bei den nachfolgenden Stellen um eins erhöht. Zum Schalten einer Transition, auch Feuern genannt, ist es jedoch erforderlich, diese zunächst zu aktivieren. Dies ist gegeben, wenn alle einlaufenden Stellen mit mindestens einer Marke belegt sind, und alle nachfolgenden Stellen noch nicht bis zu ihrer Kapazität aufgefüllt sind. Kann man einer einlaufenden Stelle genau eine nachfolgende zuordnen, so läßt sich das Schalten der Transition auch so interpretieren, daß die Marke von der einlaufenden zur nachfolgenden Stelle wandert. Man spricht dann, gemäß /ZUSE80/ Seite 28, von einem sogenannten Markenpfad.

Der Schaltvorgang wird normalerweise als zeitlos angenommen, so daß verschiedene Transitionen unabhängig voneinander schalten können. Dadurch wird die Untersuchung von unabhängigen nebenläufigen Vorgängen ermöglicht.

Ein solches Markenspiel erlaubt die Interpretation des Petrinetzes und die Simulation seines dynamischen Verhaltens.

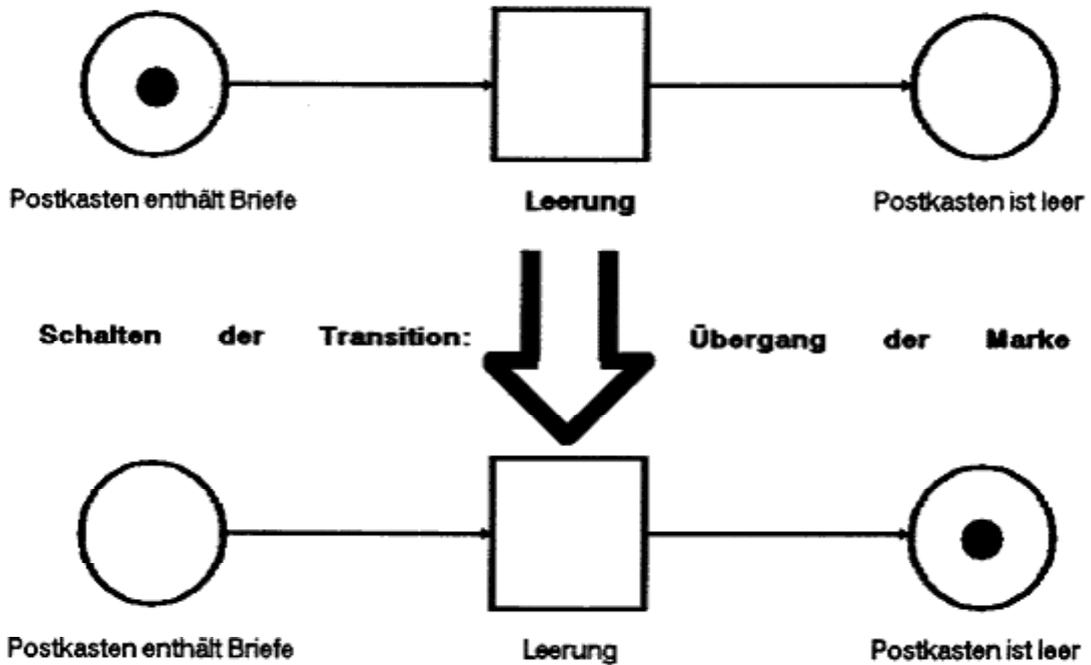


Bild 2.3: Schalten einer Transition

2.4. Petrinetze und Automaten

Die Theorie der Petrinetze und die Automatentheorie stehen in einem engen Zusammenhang. Die Automatentheorie ist ein verbreitetes Hilfsmittel zur Beschreibung komplexer Vorgänge und stellt diese an Hand von Zuständen und deren Übergängen untereinander mittels Zustandsgraphen dar (/ZUSE80/). Der Ingenieur greift im allgemeinen eher auf die Darstellung mit Zustandsgraphen zurück als auf eine abstrakte Gliederung in Netze. Aus diesem Grunde erfolgt an dieser Stelle ein kurzer Vergleich der Netz- und Automatentheorie.

Die Automatentheorie ordnet jedem möglichen Zustand einen Knoten innerhalb eines Graphen zu. Die Übergänge entsprechen gerichteten Kanten, welche die Knoten miteinander verbinden. Hierbei wird getaktetes Arbeiten des Automaten angenommen, daß heißt, der Automat durchläuft Schritt für Schritt eine Folge von Zuständen, wobei in jedem Takt ein neues Eingabesymbol von außen zugeführt wird.

Man kann nun jedem derartigen Automaten ein Petrinetz zuordnen, indem man die Zustandsknoten des Automaten auf Stellen und die Übergangskanten auf Transitionen mit je einem ein- und einem auslaufendem Pfeil (siehe Bild 2.4) abbildet.

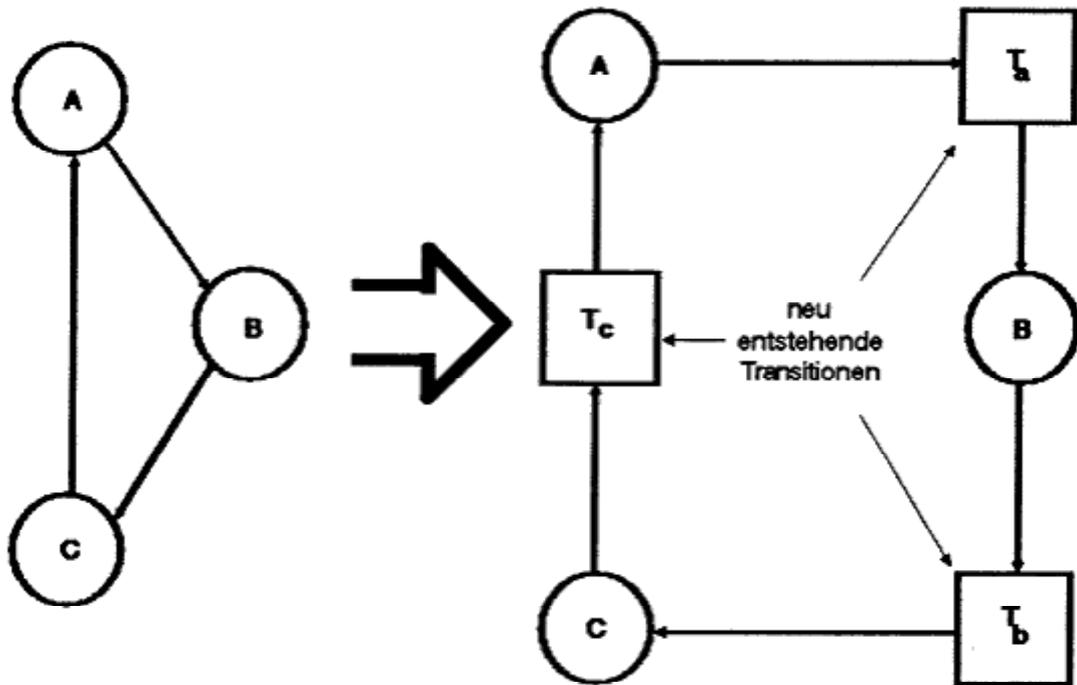


Bild 2.4: Umformung eines Automaten in ein Petrinetz

Zu beachten ist jedoch, daß bei einem Automaten dasselbe Eingabesymbol als Übergangsbedingung an verschiedenen Kanten vorkommen kann. Dieses Symbol muß bei der Umwandlung in ein Petrinetz in zwei unterschiedliche Transitionen abgebildet werden.

Es ergeben sich bei der Automatentheorie gegenüber den Petrinetzen zwei Nachteile:

- viele Vorgänge haben keine strenge zeitliche Taktung, beziehungsweise laufen parallel ab, und
- bereits bei einfachen Vorgängen ist die Anzahl der Zustände sehr hoch.

Diese Nachteile umgeht die Theorie der Petrinetze. Zum einen erlauben Petrinetze die Darstellung paralleler Abläufe, da sie keine zeitliche Taktung kennen, zum anderen ist es möglich, Vorgänge in einzelne Komponenten zu zerlegen, welche dann getrennt betrachtet werden können.

2.5. Graphische Darstellung von Petrinetzen

Bei der Darstellung von Petrinetzen werden für die Knoten und Kanten des Netzes bestimmte graphische Symbole gebraucht.

Stellen werden als Kreise dargestellt. Bei Transitionen ist es üblich, sie entweder als Rechtecke oder als vertikale Balken wiederzugeben, wie es in Bild 2.5.1 ersichtlich ist.

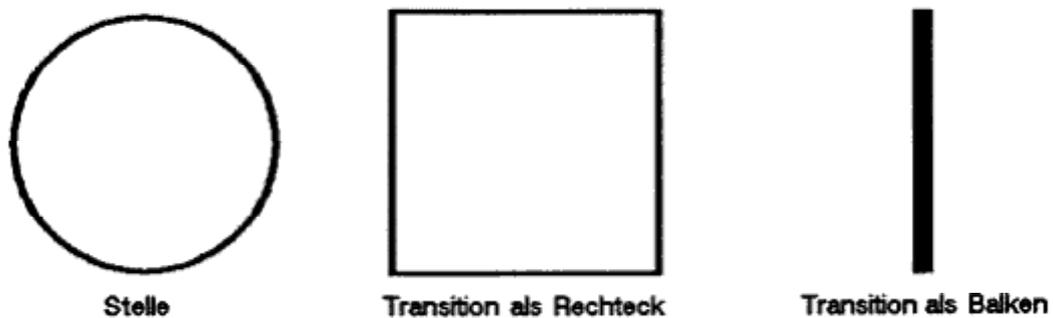


Bild 2.5.1: Symbole für Stellen und Transitionen

Die gerichteten Kanten werden in der Darstellung als Pfeile gezeichnet. Ein typisches Beispiel für graphische Darstellung eines Petrinetzes ist in Bild 2.5.2 zu erkennen.

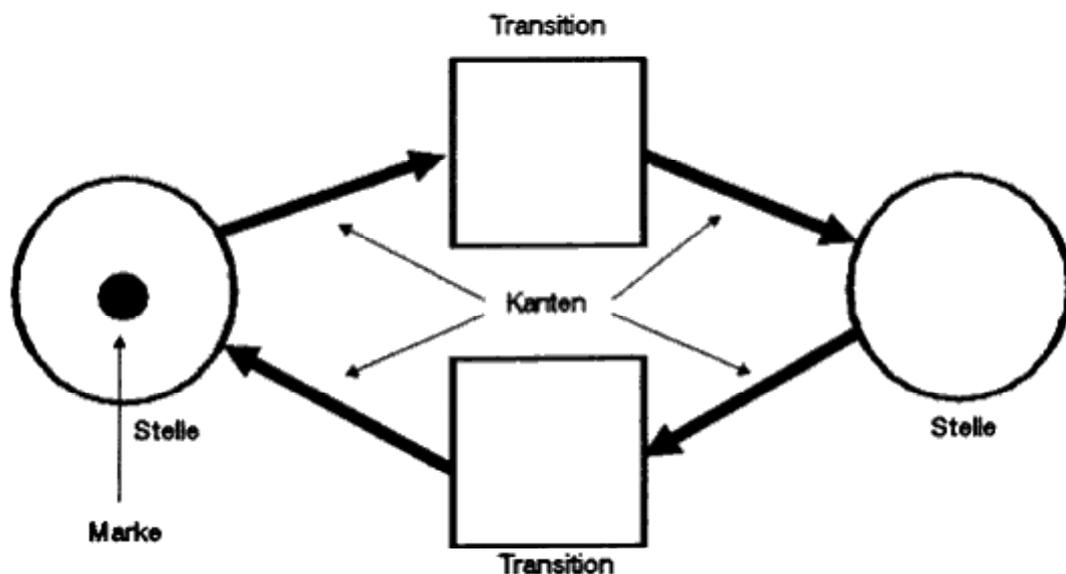


Bild 2.5.2: Graphische Darstellung eines Petrinetzes

Der Petrinetzeditor ermöglicht nun die Erstellung solcher Darstellungen auf dem Bildschirm. Dabei können die Größe und die Farbe der Kreissymbole für die Stellen beliebig gewählt werden. Für die Transitionen besteht die Möglichkeit, Farbe, Länge und Breite zu bestimmen, so daß beide Darstellungsarten der Transitionen gewählt werden können. Kanten werden als Pfeile wiedergegeben, wobei die Länge und der Winkel ihrer Pfeilspitzen einstellbar sind.

Das Zeichnen der Symbole und ihre Bearbeitung erfolgt unter Zuhilfenahme der Maus direkt am Bildschirm. Im Rahmen einer Programmeinführung wird das Zeichnen in Kapitel 5.1 näher erläutert.

3. Syntaxsteuerung durch Parsergeneratoren

3.1. Wirkungsweise und Vorteile von Parsersteuerungen

Ein Parsergenerator, wie im vorliegenden Falle YACC ("YET ANOTHER COMPILER COMPILER" /JOHN75/), dient der Generierung eines Parsers aus einer Grammatik, welche aus BNF-ähnlichen Regeln besteht, auch Produktionen genannt. Der Parser ist in der Lage zu prüfen, ob seine Eingaben den Regeln der Grammatik entsprechen (/WIRT86/, /KROE89/). Hierzu liest er schrittweise die Eingabe und geht nach Auswertung des aktuellen Eingabesymbolen in einen bestimmten Programmzustand über. Dieser Prüfvorgang kann durch einen Baum graphisch verdeutlicht werden, wie in Bild 3.1.1 zu sehen ist. Hierbei entsprechen die Knoten des Baumes den jeweils gültigen Terminalsymbolen der Grammatik und damit den zugehörigen Programmzuständen. Der Parser beginnt mit dem Lesen des Startsymbols im Startzustand und läuft den Knoten des Baumes entlang, entsprechend den eingenommenen Programmzuständen. Um die Eingabesymbole zu erhalten, bedient er sich eines sogenannten Scanners. Der Scanner hat die Aufgabe, die Eingabe zu lesen und für den Parser in Symbole umzuwandeln. Diese Symbole werden Token genannt und vom Parser ausgewertet, indem sie mit den gültigen Terminalsymbolen des aktuellen Programmzustandes verglichen werden. Bei Übereinstimmung wird der zugehörige neue Programmzustand eingenommen und das nächste Token mit Hilfe des Scanners gelesen und so weiter. Sollte an irgend einer Stelle kein passendes Terminalsymbol, und damit Programmzustand, für das aktuell gelesene Token vorhanden sein, so liegt ein syntaktischer Fehler vor, und die Eingabe entspricht nicht der Grammatik.

Im allgemeinen ist man bei Parsersteuerungen jedoch weniger an der Frage interessiert, ob die Eingabe syntaktisch richtig ist oder nicht, als vielmehr an den Schritten, welche bei der Parsung durchlaufen werden, entsprechend den Knoten des oben beschriebenen Baumes. In jedem Schritt können Prozeduren aufgerufen werden, so daß das Programm im Falle, daß die Eingabe von einem Benutzer erfolgt, in einen interaktiven Dialog mit dem Benutzer treten kann.

Der Fall, daß für eine bestimmte Eingabe keine passende Regel gefunden werden kann, ist dabei vom Programmierer des Parsers aususchalten.

Dies kann zum Beispiel dadurch geschehen, daß für alle möglichen Token immer eine entsprechende Regel vorhanden ist. Das führt jedoch im allgemeinen zu einer unüberschaubar großen und unhandlichen Grammatik. Aus diesem Grunde ist eine andere Möglichkeit vorzuziehen. Hierbei schalten die vom Parser aufgerufenen Prozeduren den Scanner in bestimmte Zustände, so daß dieser die Eingabe immer auf die aktuell jeweils gültige Tokenmenge abbildet. Dadurch halten sich die nötigen Regeln innerhalb eines vernünftigen Rahmens.

Wurde der Baum bis an eines seiner Enden ohne Fehler durchlaufen, ist damit die Eingabe als syntaktisch richtig erkannt, und der Parser hält an.

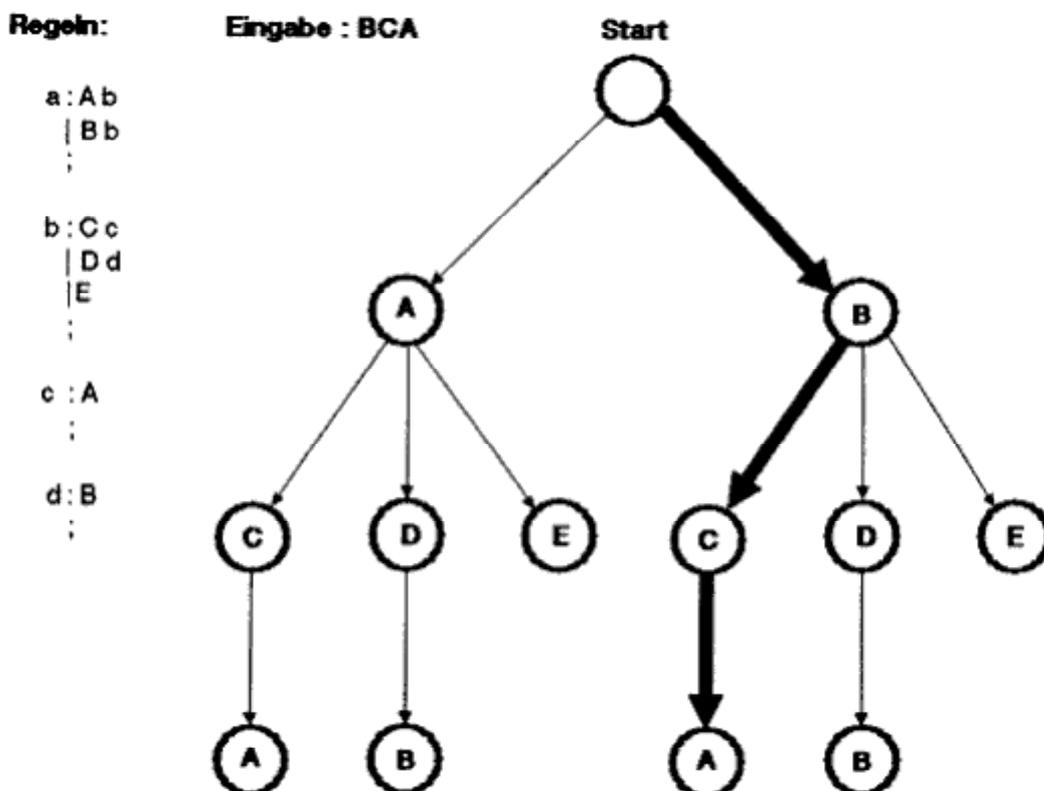


Bild 3.1.1: Verdeutlichung des Parsens durch einen Baum

Da man die Regeln zur Erzeugung des Parsers auch als ein Programm, welches mit dem Parsergenerator YACC compiliert wird, interpretieren kann, wird für diese Regeln im folgenden der Begriff des Parserprogrammes verwendet.

Ein derartiges Parserprogramm zeichnet sich durch zwei Aspekte aus:

- es ist streng strukturiert und
- nur an den YACC-Compiler gebunden.

Daraus läßt sich eine Reihe von Vorteilen ableiten:

- das Parserprogramm ist sehr übersichtlich,
- der Baum, der bei dem Parsen durchlaufen wird, und damit die Reaktion auf verschiedene Eingaben, läßt sich leicht verfolgen,
- Programmweiterungen sind einfach vorzunehmen und
- das Parserprogramm ist vom Rechnertyp unabhängig.

Des weiteren besteht die Möglichkeit, durch den Scanner die Benutzereingaben zu filtern und zusammenzufassen. Hierdurch wird der Parser stark entlastet, und die Effizienz des gesamten Programmes nimmt zu. Im vorliegenden Falle geschieht dies beispielsweise bei dem Anwählen eines Menüeintrages. Die gesamte Programmsteuerung vom Öffnen des Menüs bis zum Schließen erfolgt durch den Scanner. Dieser gibt danach den jeweiligen Menüeintrag als Token an den Parser zurück.

In Bild 3.1.2 ist noch einmal symbolisch die Funktionsweise einer Parsersteuerung zu erkennen. Der Parser erhält seine Eingaben nicht vom Benutzer direkt, sondern über den Scanner. Er analysiert die Token dann an Hand seiner Grammatik, dem Parserprogramm. Dabei ruft er die zugehörigen Prozeduren auf, welche für den Benutzer eine Antwort auf seine Eingaben darstellen.

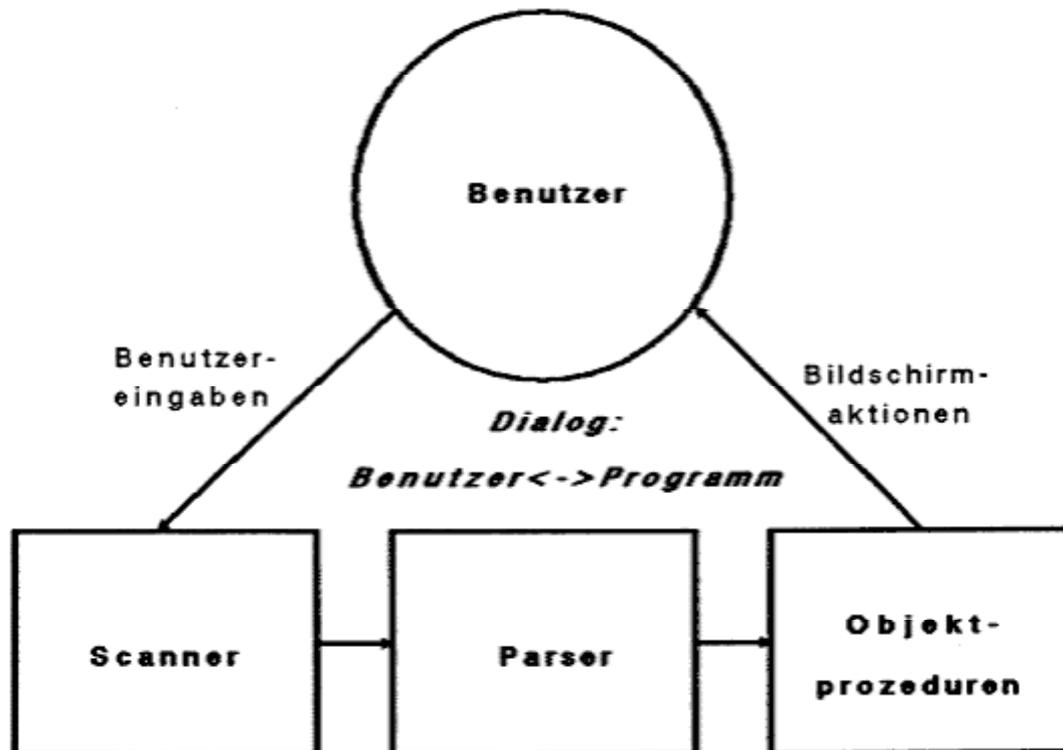


Bild 3.1.2: Funktionsweise einer Parsersteuerung

3.2. Der Compilergenerator YACC

Da der Parser zusammen mit dem Scanner und den zugehörigen Prozeduren einen Compiler darstellen, wird der Parsergenerator YACC auch als Compilergenerator bezeichnet (/JOHN75/). Ursprünglich erzeugte YACC aus der eingegebenen Grammatik einen Parser in der Programmiersprache C. Bei dieser Arbeit wurde jedoch eine abgeänderte Version des Generators verwendet, welche ein Pascalprogramm als Ausgabe liefert.

Im Normalfall besteht die Eingabedatei für YACC aus vier Teilen:

- der Tokendeklaration,
- Pascal-Variablen- und Prozedurdeklarationsteil,
- den Regeln in BNF-ähnlicher Notation und
- dem Hauptprogrammteil.

Bei der Realisierung des Petrinetzeditors wurden dabei der Prozedurdeklarationsteil und der Hauptprogrammteil in eigene Dateien ausgelagert.

Der Tokendeklarationsteil definiert die gültigen Token, welche in den Regeln als Terminalsymbole auftauchen. Ein Beispiel für eine Tokendeklaration ist:

```
%token _PLACE.
```

Hierbei wird das Token `_PLACE` als ein gültiges Terminalsymbol vereinbart.

Der Regelteil enthält die Grammatik und die zugehörigen Prozeduren. Er hat die Form:

```
A : _PLACE B  
  | _TRANSI TI ON C  
  ;
```

Hierbei stellen `_PLACE` und `_TRANSI TI ON` jeweils ein Terminalsymbol dar und B und C ein Nonterminalsymbol, daß heißt, eine weitere Regel.

In Bild 3.2 ist ein Beispiel für eine einfache YACC-Eingabedatei zu finden. Hierbei ist "begi nn" die Regel, mit welcher der Parser seine Syntaxanalyse beginnt. Eine gültige Eingabe für diese Grammatik ist zum Beispiel "KARL". Die aufgeführten Prozeduren ProcK, ProcA, ProcR und so weiter stellen dabei Programmteile dar, welche bei Erkennen des jeweiligen Tokens ausgeführt werden. Nachdem das Zeichen "L" erkannt wurde, bricht der Parser ordnungsgemäß die Programmbearbeitung ab.

```
%start      begi n
%token      K, Ä, R, L, O, T, C, E, S
%token      U, G

%%

begi n      : K { ProcK } r1
            | O { ProcO } r2
            | C { ProcC } r3
            | A { ProcA } r4
            ;

r1          : A { ProcA } r5
            ;

r2          : T { ProcT } r6
            ;

r3          : A { ProcA } r7
            ;

r4          : U { ProcU } r8
            ;

r5          : R { ProcR } L { ProcL }
            ;

r6          : T { ProcT } r9
            ;

r7          : E { ProcE } S { ProcS } A { ProcA } R { ProcR }
            ;

r8          : G { ProcG } U { ProcU } S { ProcS } r10
            ;

r9          : O { ProcO }
            ;

r10         : T { ProcT } U { ProcU } S { ProcS }
            ;

%%
```

Bild 3.2: Beispiel für eine YACC-Grammatik

3.3. Erzeugung des Parsers durch YACC und sein Einsatz

Die Generierung des Parsers mit Hilfe von YACC vollzieht sich in drei Stufen. Zuerst wird die Datei bearbeitet, welche die Grammatik mit der Tokendeklaration beinhaltet. Danach wird der Compilergenerator YACC mit dem Namen der Eingabedatei als Programmzeilenparameter aufgerufen. Die durch ihn erzeugte Datei wird dann mit einem Pascalcompiler zu einem ausführbaren Programm übersetzt. Dabei müssen die fehlenden Programmteile, wie Scanner und alle Prozeduren, mit eingebunden werden.

Sollte YACC bei der Generierung einen Fehler feststellen, so sind die Stufen von vorne zu durchlaufen.

Der Ablauf ist in Bild 3.3 noch einmal zusammengefaßt und in Kapitel 6.4 an Hand eines konkreten Beispiel dargelegt.

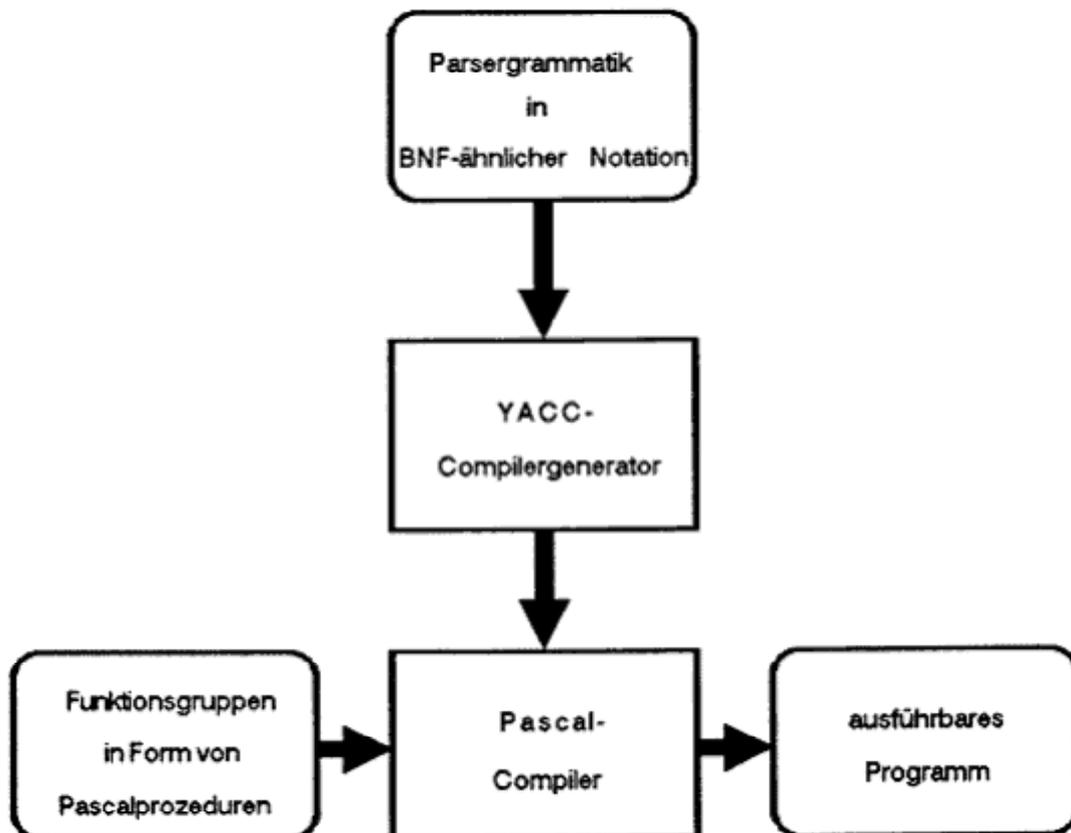


Bild 3.3: Erstellung eines Parsers mit YACC

4. Aufbau des Editors

4.1. Programmaufbau insgesamt

Der Petrinetzeditor wurde auf einem IBM PC mit einer EGA-Graphikkarte implementiert. Als Eingabegerät wurde neben der Tastatur eine Microsoft-kompatible Maus mit dazugehörigem Maustreiber eingesetzt.

Die Programmerstellung erfolgte folgendermaßen: Alle Teile außer dem Parser wurden in Pascal programmiert. Der Parser wurde durch den Parsergenerator YACC erzeugt, welcher ebenfalls ein Pascalprogramm als Ergebnis liefert. Schließlich wurden alle Teile und der in Pascal generierte Parser mit Turbo Pascal V5.5 in ein ausführbares Programm übersetzt. Die hierfür zu compilierende Hauptdatei, welche alle anderen nötigen Dateien mitsamt dem Parser einbindet, trägt den Namen "PNE. PAS".

Der gesamte Programmaufbau mit allen Funktionseinheiten ist in Bild 4.1 dargestellt.

Die erste Einheit, welche die Eingaben des Benutzers unmittelbar entgegennimmt und auswertet, ist die des Scanners. Der Scanner erkennt das Bewegen der Maus, das Drücken der Maustasten und das Betätigen der Tastatur. Er verwaltet mittels eines Maustreibers das Maussymbol auf dem Bildschirm. Da für die Steuerung des Programmes nur die linke der beiden Maustasten erforderlich ist, ist im folgenden mit der Maustaste immer die linke Maustaste gemeint.

Um den vielfältigen Arten der Eingaben gerecht zu werden, läßt sich der Scanner in mehrere Zustände schalten. Je nach Zustand interpretiert er die Eingaben verschieden und gibt dafür unterschiedliche Token aus. Bei der Anwahl eines Menüs durch Betätigen der Maustaste auf dem entsprechenden Symbol übernimmt er auch die Invertierung des Menübalkens, welcher sich unter dem Mauszeiger befindet.

Die vom Scanner ausgegebenen Token werden vom Parser sequentiell entgegengenommen und verarbeitet. Der Parser übernimmt die gesamte Verwaltung und Steuerung des Programmes.

Er besteht aus einer BNF-ähnlichen Grammatik, welche den Parser vollständig bestimmt. Diese Grammatik beschreibt die Syntax von Petrinetzen und beinhaltet Programmaktionen, welche den Scanner in die entsprechenden Zustände schalten.

Jedesmal wenn ein Terminal-Symbol innerhalb der Grammatik erwartet wird, wird der Scanner aufgerufen, um ein neues Token zu liefern. Dieses Token wird entsprechend der Grammatik interpretiert und verarbeitet. Hierzu dienen dem Parser eine Reihe von verschiedenen Funktionsgruppen.

Die Verwaltung des Bildschirms wird von einer dieser Gruppen übernommen. Dazu zählen Prozeduren, um die Stellen und Transitionen an ihrer entsprechenden Lage zu zeichnen und zu beschriften. Ebenso können Kanten mit allen Verbindungspunkten und Pfeilen an den Enden gezeichnet werden. Andere Prozeduren übernehmen das Zeichnen der oberen Menüleiste oder des Gitters. Für das Zeichnen und Löschen der Menüs und Eingabefenster sind eine ganze Reihe von Prozeduren zuständig. Hierbei können viele Parameter, wie der Durchmesser der Stellen und Verbindungspunkte, Länge und Breite der Transitionen, Größe des Gitters, Farben der Stellen, Transitionen, Kanten sowie aller Menüs, vom Benutzer frei eingestellt werden.

Eine andere Funktionsgruppe ist für die Objektverwaltung zuständig. Stellen, Transitionen, Kanten und Verbindungspunkte können erzeugt und gelöscht werden. Alte Namen sind veränderbar, ebenfalls Größe und Lage der Objekte. Verbindungspunkte können spitz zulau fend oder abgerundet dargestellt werden. Eingabe und Verbesserung von Objekttexten sind ebenfalls möglich.

Ein Petrinetz kann als gesamtes Projekt, daß heißt mit den Umgebungsparametern wie Farben, Größe der Stellen und Transitionen und so weiter, abgespeichert und geladen werden. Hierfür sind Datensicherungs- und -ladeprozeduren vorgesehen, welche auch eine Datei mit der aktuellen Konfiguration bei dem Start des Editors laden und am Programmende abspeichern.

Damit das erstellte Netz ausgedruckt werden kann, existiert eine Druckprozedur. Diese gibt die Stellen, Transitionen und Kanten als PostScript-Objekte in eine Datei aus, die später an ein PostScript-fähiges Ausgabegerät weitergegeben werden kann.

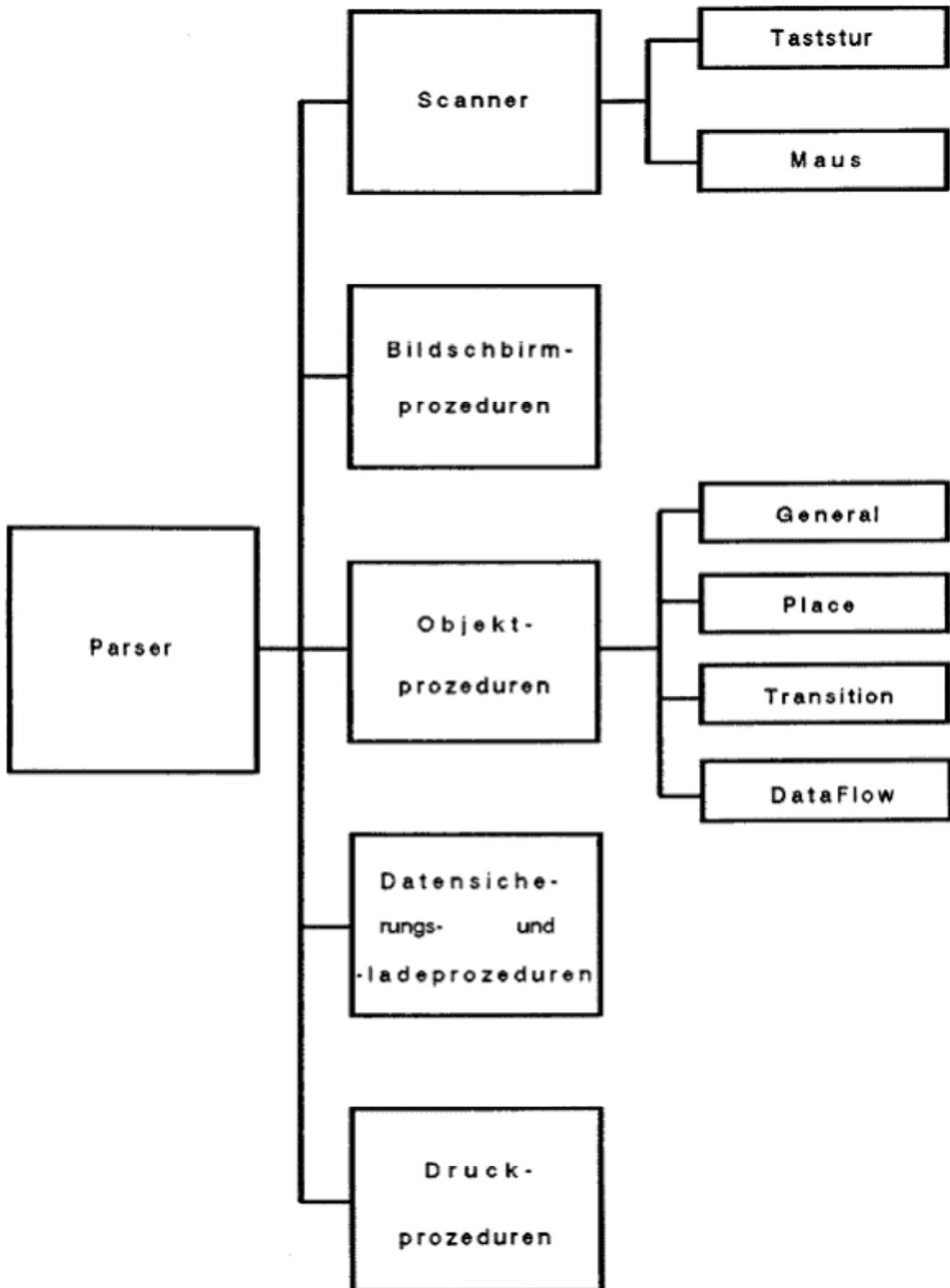


Bild 4.1: Programmaufbau des Editors mit Funktionseinheiten

4.2. Struktur der Konfigurations- und Petrinetzdateien

Um die Umgebungsparameter des Editors, dazu zählen Farben, Größe der Symbole, Gitteraktivierung und -größe et cetera, nicht bei jedem Programmstart erneut auf die individuell gewünschten Werte einstellen zu müssen, bedient sich der Editor einer Konfigurationsdatei. Diese Datei wird unmittelbar nach dem Start des Programmes geladen. In ihr sind alle Parameter, welche der Benutzer bei dem letzten Programmlauf benutzt hatte, enthalten. Mit diesen Daten werden dann die aktuellen Parameter initialisiert, so daß der Benutzer die gleiche Parameterumgebung wie beim letzten Programmlauf vorfindet. Beim Beenden des Programmes werden die zuletzt eingestellten Werte in der Konfigurationsdatei wieder abgelegt.

Das verwendete Datenformat für die Parameter sind ASCII-Strings. Zahlen werden dabei vorher in ein achtstelliges Format umgewandelt. In Kapitel 4.4.5 sind die dafür verwendeten Prozeduren genau beschrieben. Bild 4.2.1 zeigt den Aufbau der Konfigurationsdatei und die Bedeutung jedes ASCII-Strings.

Um ein ganzes Petrinetzwerk abspeichern zu können, wird eine umfangreichere Struktur benötigt. Hierbei müssen die Informationen über alle Stellen, Transitionen und Kanten mit gesichert werden. Die dafür benutzte Dateistruktur gliedert sich in fünf Teile:

- Umgebungsparameter,
- Textarray für eventuellen Setuptext,
- Liste der Stellen,
- Liste der Transitionen,
- Liste der Kanten,
- Liste der Hintergrundtexte

und ist in Bild 4.2.2 wiedergegeben.

Der Aufbau der Listen für die Stellen, Transitionen, Kanten und Hintergrundtexte wird im nächsten Kapitel erläutert.

```
PNE. CFG          { Namenskennung          }
"END"            { Ende der Namenskennung }

INIT             { Beginn des INIT-Teiles  }
Test             { Name der zuletzt bearbeiteten Datei }
Test-Datei      { Name des zuletzt bearbeiteten Netzes }
10. 07. 90      { Datum des zuletzt bearbeiteten Netzes }
0004000         { Breite der Transitionen }
0006000         { Hoehe der Transitionen }
0002000         { Radius der Stellen }
0000300         { Radius der Verbindungspunkte }
0005236         { Winkel der Pfeile an den Kanten }
0000800         { Laenge der Pfeile an den Kanten }
0000300         { Laenge der abgerundeten Kantenecken }
0000300         { Breite des Kantenaktivierungsbereiches }
0001200         { Gittergroesse }

"FALSE"         { Gitter aktiv JA/NEIN   }
"TRUE"          { Gitter sichtbar JA/NEIN }
"FALSE"         { Abgerundete Ecken JA/NEIN }
"TRUE"          { Symbolnamen zentriert JA/NEIN }
"FALSE"         { Kanten rechtwinklig JA/NEIN }

0000000         { Farbe Menue Hintergrund }
0000009         { Farbe Menue Rahmen }
0000014         { Farbe Menue Ueberschrift }
0000015         { Farbe Menue Eintrag aktiv }
0000007         { Farbe Menue Eintag nicht aktiv }
0000010         { Farbe Eingabe-String }
0000015         { Farbe Confirm Text }
0000012         { Farbe Confirm Auswahlpunkt JA }
0000015         { Farbe Confirm Auswahlpunkt NEIN }
0000014         { Farbe Confirm Rahmen }
0000001         { Farbe General Rahmen }
0000014         { Farbe General Projekt }
0000012         { Farbe General Projektname }
0000014         { Farbe General Level }
0000012         { Farbe General Level name }
0000012         { Farbe Stelle }
0000014         { Farbe Stellenname }
0000002         { Farbe Transition }
0000014         { Farbe Transitionsname }
0000009         { Farbe Kante }
0000014         { Farbe Kantename }
0000007         { Farbe Gitter }
0000007         { Farbe Hintergrundtext Rahmen }
0000015         { Farbe Hintergrundtext }
"END"           { Ende des INIT-Teiles }
```

Bild 4.2.1: Struktur einer Konfigurationsdatei

```
PNE-FILE { Namenskennung }
"END" { Ende der Namenskennung }
INIT { Beginn des INIT-Teiles }
Test-Datei { Name des Petrinetzes }
10.07.90 { Datum des Petrinetzes }
0004000 { Breite der Transitionen }
0006000 { Hoehe der Transitionen }
0002000 { Radius der Stellen }
0000300 { Radius der Verbindungspunkte }
0005236 { Winkel der Pfeile an den Kanten }
0000800 { Laenge der Pfeile an den Kanten }
0000300 { Laenge der abgerundeten Kantenecken }
0000300 { Breite des Kantenaktivierungsbereiches }
0001200 { Gittergroesse }
"FALSE" { Gitter aktiv JA/NEIN }
"TRUE" { Gitter sichtbar JA/NEIN }
"FALSE" { Abgerundete Ecken JA/NEIN }
"TRUE" { Symbolnamen zentriert JA/NEIN }
"FALSE" { Kanten rechtwinklig JA/NEIN }
0000000 { Farbe Menue Hintergrund }
0000009 { Farbe Menue Rahmen }
0000014 { Farbe Menue Ueberschrift }
0000015 { Farbe Menue Eintrag aktiv }
0000007 { Farbe Menue Eintrag nicht aktiv }
0000010 { Farbe Eingabe-String }
0000015 { Farbe Confirm Text }
0000012 { Farbe Confirm Auswahlpunkt JA }
0000015 { Farbe Confirm Auswahlpunkt NEIN }
0000014 { Farbe Confirm Rahmen }
0000001 { Farbe General Rahmen }
0000014 { Farbe General Projekt }
0000012 { Farbe General Projektname }
0000014 { Farbe General Level }
0000012 { Farbe General Level name }
0000012 { Farbe Stelle }
0000014 { Farbe Stellenname }
0000002 { Farbe Transition }
0000014 { Farbe Transitionsname }
0000009 { Farbe Kante }
0000014 { Farbe Kantename }
0000007 { Farbe Gitter }
0000007 { Farbe Hintergrundtext Rahmen }
0000015 { Farbe Hintergrundtext }
0000001 { Anzahl Zeilen bei dem Setup-Text }
(*Dies ist der Setup text*) { Setup-Text }
"END" { Ende des INIT-Teiles }
PLACE { Beginn des Stellen-Teiles }
0000001 { Nummer der Stelle }
0008400 { X-Position der Stelle }
0011802 { Y-Position der Stelle }
0002000 { Radius der Stelle }
0000000 { Textposition der Stelle }
Place 0 { Name der Stelle }
"TRUE" { Textteil zu Type angelegt JA/NEIN }
0000001 { Anzahl der Zeilen von Type }
(* Dies ist der Placetext *) { Text zu Type }
"END" { Ende des Stellen-Teiles }
TRANSITION { Beginn des Transitionen-Teiles }
0000001 { Nummer der Transition }
0022067 { X-Position der Transition }
0011796 { Y-Position der Transition }
0004000 { Breite der Transition }
0006000 { Hoehe der Transition }
0000000 { Textposition der Transition }
Transition 0 { Name der Transition }
"FALSE" { Textteil zu Condition angelegt JA/NEIN }
"FALSE" { Textteil zu Deklay angelegt JA/NEIN }
"FALSE" { Textteil zu Procedure angelegt JA/NEIN }
"END" { Ende des Transitionen-Teiles }
DATA-FLOW { Beginn des Kanten-Teiles }
0000001 { Nummer der Kante }
0000001 { Nummer des Verbindungspunktes }
0010494 { X0-Position des Verbindungspunktes }
0011801 { Y0-Position des Verbindungspunktes }
0020016 { X1-Position des Verbindungspunktes }
0013797 { Y1-Position des Verbindungspunktes }
"END" { Ende der Verbindungspunkte }
0000001 { Nummer des letzten Verbindungspunktes }
"TRUE" { Kante von Stelle nach Transition JA/NEIN }
0000001 { Nummer der verbundenen Stelle }
0000001 { Nummer der verbundenen Transition }
0019323 { X0-Position des Kantenpfeiles }
001359 { Y0-Position des Kantenpfeiles }
001323 { X1-Position des Kantenpfeiles }
001235 { Y1-Position des Kantenpfeiles }
"END" { Ende des Kanten-Teiles }
BACKGROUNDTEXT { Beginn des Hintergrundtext-Teiles }
0000001 { Nummer des Hintergrundtextes }
0011725 { X-Position des Hintergrundtextes }
0019474 { Y-Position des Hintergrundtextes }
0000013 { Breite des Hintergrundtextes }
0000001 { Hoehe des Hintergrundtextes }
Bei spiel -text { Hintergrundtext }
"END" { Ende des Hintergrundtext-Teiles }
```

Bild 4.2.2: Struktur einer Petrinetzdatei

4.3. Globale Variablen und Typen

Da das Programm aus mehreren Teilen besteht, muß eine Möglichkeit für den Informationsaustausch der Teile untereinander gefunden werden. Ein Lösungsansatz ist die Verwendung von langen Parameter-teilen bei den Prozedurköpfen. Dies hat jedoch den Nachteil, daß der Ablauf erheblich verzögert und die Übersicht verschlechtert wird. Bei der vorliegenden Implementation wurde daher dem Einsatz globaler Variablen der Vorzug gegeben. Jede Prozedur und Funktion kann bei Bedarf auf diese zugreifen.

Die Menüverwaltung erfolgt über insgesamt sechs Menürecords für das General-Menü, das Background-Menü, das Place-Menü, das Transition-Menü, das Dataflow-Menü sowie das Backgroundtext-Menü. Die Records enthalten Informationen der Menüs über die jeweils aktuelle Bildschirmposition beim Öffnen, über die Anzahl der Einträge, über die Namen, Verfügbarkeit und Tokennummer der Einträge sowie über Überschrift des Menüs und die gewählte Eintragsnummer (siehe Bild 4.3.1). Mit der booleschen Variablen "EntryActive" wird bestimmt, ob ein Eintrag im Augenblick anwählbar ist oder nicht. Dies ist beispielsweise bei der "Undelete"-Funktion von Bedeutung, da diese erst gültig wird, falls vorher ein Symbol gelöscht wurde. In der Variablen "Choice" wird die Eintragsnummer abgelegt, wenn ein gültiger Eintrag aus dem Menü ausgewählt wurde; im Falle, daß kein Eintrag ausgewählt wird, erhält "Choice" den Wert 255.

Alle Menürecords werden unmittelbar nach dem Programmstart initialisiert. Während des Programmverlaufes werden einige Eintragsnamen ("Grid on/off") und deren Verfügbarkeit ("Undelete") geändert.

Angaben über die Breite und Höhe des jeweiligen Menüs sind in den Menürecords nicht unmittelbar abgespeichert. Diese Angaben werden von den Prozeduren, welche die Menüs verwalten, an Hand der Anzahl der Einträge und durch Bestimmung der Länge des längsten Menüeintrages jedesmal neu ermittelt. Dies bietet den Vorteil, daß bei einer Erweiterung des Menüs der Programmierer nur den Eintragsnamen hinzufügen und die Eintragsnummer um eins erhöhen muß. Es braucht keine neue Breite und Höhe berechnet zu werden, was ansonsten sehr aufwendig wäre (siehe auch Beispiel in Kapitel 6.1).

Type

```
MenueRec      = Record                { Menue-Record          }
              Headline : String;      { Ueberschrift        }
              Number   : LongInt;     { Anzahl der Eintraege }
              EntryName : Array [0..MenueMaxNumber] of String;
                                      { Eintragnamen        }
              EntryActiv : Array [0..MenueMaxNumber] of Boolean;
                                      { Eintrag aktiv      }
              EntryToken : Array [0..MenueMaxNumber] of TokenType;
                                      { Tokennummer des Eintrages }
              End; {Record}
MenuePtr      = ^MenueRec;
```

Var

```
MenueR      : Record                { MenueListe          }
              MenuePointer : MenuePtr; { Zeiger auf Menue    }
              X            : LongInt;  { X-Position         }
              Y            : LongInt;  { Y-Position         }
              Choice       : LongInt;  { Eintragsnummer     }
              End; {Record}

BackgroundMenue : MenuePtr; { Pointer auf Hintergrund-Menue }
General Menue   : MenuePtr; { Pointer auf General . Menue   }
PlaceMenue     : MenuePtr; { Pointer auf Stellen-Menue     }
TransitionMenue : MenuePtr; { Pointer auf Transitionen-Menue }
DataFlowMenue  : MenuePtr; { Pointer auf Kanten-Menue     }
BackgroundTextMenue : MenuePtr; { Pointer auf Hintergrundtext-Menue }
```

Bild 4.3.1: Die Menürecords

Stellen, Transitionen, Kanten und Hintergrundtexte werden mittels einfach verketteter Listen verwaltet (/WIRTS83/). Jede Liste hat hierbei eine Wurzel, welche auf den Beginn der jeweiligen Liste zeigt.

Alle Listeneinträge bestehen aus einem Record, welche eine Nummer zur Identifikation des Eintrages, dessen Position, einen Pointer auf den nächsten Eintrag (zeigt auf NIL im Falle keines weiteren Eintrags) sowie einige spezielle Daten enthält (siehe Bild 4.3.2).

Die Stellenrecords enthalten weiterhin den Stellennamen, die Namensposition, den Radius sowie einen Pointer auf ein Textarray.

Type

```

Textprimitiv      = Array [0..TextPrimitivXMax, 0..TextPrimitivYMax] of Byte;
TextPrimPtr      = ^Textprimitiv;

PlaceRec         = Record
    Number       : LongInt;
    Next         : PlacePtr;
    Xpos         : LongInt;
    Ypos         : LongInt;
    Radius       : LongInt;
    Textpos      : Byte;
    Text         : TextString;
    PlType       : TextPrimPtr;
End; { Record }

PlacePtr         = ^PlaceRec;

TransitionRec    = Record
    Number       : LongInt;
    Next         : TransitionPtr;
    Xpos         : LongInt;
    Ypos         : LongInt;
    Xlength      : LongInt;
    Ylength      : LongInt;
    Textpos      : Byte;
    Text         : TextString;
    TrCondition  : TextPrimPtr;
    TrDelay      : TextPrimPtr;
    TrProcedure  : TextPrimPtr;
End; { Record }

TransitionPtr    = ^TransitionRec;

DataFlowRec     = Record
    Number       : LongInt;
    Next         : DataFlowPtr;
    Link         : DFPointPtr;
    LastLink     : LongInt;
    PlToTr       : Boolean;
    PlPtr        : PlacePtr;
    TrPtr        : TransitionPtr;
    ArrowX0      : LongInt;
    ArrowY0      : LongInt;
    ArrowX1      : LongInt;
    ArrowY1      : LongInt;
End; { Record }

DataFlowPtr     = ^DataFlowRec;

DFPointRec      = Record
    Number       : LongInt;
    Next         : DFPointPtr;
    X0           : LongInt;
    Y0           : LongInt;
    X1           : LongInt;
    Y1           : LongInt;
End; { Record }

BackgroundTextRec = Record
    Number       : LongInt;
    Next         : BackgroundTextPtr;
    Xpos         : LongInt;
    Ypos         : LongInt;
    Xlength      : LongInt;
    Ylength      : LongInt;
    Text         : TextPrimPtr;
End; { Record }

BackgroundTextPtr = ^BackgroundTextRec;

```

Var

```

PlaceRoot       : PlacePtr;
TransitionRoot  : TransitionPtr;
DataFlowRoot    : DataFlowPtr;
BackgroundTextRoot : BackgroundTextPtr;

```

Bild 4.3.2: Die verketteten Listen

Dieser Textpointer zeigt auf NIL, wenn kein Text unter "Place type" eingegeben wurde. Sobald dieser Menüpunkt angewählt wird, erfolgt die Generierung eines entsprechenden Textarray auf dem Heap und die Zuweisung an den Pointer. In ihm wird der Text des Benutzers abgelegt und nach erfolgreicher Eingabe auf syntaktische Richtigkeit bezüglich der Grammatik der NSL-Sprache überprüft.

Für Transitionen sind anstelle des Radius, Breite und Höhe vorgesehen sowie drei Textpointer für "Transition condition", "Transition delay" und "Transition procedure", welche ebenfalls nach Eingabe geparkt werden.

Bei den Kanten entfallen derartige Textarrays, dafür sind die Koordinaten der Pfeilspitze, Pointer auf die damit verbundene Stelle und Transition sowie eine einfach verkettete Liste mit allen Verbindungspunkten und deren Anzahl vorhanden. Die Boolesche Variable "PI ToTr" legt die Richtung der Kante fest. Hat sie den Wert "true", läuft die Kante von einer Stelle zu einer Transition, ansonsten umgekehrt.

Hintergrundtextrecords beinhalten außerdem die Breite und Höhe des Rahmens, sowie einen Pointer auf das entsprechende Textarray.

Um das Programm den Wünschen des Anwenders gegenüber flexibel zu halten, lassen sich die Farben für die Menüs, Rahmen, Texte und so weiter beliebig ändern (siehe Bild 4.3.3). Diese Daten werden zusammen mit anderen beim Start des Programmes mittels einer Konfigurationsdatei geladen und bei Beendigung wieder gespeichert. Wird ein Petrinetz geladen oder gesichert, werden diese Daten ebenfalls mitübertragen.

ColorMB	: LongInt;	{ Farbe Menue-Background	}
ColorMR	: LongInt;	{ Farbe Menue-Kamen	}
ColorMH	: LongInt;	{ Farbe Menue-Headline	}
ColorMEA	: LongInt;	{ Farbe Menue-Eintrag aktiv	}
ColorMEN	: LongInt;	{ Farbe Menue-Eintrag nicht aktiv	}
ColorCT	: LongInt;	{ Farbe Confi rm-Ausgabe-Text	}
ColorCB	: LongInt;	{ Farbe Confi rm-Button-Text	}
ColorCI B	: LongInt;	{ Farbe Confi rm-Invers-Button-Text	}
ColorCR	: LongInt;	{ Farbe Confi rm-Button-Ramen	}
ColorP	: LongInt;	{ Farbe Place	}
ColorPT	: LongInt;	{ Farbe Place-Text	}
ColorT	: LongInt;	{ Farbe Transi ti on	}
ColorTT	: LongInt;	{ Farbe Transi ti on-Text	}
ColorDF	: LongInt;	{ Farbe Data-Fl ow	}
ColorDFT	: LongInt;	{ Farbe Data-Fl ow-Text	}
ColorGRI D	: LongInt;	{ Farbe Gi tter	}
ColorBTF	: LongInt;	{ Farbe BackgroundText-Ramen	}
ColorBTT	: LongInt;	{ Farbe BackgroundText-Text	}

Bild 4.3.3: Die Variablen für die Farben

Wenn ein Confirm-Fenster eröffnet wird, werden die Daten des "YES"- und "NO"-Auswahlpunktes in einem Confirm-Record abgelegt, so daß der Scanner die Mausposition mit den entsprechenden Koordinaten vergleichen und die Auswahlpunkte gegebenenfalls hervorheben kann (siehe Bild 4.3.4). Die Daten stellen die linke obere Ecke ("X00";"Y00") beziehungsweise ("X10";"Y10") sowie die rechte untere Ecke ("X01";"Y01") beziehungsweise ("X11";"Y11") der Auswahlpunkte dar.

```
ConfirmR : Record
    X00    : LongInt;      { linke obere Ecke Taste YES      }
    Y00    : LongInt;
    X01    : LongInt;      { rechte untere Ecke Taste YES      }
    Y01    : LongInt;
    X10    : LongInt;      { linke obere Ecke Taste NO        }
    Y10    : LongInt;
    X11    : LongInt;      { rechte untere Ecke Taste NO        }
    Y11    : LongInt;
    Choice : LongInt      { Variable fuer Ergebnis: YES/NO    }
End; { Record }
```

Bild 4.3.4: Das Confirm-Record

Damit das Löschen von Objekten wieder rückgängig gemacht werden kann, wird ein Undelete-Record geführt (siehe Bild 4.3.5). Beim Löschen eines Objektes wird das Undelete-Record gelöscht, das Objekt aus der jeweiligen verketteten Liste entfernt und in die entsprechende Komponente des Undelete-Records eingetragen. Wird daraufhin die "Undelete"-Funktion aufgerufen, wird das Objekt wieder in die entsprechende Liste eingehängt. Voraussetzung hierfür ist, daß vor diesem Aufruf kein anderes Objekt gelöscht wurde. Ansonsten werden die Daten des ersten Objektes gelöscht und die Daten des zweiten Objektes in das Undelete-Record aufgenommen.

```
UndeleteR : Record
    Pt      : Boolean;      { Stellen aktiv                      }
    PIPtr   : PPlacePtr;    { Stellen-Pointer                    }
    Tr      : Boolean;      { Transition aktiv                  }
    TrPtr   : TransitionPtr; { Transitionen-Pointer              }
    OF      : Boolean;      { Kante aktiv                       }
    DFPtr   : DataFlowPtr;  { Kanten-Pointer                   }
    DFP     : Boolean;      { Verbindungspunkt aktiv           }
    OFPPtr  : OFPointPtr;   { Verbindungspunkt-Pointer         }
    BT      : Boolean;      { Hintergrundtext aktiv            }
    BTPtr   : BackgroundTextPtr; { Hintergrundtext-Pointer          }
End; { Record }
```

Bild 4.3.5: Das Undelete-Record

Für den Bereich der Benutzereingabe sind weitere globale Variablen vorgesehen, insbesondere für die Mausposition ("MouseX"; "MouseY") und den Status der Maustasten ("MouseButton0", "MouseButton1") sowie die Position des Cursors innerhalb des Editorfensters ("ScannerXEdit"; "ScannerYEdit"). Diese Variablen werden vom Scanner und seinen Prozeduren gesetzt und verwaltet (siehe Bild 4.3.6).

Var

```
KeyboardStatus      : LongInt;      { 0: keine Taste, 1: ASCII, 2: F-Taste }
KeyboardVal ue     : Byte;          { ASCII-Wert }
KeyboardFVal ue    : Byte;          { Funktionstasten-Wert }
KeyboardStrVal ue  : String;        { eingegebener String }
KeyboardStrLength  : Byte;          { Laenge des eingegebenen Strings }
MouseMaskStyl e    : LongInt;      { Maus-Cursor exodieren o. Ueberm. }
MouseX              : LongInt;      { Maus-X-Koordinate (0-4095) }
MouseY              : LongInt;      { Maus-Y-Koordinate (0-4095) }
MouseButton0       : Boolean;      { linke Maustaste gedrueckt }
MouseButton1       : Boolean;      { rechte Maustaste gedrueckt }
ScannerXEdit       : LongInt;      { Texteditor X-Position }
ScannerYEdit       : LongInt;      { Texteditor Y-Position }
ScannerXOldEdit    : LongInt;      { Texteditor alte X-Position }
ScannerYOldEdit    : LongInt;      { Texteditor alte Y-Position }
ParseChr           : Byte;          { Chr fuer Parser }
LookAhead          : Byte;          { LookAheadSymbol fuer Parser }
```

Bild 4.3.6: Variablen des Scannerteils

Zur Skalierung und Verwaltung des Bildschirms existieren einige Konstanten und Variablen (siehe Bild 4.3.7). Die Größe in X-Richtung wird durch "ScreenXSize", die in Y-Richtung durch "ScreenYSize" bestimmt. Das Verhältnis dieser logischen Größen zu der tatsächlichen physikalischen Auflösung des aktuellen Bildschirmadapters wird zu Anfang einmal ermittelt und in "XFactor" und "YFactor" gespeichert. Die physikalische Höhe und Breite eines Buchstaben befindet sich in "LetterXR" und "LetterYR", die logische Höhe und Breite in "LetterX" und "LetterY". Alle Positionsangaben innerhalb des Programmen beziehen sich immer auf logische Koordinaten. Das hat zur Folge, daß für das Programm der Bildschirm eine Größe von "ScreenXSize" * "ScreenYSize" hat. Diese Werte werden dann vor dem Zeichnen in die entsprechenden physikalischen Koordinaten umgerechnet.

Zur Anpassung der unterschiedlichen Verhältnisse von Breite und Höhe ist zum Ausgleich die Variable "YAspect" vorgesehen. Dadurch erscheinen Kreise nicht oval, sondern rund auf dem Bildschirm.

Const

ScreenXSi ze	= \$8000;	{ Logi sche Brei te des Bi ldschi rms	}
ScreenYSi ze	= \$8000;	{ Logi sche Hoehe des Bi ldschi rms	}
LetterXR	= 8;	{ physi kal i sche Brei te ei nes Buchstaben	}
LetterYR	= 10;	{ physi kal i sche Hoehe ei nes Buchstaben	}
Wi ndowMax	= 3;	{ maxi mal Anzahl geoeffneter Fenster	}
MNormal	= \$00;	{ Schri ftart: Normal	}
MBol d	= \$01;	{ Schri ftart: Fett	}
MWeak	= \$02;	{ Schri ftart: Fei n	}
MI nversNormal	= \$10;	{ Schri ftart: I nvers-Normal	}
MI nversBol d	= \$11;	{ Schri ftart: I nvers-Fett	}
MI nversWeak	= \$12;	{ Schri ftart: I nvers-Fei n	}

Var

Yaspect	: Real ;	{ Verhael tni sfaktor fuer Graphi kkarte	}
ScreenXReal Si ze	: Longl nt;	{ physi kal i sche Laenge	}
ScreenYReal Si ze	: Longl nt;	{ physi kal i sche Hoehe	}
Xfactor	: Real ;	{ Brei ten-Verhael tni s	}
Yfactor	: Real ;	{ Hoehen-Verhael tni s	}
LetterX	: Longl nt;	{ physi kal i sche Brei te ei nes Buchstabens	}
LetterY	: Longl nt;	{ physi kal i sche Hoehe ei nes Buchstabens	}
Wi ndowCounter	: Longl nt;	{ Fenster-Zaehl er	}

Bild 4.3.7: Konstanten und Variablen des Bildschirmteils

Alle aufgeführten Konstanten und Variablen können im Programm an beliebiger Stelle abgefragt werden.

4.4. Beschreibung der einzelnen Funktionsgruppen

4.4.1. Scanner

Der Scanner ist die Schnittstelle zwischen dem Benutzer und dem Parser. Er bildet die erste Phase des Editors. Seine Aufgabe ist die Umwandlung der Benutzereingaben in Token, welche dann vom Parser weiter verarbeitet werden. Dabei ist es sinnvoll, einen Teil der lexikalischen Analyse dem Scanner zu übertragen.

Hiervon wird beispielsweise bei der Verwaltung der Menüs Gebrauch gemacht, wo der Scanner die Programmsteuerung vom Öffnen bis zum Schließen des Menüs übernimmt.

Aufgrund seiner Funktion wird er häufig aufgerufen, wodurch die Effizienz des Programmes stark von ihm abhängt (vergleiche /WIRT86/ und /AHOS88/).

Der Scanner ist als Funktion "YyLex (Var YyLVal : LongInt) : LongInt" aufgebaut und übergibt das Token als Funktionswert an die aufrufende Prozedur zurück; weiterhin ist die Möglichkeit vorgesehen, den Interpretationswert des Tokens mittels "YyLVal " zurückzuliefern.

Um den verschiedenen Programmzuständen und Eingabemöglichkeiten gerecht zu werden, kann der Scanner in mehrere Zustände gesetzt werden, in welchen die Eingaben entsprechend unterschiedlich interpretiert werden (siehe Bild 4.4.1.1).

Const

StatusCancel	= 0;	{ Verwerfen einer Aktion	}
StatusOuit0	= 1;	{ Programm verlassen 1	}
StatusOuit1	= 2;	{ Programm verlassen 2	}
StatusTrack	= 3;	{ Mausbewegung verfolgen	}
StatusMenue	= 4;	{ Menue verwalten	}
StatusConfirm	= 5;	{ Confirm-Window verwalten	}
StatusNormal	= 6;	{ normale Eingabe	}
StatusTextParse	= 7;	{ Text parsen	}

Bild 4.4.1.1: Die Zustände des Scanners

In Bild 4.4.1.2 sind die unterschiedlichen Eingabemöglichkeiten des Scanners aufgezeichnet.

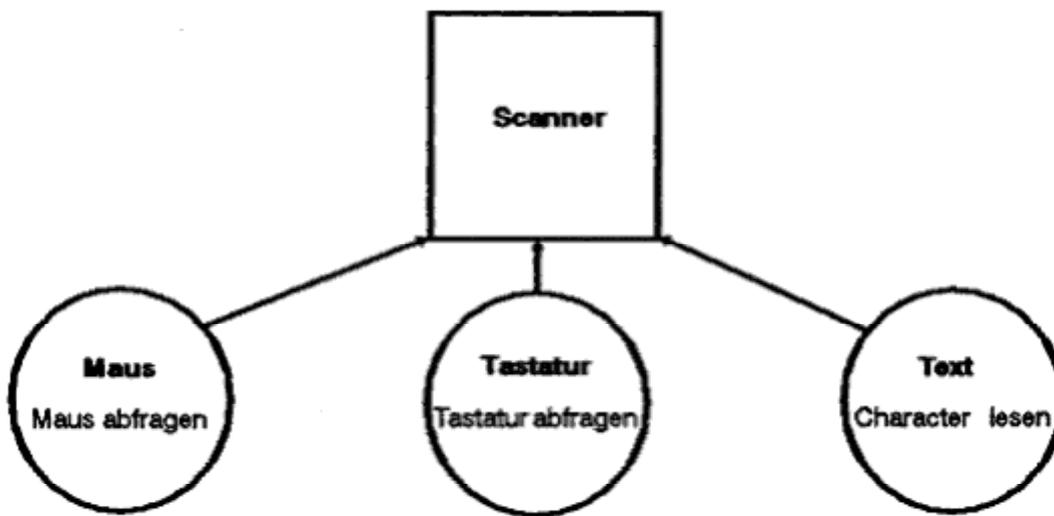


Bild 4.4.1.2: Die möglichen Eingabearten des Scanners

Die verschiedenen Zustände des Scanners werden durch den Parser und seine Prozeduren gesetzt und der Scanner danach aufgerufen.

Der Grundzustand des Scanners ist "StatusNormal". Er dient der Eingabe von Symbolen und dem Öffnen von Menüs. In diesem Zustand wartet der Scanner zunächst, bis die Maustaste losgelassen wurde, um einen definierten Ausgangszustand zu erhalten. Alsdann erfolgt eine Eingabeschleife, welche durch das Drücken der Maustaste beendet wird.

Wurde die Maustaste in der oberen Kopfzeile gedrückt, erhält die Funktion den Wert "_GENERAL" und wird beendet. Im anderen Fall wird nacheinander geprüft, ob sich die Position der Maus auf einer Stelle, einer Transition, einer Kante oder einem Hintergrundtext befindet. Dies geschieht, indem sukzessive alle Pointer der entsprechenden Listen von der Wurzel an durchlaufen werden. Hierbei werden die aktuelle Mausposition und die Umrandungen der Symbole mittels einer Prozedur zur Positionsprüfung verglichen. Liegt ein Symbol auf der Mausposition, wird je nach Objektart als Token "_PLACE", "_TRANSITION", "_DATAFLOW", "_DATAFLOWPOINT" oder "_BACKGROUNDTEXT" ausgegeben. Falls kein entsprechendes Objekt gefunden werden kann, befindet sich die Position der Maus auf dem Hintergrund, und es wird als Token "_BACKGROUND" ausgegeben.

Der Menüverwaltung dient der Zustand "StatusMenue", auf welchen unmittelbar nach Öffnung eines Menüs geschaltet wird. Hierbei wird zunächst mittels eines zuvor gesetzten Menü-Records die Position des Menüfensters, dessen Höhe und Breite sowie die Position seiner Einträge berechnet. Nun erfolgt eine Eingabeschleife, welche laufend die Mausposition verfolgt, bis die Maustaste losgelassen wird.

Befindet sich die Mausposition auf einem Eintrag, wird dieser, sofern sein Menüeintrag "MenueR. EntryActiv" auf "true" gesetzt ist, hell dargestellt. Im Falle, daß sein Menüeintrag auf "false" gesetzt ist, erfolgt keine Hervorhebung. Wenn sich die Mausposition auf keinem Eintrag befindet oder den Eintrag wechselt, so wird der alte Eintrag wieder in normaler Schrift dargestellt.

Wird die Maustaste auf einem Menüeintrag losgelassen, so erfolgt die Ausgabe des Tokens gemäß dem Wert des entsprechenden Menürecords "MenueR. EntryToken". Die Menüeintragsnummer wird in "MenueR. Choice" gespeichert. Befand sich der Mauszeiger beim Loslassen auf keinem Eintrag, wird "_MENUENOTOK" ausgegeben.

Um ein Confirm-Window zu verwalten, wird der Scanner in den Zustand "StatusConfirm" gesetzt. Hier wird die Mausposition mit den Daten der Auswahlpunkte im Confirm-Record verglichen. Befindet sich der Mauszeiger über einem Auswahlpunkt, wird dieser hervorgehoben. Betätigt man die Maustaste, wird je nach Auswahlpunkt "_YES" oder "_NO" zurückgegeben.

Um Objekte verschieben zu können, gibt es den Status "StatusTrack". In diesem wird lediglich die Mausposition ("MouseX"; "MouseY") laufend aktualisiert. Wird die Maustaste gedrückt, wird als Token "_MOUSEBUTTON" übergeben, ansonsten "_MOUSETRACK".

Der Scanner ist auch in der Lage, Texte zu scannen. Hierfür wird er in den Status "StatusTextParse" geschaltet. Nun erfolgt die zu scannende Eingabe nicht mehr über die Tastatur oder die Maus, sondern über das Textarray PageBuffer.

Die aktuelle Position im Textarray wird mittels ("ScannerXEdit"; "ScannerYEdit") festgelegt. Zeichen für Zeichen wird das Array gelesen und entsprechend ausgewertet. Wurde ein Wort erkannt, wird das dazugehörige Token zurückgeliefert.

Ist das Ende des Textarrays erreicht, wird "_ENDPARSE" übergeben.

Um dem Parser anzuzeigen, daß ein Regelzweig an einer bestimmten Stelle abgebrochen werden soll, kann der Scanner hierfür das Token "_CANCEL" übergeben. Dies ist zum Beispiel beim Zeichnen einer Kante der Fall, wenn diese während des Zeichnens verworfen werden soll.

Der Scanner ist dafür in den Zustand "StatusCancel " zu schalten.

Die letzten Zustände des Scanners sind "StatusQuit0" und "StatusQuit1", welche angenommen werden, wenn das Programm beendet werden soll. Sie dienen dazu, dem Parser das ordnungsgemäße Ende des Programms anzuzeigen.

Hierbei setzt sich der Scanner im Zustand "StatusQuit0" beim nächsten Aufruf selbst in den Zustand "StatusQuit1". Dies hat zur Folge, daß im Zustand "StatusQuit1" das Token "0" übergeben wird, um den Parser zum Beenden des Programms zu veranlassen.

4.4.2. Parser

Der Parser besteht aus einer Anzahl von Regeln, welche das Verhalten des Programmes in Abhängigkeit der vom Scanner gelieferten Token beschreiben.

Für den Fall, daß der Benutzer auf dem Hintergrund die Maustaste betätigt, sind in Bild 4.4.2.1 die dazugehörigen Regeln aufgeführt.

```
BackgroundExec : _ADDPL      AddPlOp      /* Hinzufuegen einer Stelle */
                 | _ADDTR      AddTrOp      /* Hinzufuegen einer Transition */
                 | _ADDTEXT    AddTextOp     /* Hinzufuegen eines Textes */
                 | _SETUP      SetupOp        /* Erstellen des Setup-Teiles */
                 | _MENUNOTOK  /* Kei n Menuepunkt angewaehlt */

                 {
                   Begin { BackgroundExec }
                     MenuClose (BackgroundMenue, MouseXMenue, MouseYMenue)
                   End; { BackgroundExec }
                 }
;

```

Bild 4.4.2.1: Die Regeln für das Anwählen auf den Hintergrund

Insgesamt ist der Parser in sechs Teile untergliedert. Der erste Teil besteht aus den möglichen Terminalsymbolen, welche später als Token vom Scanner geliefert werden. Alle weiteren Teile bestehen aus den eigentlichen Regeln: Eingabeschleife, Menüteil, Menüeintragsteil, Abarbeitung der einzelnen Menüpunkte und der Syntax für die Texteingaben.

Die Eingabeschleife, in Bild 4.4.2.2 zu erkennen, beschreibt den Start- und Grundzustand des Programmes. Sobald der Scanner ein Token von "_GENERAL" bis "_BACKGROUNDTEXT" liefert, erfolgt eine Verzweigung in den Menüteil. Hier wird das entsprechende Menü eröffnet und in den Menüeintragsteil übergegangen, in dem auf das nächste Token gewartet wird. Dieses Token gibt den Menüeintrag an, welchen der Benutzer aus dem Menü gewählt hat. Nun erfolgt die Abarbeitung des Menüeintrages.

```
PNE      : InputLoop Exit      /* Startzustand      */
          ;

InputLoop : InputLoop Input    /* Eingabeschleife    */
          | Input
          ;

Exit      : _ENDACTIVATED      /* Programm verlassen */
          {
            Begin { Exit }
            End; { Exit }
          }
          ;

Input     : _GENERAL          General Menue /* General -Menue    */
          | _BACKGROUND      BackgroundMenue /* Hintergrund-Menue */
          | _PLACE           PlaceMenue /* Stellen-Menue     */
          | _TRANSITION      TransitionMenue /* Transitionen Menue */
          | _DATAFLOW        DataFlowMenue /* Kanten-Menue     */
          | _DATAFLOWPOINT   DataFlowPointMenue /* Verbindungspunkt-Menue */
          | _BACKGROUNDTEXT  BackgroundTextMenue /* Hintergrundtext-Menue */
          ;
```

Bild 4.4.2.2: Die Eingabeschleife

Eine besondere Bedeutung kommt dem letzten Teil zu, der Syntax der Texteingaben. Hier sind Regeln von der NSL-Sprache aufgeführt. Für den Stellentyp sind sie in Bild 4.4.2.3 wiedergegeben. Der Einsprung erfolgt bei der Regel "typeblock", nachdem der Scanner in den Zustand "StatusTextParse" geschaltet wurde.

```
typebtock : _CAPACI TY capaci tyconstant ';'
           | _CAPACI TY capaci tyconstant _OF pl acetype ';'
           | capaci tyconstant _OF pl acetype ';'
           | pl acetype ';'
;

capaci tyconstant : unsi gnedi nteger
                  | constanti denti fi er
;

pl acetype : type
           | _DDT
;

unsi gnedi nteger : _DI GI TSEQUENCE
;

constanti denti fi er : _I DENTI FI ER
;

type : si mpl etype
     | structuredtype
     | poi ntertype
;

si mpl etype : scal artype
            | subrangetype
            | typei denti fi er
            | stri ngtype
;

structuredtype : unpackedstructuredtype
              | _PACKED unpackedstructuredtype
;

poi ntertype : poi ntersymbol poi nteri denti fi er
;

scal artype : '(' scal ari denti fi er l i st ')'
;

subrangetype : rangeconstant _TWOPOI NTS rangeconstant
;

typei denti fi er : _I DENTI FI ER
                 | standardtype
;

stri ngtype : _STRI NG stri ngspeci fi cati on
;

unpackedstructuredtype : arraytype
                       | recordtype
                       | settype
                       | fi letype
;

poi ntersymbol : '^'
              | '@'
;

poi nteri denti fi er : _I DENTI FI ER
                    | standardtype
;

scal ari denti fi er l i st : scal ari denti fi er l i st comma scal ardecl arati oni denti fi er
                          | scal ardecl arati oni denti fi er
;

rangeconstant : unsi gnedi nteger
              | si gn unsi gnedi nteger
              | constanti denti fi er
              | si gn constanti denti fi er
;

standardtype : _BOOLEAN
             | _I NTEGER
             | _REAL
             | _CHAR
             | _TEXT
;

stri ngspeci fi cati on : '[' _DI GI TSEQUENCE ']'
                      | '(' _DI GI TSEQUENCE ')'
```

Bild 4.4.2.3: Die NSL-Grammatik für den Stellentyp

Falls bei der Parsung dieser Regeln ein Fehler auftritt, da der Benutzer einen syntaktisch falschen Text eingegeben hat, muß der Parser wieder synchronisiert werden. Dies ist gegeben, wenn der Text nicht den Regeln der NSL-Sprache entspricht. Die Programmabarbeitung erfolgt dann wieder an einer wohldefinierten Stelle. Die dazugehörige Synchronisationsregel für den Stellentyp ist in Bild 4.4.2.4 zu ersehen. Hierbei wird im Falle eines Syntaxfehlers in den Zweig "error" gesprungen. Dort wird die Fehlermeldung ausgegeben, und der Text kann neu verbessert werden. Am Ende erfolgt der Aufruf der Regel "TypePI Op0", welche für eine erneute Syntaxprüfung sorgt.

```
TypePI Op          :                               /* Fenster schließen          */
                  {
                    Begin { TypePI Op }
                    MenuClose (PlaceMenu, MouseXMenu, MouseYMenu);
                    TypePI Proc0 (0, 0);
                    End; { TypePI Op }
                }

TypePI Op0         : _YES                          /* YES angewählt          */
                  {
                    Begin { TypePI Op0 }
                    CloseConfirmWindow (MouseXOrg, MouseYOrg, 'Save edit?');
                    ScannerStatus := StatusTextParse;
                    ScannerXEdit := 0;
                    ScannerYEdit := 0;
                    ParseChr := Ord (' ');
                    LookAhead := ParseChr;
                    End; { TypePI Op0 }
                }
                | TypePI Op1
                {
                    Begin { TypePI Op0 }
                    TypePI Proc1;
                    ScannerStatus := StatusNormal;
                    End; { TypePI Op0 }
                }
                | _NO                               /* NO angewählt          */
                {
                    Begin { TypePI Op0 }
                    CloseConfirmWindow (MouseXOrg, MouseYOrg, 'Seve edit?');
                    End; { TypePI Op0 }
                }
                | error                             /* Fehlersynchronisation */
                {
                    Begin { TypePI Op0 }
                    yyerrflag := 0;
                    ScannerStatus := StatusNormal;
                    DisplayOKMessage (MouseXOrg, MouseYOrg, 'Place type', 'ERROR found!');
                    TypePI Proc1;
                    TypePI Proc0 (ScannerXEdit-1, ScannerYEdit);
                    yychar := Yyl ex (yyl val);
                    End; { TypePI Op0 }
                }
                ;

TypePI Op1         : PlaceType _ENDPARSE          /* Parsen bis zum Textende */
                  ;
```

Bild 4.4.2.4: Fehlersynchronisation für den Stellentyp

4.4.3. Bildschirmprozeduren

Die Bildschirmprozeduren sind zuständig für das Zeichnen der Objekte, die Verwaltung der Fenster, sowie für die Textein- und -ausgabe. Sie sind in der Unit "SCREEN" zu finden und in Bild 4.4.3 abgebildet.

Für die Fensterverwaltung sind die Prozeduren "SaveWindow", "ClearWindow", "MakeWindow" und "RestoreWindow" vorgesehen, welche den aktuellen Bildschirminhalt zwischenspeichern, das Fenster löschen, Window-Rahmen und Texte innerhalb des Fenster zeichnen sowie den alten Bildschirminhalt wieder zurückkopieren können. Damit kann ein Fenster an beliebiger Stelle gezeichnet und nach Gebrauch wieder geschlossen werden.

Menüs werden in den Prozeduren "MenuOpen" und "MenuClose" behandelt. "MenuOpen" sichert zuerst den alten Bildschirminhalt und gibt daraufhin das Menü mit allen Einträgen an der gewünschten Stelle aus. Das Menüfenster wird mit Hilfe von "MenuClose" wieder geschlossen.

Mit den Prozeduren "GrMove", "GrLine", "GrRectangle" und "GrCircle" werden der Graphikcursor gesetzt sowie Linien, Rechtecke und Kreise in einer angegebenen Farbe gezeichnet. Die Prozeduren "GrXORLine", "GrXORRectangle" und "GrXORCircle" exodieren die entsprechenden Objekte.

"GrWrite" gibt einen String in einer bestimmten Farbe an der jeweiligen Position aus. Dieser Prozedur wird auch ein Textattribut übergeben, welches das Aussehen der Buchstaben bestimmt. Unterstützt werden Normal-, Fett-, Invers- und Feinschrift.

Für die Objektverwaltung finden sich noch Prozeduren zum Zeichnen, Exodieren und Benennen von Stellen, Transitionen und Kanten. Für die Stellen lauten sie: "DrawPlace", "XorPlace" und "NamePlace".

Der Hintergrundtext wird mittels "DrawBackgroundText" ausgegeben und kann mit "XORBackgroundText" verschoben werden.

```
{ *** Fenster Funktionen *** }

Procedure ClearWindow (X0, Y0, X1, Y1 : LongInt);
Procedure RefreshWindow;
Procedure SaveWindow (X0, Y0, X1, Y1 : LongInt);
Procedure RestoreWindow (X0, Y0, X1, Y1 : LongInt);
Procedure MakeWindow (X0, Y0, XL, YL : LongInt; Headline : DatenString);

{ *** Elementare Zeichenfunktionen *** }

Procedure GrMove (X, Y : LongInt);
Procedure GrWrite (X, Y, Color, Mode : LongInt; S : DatenString);
Procedure GrLine (X0, Y0, X1, Y1, Color : LongInt);
Procedure GrXORLine (X0, Y0, X1, Y1, Color : LongInt);
Procedure GrRectangle (X0, Y0, X1, Y1, Color : LongInt);
Procedure GrXORRectangle (X0, Y0, X1, Y1, Color : LongInt);
Procedure GrCircle (X, Y, R, Color : LongInt);
Procedure GrXORCircle (X, Y, R, Color : LongInt);

{ *** Menueverwaltung *** }

Procedure MenueOpen (MenuePointer : MenuePtr; Var X, Y : LongInt);
Procedure MenueClose (MenuePointer : MenuePtr; X, Y : LongInt);

{ *** Fensterverwaltung *** }

Function InputString (X, Y : LongInt; Headline : DatenString; L : LongInt; Var S : DatenString) : Boolean;
Procedure OpenConfirmWindow (X : LongInt; Y : LongInt; Headline : String);
Procedure CloseConfirmWindow (X : LongInt; Y : LongInt; Headline : DatenString);
Procedure DisplayOKMessage (X : LongInt; Y : LongInt; Headline : String; Message : DatenString);
Procedure OpenDisplayMessage (X : LongInt; Y : LongInt; Headline : DatenString; Message : String);
Procedure CloseDisplayMessage (X : LongInt; Y : LongInt; Headline : DatenString; Message : String);
Procedure OpenDefaultWindow (X0, Y0, X1, Y1 : LongInt; Headline : String);
Procedure CloseDefaultWindow;
Function GetString (X, Y : LongInt; L : LongInt; Var S : DatenString) LongInt;

{ *** Funktionen fuer Stellen *** }

Procedure DrawPlace (P : PlacePtr);
Procedure XorPlace (P : PlacePtr);
Procedure NamePlace (P : PlacePtr);

{ *** Funktionen fuer Transitionen *** }

Procedure DrawTransition (P : TransitionPtr);
Procedure XorTransition (P : TransitionPtr);
Procedure NameTransition (P : TransitionPtr);

{ *** Funktionen fuer Kanten *** }

Procedure DrawDataFlow (P : DataFlowPtr);
Procedure XorFirstDataFlow (P : DataFlowPtr);
Procedure XorDataFlow (P : DataFlowPtr);
Procedure XorPtDataFlow (P : DataFlowPtr);
Procedure XorTrDataFlow (P : DataFlowPtr);
Procedure DrawDFPart (P : DFPointPtr);
Procedure XorDFPart (P : DFPointPtr);
Procedure DrawDFPoint (P : DFPointPtr);
Procedure XorDFPoint (P : DFPointPtr);

{ *** Funktionen fuer Hintergrundtexte *** }

Procedure DrawBackgroundText (P : BackgroundTextPtr);
Procedure XORBackgroundText (P : BackgroundTextPtr);
```

Bild 4.4.3: Die Bildschirmprozeduren

4.4.4. Objektprozeduren

Nach Anwahl eines bestimmten Menüpunktes bestimmt der Parser, welche Aktion als nächstes ausgeführt werden soll. Die dazugehörigen Objektprozeduren sind in ihre Menüs unterteilt. Dazu gehören die Units "GENERAL. PAS", "PLACE. PAS", "TRANS. PAS", "DATAFLOW. PAS" und "BGTEXT. PAS".

In jeder Unit finden sich zu den einzelnen Menüpunkten Prozeduren, die vom Parser aus gestartet werden. Nach der Ausführung der entsprechenden Aktion wartet der Parser auf ein weiteres Token vom Scanner, um die nächste Aktion zu ermitteln.

Weiterhin sind Funktionen vorhanden, um Objekte auf dem Heap zu erzeugen und die aktuelle Position mit allen Einträgen einer Objektliste zu vergleichen.

Für die Stellen sind diese Prozeduren in Bild 4.4.4.1 aufgeführt.

```
Procedure AddPI Proc;                { Stelle hinzufuegen    }
Procedure NewPlace (X, Y : LongInt; Var P : PlacePtr);
                                     { Stelle anlagen        }
Function TestInPlace (Var P : PlacePtr; X, Y : LongInt) : Boolean;
                                     { Position mit Stelle ver-
gl e i c h e n                }
Procedure SourcePtProc;              { Beginn einer Kante    }
Procedure HovePI Proc0;              { Stelle verschieben 0  }
Procedure MovePI Proc1;              { Stelle verschieben 1  }
Procedure HovePLProc2;              { Stelle verschieben 2  }
Procedure Resi zePI Proc0;           { Groesse veraendern 0  }
Procedure Resi zePI Proc1;           { Groesse veraendern 1  }
Procedure Resi zePI Proc2;           { Groesse veraendern 2  }
Procedure DefResi zeP(Proc;          { Radius auf Vorgebe    }
Procedure DeletePLProc;             { Stelle loeschen       }
Procedure RenamePI Proc;            { Stelle umbenennen     }
Procedure TakeAsDefaultPLProc;      { Radius als Vorgabe    }
Procedure TypeP1Proc0 (X : LongInt; Y: LongInt); { Texteingabe fuer Type }
Procedure TypePtProc1;              { Texteingabe fuer Type }
```

Bild 4.4.4.1: Die Objektprozeduren für Stellen

Als Beispiel für eine Objektprozedur sei hier in Bild 4.4.4.2 die Prozedur "RenamePI Proc" beschrieben.

Als erstes wird das "ChangeFlag" gesetzt, da an dem Petrinetz eine Veränderung vorgenommen wurde. Nun wird mit Hilfe der Funktion "TestInPlace" der Pointer "P" auf die Stelle unterhalb des Mauszeigers gesetzt. Nach Eingabe eines String durch Aufruf von "InputString" wird geprüft, ob es sich um einen Leerstring handelt. In diesem Falle wird die Stelle mit "Place n" benannt, wobei die kleinste natürliche Zahl n gewählt wird, so daß "Place n" eindeutig ist. Falls ein String ungleich einem Leerstring eingegeben wurde, wird geprüft, ob dieser bereits an eine andere Stelle vergeben ist. Falls ja, wird eine Fehlermeldung ausgegeben, und die Eingabe erfolgt erneut. Ansonsten wird der Name dem Stellen-Record zugewiesen. Danach wird das Bild mit "RefreshWindow" neu aufgebaut, um den neuen Namen anzuzeigen.

```
Procedure RenamePtProc;
{
  Stelle mit neuem Namen versehen
}

Label
  Marke1;

Var
  P : PPlacePtr;
  S : TextString;
  U : TextString;
  I : LongInt;
  B : Boolean;

Begin { RenamePLProc }
  ChangeFlag := true;           { Petrinetz veraendert }
  P := PPlaceRoot^.Next;      { aktuelle Stelle holen }
  If not TestInPlace (P, MouseXOrg, MouseYOrg) then Beep;
  S := P^.Text;                { Text holen }

Marke1:
  B := InputString (MouseXOrg, MouseYOrg, 'Rename place', TextStringLength, S);
  If B then
    Begin { then }
      If S = ' ' then
        Begin { then }
          I := 0;
          Repeat
            Str (I, U);
            S := 'Place ' + U;
            Inc (I);
          until not TestName (S)
          { S="PLACE n" }
        End { then }
      else
        Begin { else }
          While (UpperString(S) <> UpperString(P^.Text)) and TestName(S) do
            Begin { While }
              DisplayOKMessage (MouseX, MouseY, 'Rename place', 'The name "' + S + '" already exists!');
              Goto Marke1;
            End; { While }
          End; { else }
          P^.Text := S;
          RefreshWindow;
          { Text zuweisen }
          { neuen Namen anzeigen }
        End; { then }
      End; { RenamePI Proc }
```

Bild 4.4.4.2: Die Prozedur "RenamePIProc"

4.4.5. Datensicherungs- und -ladeprozeduren

Zum Laden und Sichern von Daten werden ASCII-Dateien verwendet. Das zugrunde liegende Dateiformat ist in Kapitel 4.2 spezifiziert. ASCII-Dateien bieten den Vorteil, daß sie unabhängig vom Rechnertyp und Pascal-Compiler verarbeitet werden können. Dabei werden sowohl Zahlen als auch boolesche Werte in Strings umgewandelt.

Bild 4.4.5.1 zeigt die für die Zahlenumwandlung zuständigen Funktionen "IntToStr" und "StrToInt".

Das Format eines Strings ist: "SNNNNNNN". Hierbei steht S für das Vorzeichen der Zahl ('-' für negative, ' ' für positive Zahlen) und N für eine Ziffer von 0 bis 9 (WI RT85/).

```

Function IntToStr (N : LongInt) : ASCIIIntStr;
{
Wandelt eine LongInt-Zahl in einen ASCII-String um
}
Var
S : ASCIIIntStr;
Begin { IntToStr }
If N >= 0 then { Vorzeichen " " }
Begin { then }
S := ' ';
End { then }
else
Begin { else } { Vorzeichen "-" }
S := '-';
N := Abs(N);
End; { else }
S := S +
Chr (N div 1000000 mod 10 + $30) + { Ziffern zusammensetzen }
Chr (N div 100000 mod 10 + $30) +
Chr (N div 10000 mod 10 + $30) +
Chr (N div 1000 mod 10 + $30) +
Chr (N div 100 mod 10 + $30) +
Chr (N div 10 mod 10 + $30) +
Chr (N div 1 mod 10 + $30);
IntToStr := S { Ergebnis uebergeben }
End; { IntToStr }

Function StrToInt (S : ASCIIIntStr) : LongInt;
{
Wandelt einen ASCII-String in eine LongInt-Zahl um
}
Var
N : Reat;
I : LongInt;
Begin { StrToInt }
S := Copy (S, 1, 8); { Zahl auf 8 Stellen setzen }
If Copy (S, 1, 1) = '-' then { Zahl negativ }
Begin { then }
N := -1;
S := Copy ('00000000', 1, 9-Length (S)) + Copy (S, 2, 7);
End { then }
else
Begin { else } { Zahl positiv }
N := 1;
If Copy (S, 1, 1) = ' ' then
S := Copy ('00000000', 1, 9-Length (S))+Copy (S, 2, 7)
else
S := Copy ('00000000', 1, 8-Length (S))+Copy (S, 1, 8);
End; { else }
For I := 2 to 8 do { Ziffern zusammensetzen }
If (Mem [Seg (S):Ofs (S)+I]<$30) or (Mem [Seg (S):Ofs (S)+I]>$39) then
Mem [Seg (S):Ofs (S)+I] := $30;
N := N*(Mem [Seg (S):Ofs (S)+2]-$30)/1*1000000 +
(Mem [Seg (S):Ofs (S)+3]-$30)/1*100000 +
(Mem [Seg (S):Ofs (S)+4]-$30)/1*10000 +
(Mem [Seg (S):Ofs (S)+5]-$30)/1*1000 +
(Mem [Seg (S):Ofs (S)+6]-$30)/1*100 +
(Mem [Seg (S):Ofs (S)+7]-$30)/1*10 +
(Mem [Seg (S):Ofs (S)+8]-$30)/1);
StrToInt := RW (N) { String uebergeben }
End; { StrToInt }

```

Bild 4.4.5.1: Die Prozeduren zur Zahlenumwandlung

Zum Abspeichern und Laden der Strings stehen die Funktionen "DataWrite" und "DataRead" zur Verfügung, welche in Bild 4.4.5.2 zu sehen sind. Zu beachten ist hierbei, daß beim Abspeichern an den String ein "CR/LF" (\$0D+\$0A) angehängt und beim Laden wieder entfernt wird.

Das Laden und Speichern eines gesamten Projektes übernehmen die Prozeduren "LoadFile" und "SaveFile". Hierzu werden zum Sichern zunächst die allgemeinen Daten wie Farben, Gitter an/aus et cetera abgespeichert und dann die Stellen, Transitionen, Kanten und Hintergrundtexte. Dabei werden die Objektlisten von der Wurzel an durchlaufen und Eintrag für Eintrag mit allen Daten nacheinander abgespeichert. Im Falle eines I/O-Fehlers erfolgt ein Sprung nach "Marke", wo die Datei geschlossen und die Funktion wohldefiniert beendet wird. Das Laden erfolgt analog hierzu.

```

Function DataWrite (Var F : DataFile; S : String) : Boolean;
{
  Schreibt einen String in die entsprechende Datei
}
Var
  I : LongInt;

Begin { DataWrite }
  S := S+#SOD+#$0A;           { CR/LF anhaengen }
  For I := 1 to Length (S) do
    Begin { For }
      Write (F,Mem [Seg (S):Ofs (S)+I]); { Character schreiben }
      If IOResult<>0 then
        Begin { then }
          DataWrite := false; { Fehler --> }
          Exit;
        End { then }
      End; { For }
    DataWrite := true;        { Kein Fehler }
  End; { DataWrite }

Function DataRead (Var F : DataFile; Var S : String) : Boolean;
{
  Liest einen String aus der entsprechenden Datei
}
Var
  T : Byte;

Begin { DataRead }
  If Eof (F) then
    Begin { then }
      DataRead := false; { End Of File gefunden }
      Exit;
    End; { then }
  S := '';
  Repeat
    Read (F,T);           { Character lesen }
    If IOResult <> 0 then
      Begin { then }
        DataRead := false; { Fehler --> }
        Exit;
      End; { then }
    S := S+Chr (T);      { Character anhaengen }
  until (T=$0D) or Eof (F); { Bis CR oder EOF lesen }
  Read (F,T);           { LF ueberlesen }
  If IOResult <> 0 then
    Begin { then }
      DataRead := false; { Fehler --> }
      Exit;
    End; { then }
  S := Copy (S,1,Length (S)-1); { CR abschneiden }
  DataRead := true;     { Kein Fehler }
End; { DataRead }

```

Bild 4.4.5.2: Die Prozeduren zum Laden und Speichern von Strings

4.4.6. Druckprozedur

Die Druckprozedur "SavePrintFile" befindet sich im Unit "DISK.PAS", da sie den Bildschirminhalt als PostScript-Datei abspeichert.

Nach dem Öffnen der Datei wird zuerst der Zustand des Ausgabege­rät­es mit "gsave" gesichert, um dann den Zeichensatz, die Strichbreite und das Koordinatensystem für das Petrinetz entsprechend zu setzen (siehe Bild 4.4.6). Danach erfolgen die Angaben für das Zeichnen und Beschriften der Stellen, Transitionen und Kanten. Zuletzt werden die Hintergrundtexte abgespeichert, die Seite durch "showpage" angezeigt und der alte Zustand des Ausgabege­rät­es mit "grestore" wiederherge­stellt. Danach erfolgt das Schließen der Datei.

Der Kopf der PostScript-Datei wird durch den Prozedurteil in Bild 4.4.6 erzeugt.

```
{ *** Datei oeffnen *** }
Assign (PNEFile, PNEDir+PNEName+'.PS');
Rewrite (PNEFile);
If IOResult<>0 then Goto Marke;

{ *** Kommentar *** }
If not DataWrite (PNEFile, '%PNE POST SCRIPT FILE') then Goto Marke;
If not DataWrite (PNEFile, '%Projectname: '+ProjectName) then Goto Marke;
If not DataWrite (PNEFile, '%Projectdate: '+ProjectDate) then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;

{ *** alten Status sichern *** }
If not DataWrite (PNEFile, 'gsave') then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;

{ *** Multiplikatoren setzen *** }
If not DataWrite (PNEFile, '/UniTX ( 41.8 div) def') then Goto Marke;
If not DataWrite (PNEFile, '/UniTY (-57.4 div) def') then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;

{ *** Schrifntfont laden *** }
If not DataWrite (PNEFile, '/Helvetica findfont 15 scalefont setfont') then Goto Marke;
If not DataWrite (PNEFile, " ") then Goto Marke;

{ *** Koordinatensystem setzen *** }
If not DataWrite (PNEFile, '-90 rotate') then Goto Marke;
If not DataWrite (PNEFile, '-808 572 translate') then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;

{ *** Liniendicke und Muster setzen *** }
If not DataWrite (PNEFile, '1 setlinewidth') then Goto Marke;
If not DataWrite (PNEFile, '[] 0 setdash') then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;
If not DataWrite (PNEFile, ' ') then Goto Marke;
```

Bild 4.4.6: Die Erzeugung des Kopfes der PostScript-Datei

4.5. Beispiel eines Programmablaufs

Um den Programmablauf innerhalb des Editors zu verdeutlichen, sei hier das Zeichnen einer Stelle auf dem Hintergrund als ein typisches Beispiel ausgewählt.

Das Bewegen des Maussymboles auf dem Bildschirm wird während des gesamten Programmes vom Maustreiber übernommen. Hierbei wird das Maussymbol lediglich zum Zeichnen von Objekten und Schreiben von Texten kurzzeitig ausgeschaltet, da dies aufgrund der eingesetzten Grafikkarte (IBM-EGA) erforderlich ist.

Die Programmsteuerung an sich erfolgt über den Parser. Der Parser ruft den Scanner auf, welcher sich im Zustand "StatusNormal" zunächst in einer Eingabeschleife befindet. In dieser ruft der Scanner die Mausstatusfunktion des Maustreibers auf, um zu erkennen, wann die linke Maustaste gedrückt wird. Sobald dies geschehen ist, legt der Scanner die Mauskoordinaten in den Variablen ("MouseX";"MouseY") ab und prüft, ob die Koordinaten innerhalb der oberen Menüleiste, im normalen Hintergrund oder innerhalb einer Stelle, Transition oder auf einer Kante liegen.

In unserem Beispiel sei dies der Hintergrund. Dementsprechend gibt der Scanner das Token "_BACKGROUND" aus.

Dieses Token wird nun vom Parser weiterverarbeitet. Der Parser erkennt an Hand des Tokens, daß die Maustaste auf dem Hintergrund gedrückt wurde und ruft eine Prozedur auf, um das Hintergrundmenü zu eröffnen. Die Verwaltung des Menüs, also das Bewegen und Invertieren des Menübalkens, erfolgt dann wieder über den Scanner. Dieser wird vorher in den Zustand "StatusMenue" gesetzt, damit er erkennt, daß er eine Menüeingabe zu verwalten hat. Sobald der gewünschte Menüpunkte ausgewählt und die Maustaste auf diesem losgelassen wird, setzt der Scanner die Variable "MenueR. Cho ice" auf die gewählte Eintragsnummer des Menüs, in diesem Fall "0". Dann wird das Token "_ADDPL" zum Setzen einer Stelle ausgegeben.

"_ADDPL" veranlaßt den Parser nun die Prozedur "AddPI Proc" aufzurufen. "AddPI Proc" ruft ihrerseits "NewPI ace" auf. Diese Prozedur erzeugt ein neues Stellen-Record auf dem Heap und reiht es in die einfach verkettete Liste der Stellen ein.

Danach wird mit "InputString" der Name der Stelle abgefragt und zusammen mit den Koordinaten, der Größe, der laufenden Nummer, einem Pointer auf die nächste Stelle und einen Pointer auf einen eventuellen Textpuffer abgelegt. Schließlich wird die Stelle unter Aufruf von "DrawPlace" gezeichnet und durch "NamePlace" mit ihrem Namen am Bildschirm versehen.

Hiermit ist die Stelleneingabe abgeschlossen, und es erfolgt erneut der Aufruf des Scanners, um weitere Eingaben zu analysieren und auszuwerten.

Ähnlich verläuft auch die Eingabe einer Transition oder einer Kante. Der Parser ruft jedesmal den Scanner auf, welcher eine Benutzereingabe abwartet und ein entsprechendes Token ausgibt. An Hand des Tokens entscheidet der Parser über den weiteren Programmverlauf.

5. Anwendung des Editors

5.1. Programmeinführung

Die Bedienung des Editors erfolgt über Tastatur und Maus, wobei die Tastatur zur Eingabe von Text und für alle weiteren Aktionen die Maus vorgesehen ist.

Die Maus kann jedoch auch mit der Tastatur emuliert werden. Die dafür vorgesehenen Tasten und ihre Bedeutung sind in Bild 5.1.1 ersichtlich. Dabei wird der Mauszeiger durch Drücken der entsprechenden Taste bewegt, beziehungsweise das Drücken der linken Maustaste emuliert.

*** Maus langsam bewegen ***

Pfeil-Links	= CTL-S : einen Punkt nach links
Pfeil-Rechts	= CTL-D : einen Punkt nach rechts
Pfeil-Hoch	= CTL-E : einen Punkt nach oben
Pfeil-Runter	= CTL-X : einen Punkt nach unten

*** Maus schnell bewegen ***

CTL-Pfeil-Links	= CTL-A : fünfzig Punkte nach links
CTL-Pfeil-Rechts	= CTL-F : fünfzig Punkte nach rechts
PAGE-UP	= CTL-R : fünfzig Punkte nach oben
PAGE-DOWN	= CTL-C : fünfzig Punkte nach unten

*** Maustaste betätigen ***

SCROLL-LOCK : linke Maustaste drücken/lassen

Bild 5.1.1: Mausemulation auf der Tastatur

Da die Verwaltung der Maus von einem Maustreiber übernommen wird, muß der Treiber vor dem Start des Programmes geladen sein.

Der Editor wird aus der DOS-Ebene durch den Aufruf "PNE" gestartet. Unmittelbar nach dem Start wird die Konfigurationsdatei geladen, welche Informationen über den Programmstatus der letzten Sitzung, wie Farben, Gitter an/aus, Gittergröße, Größe der Stellen und Transitionen sowie weitere Einzelheiten enthält. Danach werden einige Variablen initialisiert. Hierzu zählen Menüeinträge, verkettete Listen für die zu verwaltenden Objekte et cetera.

Nach der Initialisierung und dem Aufbau des Bildschirms kann der Benutzer Eingaben tätigen. Hierzu ist der Bildschirm in zwei Bereiche unterteilt. Der obere Bereich wird begrenzt durch eine umrandete Menüleiste. Drückt man innerhalb dieser Leiste die Maustaste, so erscheint das Generalmenü, in welchem allgemeine Programmparameter, wie Gitter an/aus, Kanten abgerundet/eckig und so weiter, angewählt werden können. Der zweite Bereich ist der Hintergrundbereich unterhalb der Menüleiste. Hier können Objekte wie Stellen, Transitionen, Kanten und Hintergrundtexte bearbeitet werden (siehe Bild 5.1.2).

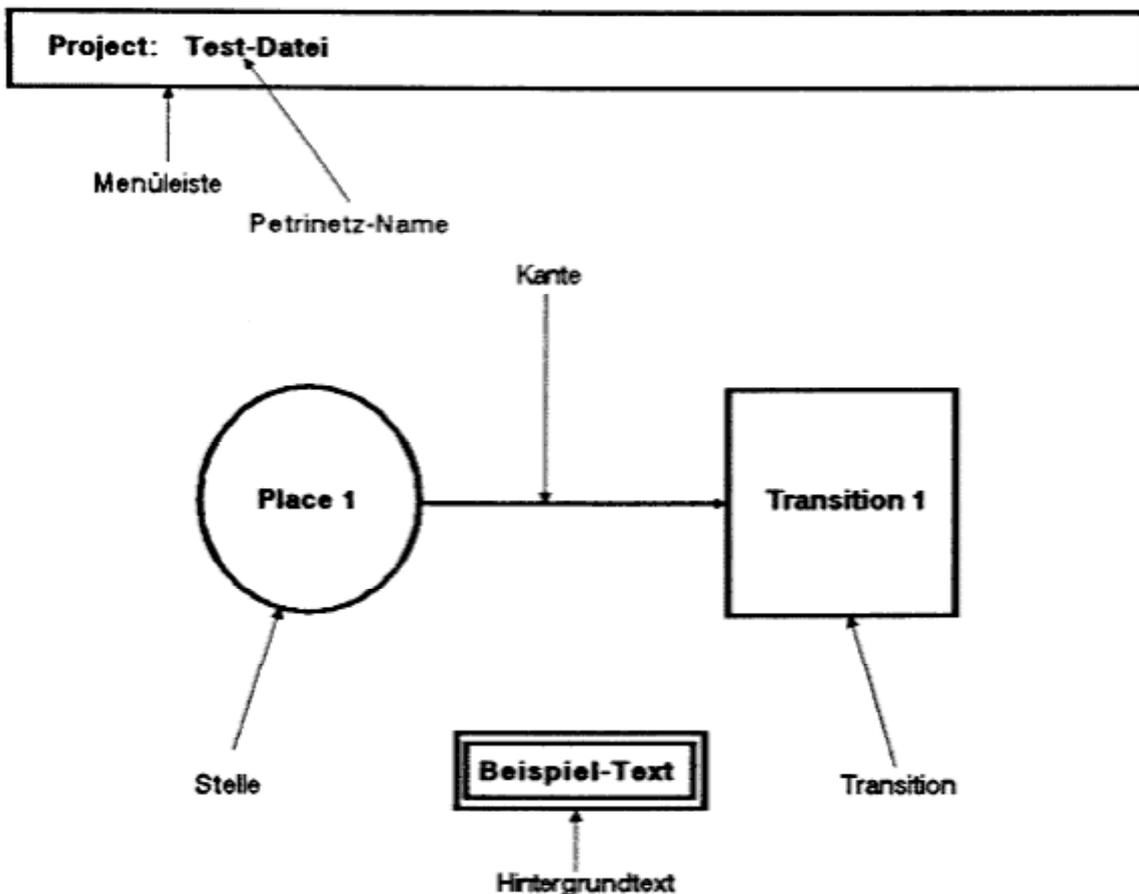


Bild 5.1.2: Aufteilung des Bildschirms

Stellen, Transitionen und Hintergrundtexte können unmittelbar auf dem leeren Hintergrund eingegeben werden. Zur Eingabe von Kanten ist es erforderlich, daß vorher eine Stelle oder Transition eingegeben wurde, da Kanten immer von diesen und nicht vom Hintergrund ausgehen.

Soll eine Stelle oder Transition eingegeben werden, so plziert man den Mauszeiger auf der gewünschten Stelle innerhalb des Hintergrundes und drückt die linke Maustaste. Hierauf erscheint das Hintergrundmenü, und man kann sich die gewünschte Objektart aus dem Menü aussuchen.

Danach wird man zur Eingabe eines Namens aufgefordert. Wird kein Name eingegeben, so werden Stellen mit "Place n" benannt, Transitionen mit "Transition m", wobei n und m fortlaufende, bei 1 startende, natürliche Zahlen sind. Es erfolgt ebenfalls eine Prüfung, ob der eingegebene Name bereits vorher einmal eingegeben wurde.

In dem Fall, daß der Name bereits vorher schon an ein gleiches Objekt vergeben wurde, erscheint eine Fehlermeldung. Nun muß der Name neu eingegeben werden, da die Namen innerhalb der Stellen beziehungsweise Transitionen bei Petrinetzen eineindeutig sein müssen. Das Objekt wird danach an der angewählten Position gezeichnet und mit dem entsprechenden Namen versehen.

Soll eine Kante eingefügt werden, so plziert man den Mauszeiger auf eine bereits existierende Stelle oder Transition, je nachdem, ob die Kante von einer Stelle oder von einer Transition ausgehen soll. Nach dem Drücken der Maustaste wählt man den Punkt "Source" aus dem Menü aus. Jetzt kann man entweder direkt von einer Stelle zu einer Transition beziehungsweise von einer Transition zu einer Stelle die Kante zeichnen, oder man kann während des Zeichnens Verbindungspunkte setzen. Möchte man direkt von einer Stelle zu einer Transition eine Kante eingeben, so beginnt man bei der Stelle, wählt "Source", bewegt den Mauszeiger auf die gewünschte Transition und wählt "Destination". Entsprechendes gilt für das Zeichnen von Transitionen zu Stellen. Sollen Verbindungspunkte eingegeben werden, so drückt man während des Zeichnens über dem Hintergrund die Maustaste und wählt "Add connect point" aus.

Während des gesamten Zeichenvorganges wird die Kante laufend von ihrem Beginn beziehungsweise vom letzten Verbindungspunkt aus bis zur aktuellen Position der Maus gezeichnet.

Hintergrundtexte lassen sich überall auf dem Hintergrund plazieren. Hierzu wählt man im Hintergrundmenü den Punkt "Add Text" aus. Nach Eingabe des Textes wird dieser an der entsprechenden Stelle plaziert. Durch Eingabe einer geeigneten Rahmenfarbe (Schwarz) kann der Rahmen der Hintergrundtexte auch unterdrückt werden (Auswahlpunkt "Defaults" im Generalmenü).

Es lassen sich alle Eingaben als Projekt abspeichern und später wieder laden. Dafür wählt man im Generalmenü den Punkt "Load" beziehungsweise "Save" und gibt den Dateinamen an. Das Verzeichnis, in welchem die Lade- und Speicheraktionen ausgeführt werden, kann im Punkt "Defaults" im Generalmenü festgelegt werden. Sollte bei dem Abspeichern eines Projektes bereits eine Datei gleichen Namens im jeweiligen Verzeichnis vorhanden sein, so erfolgt eine Sicherheitsabfrage, ob die alte Datei gelöscht werden soll. In diesem Fall wird sie durch das aktuelle Projekt ersetzt. Im anderen Fall kann ein neuer Name angegeben werden. Die Datei wird beim Abspeichern zusätzlich mit der Extension ".PNE" versehen.

Sollten vor dem Laden eines neuen Projektes die Änderungen noch nicht abgespeichert worden sein, welche nach dem letzten Laden oder Speichern erfolgten, erscheint ebenfalls eine Sicherheitsabfrage, ob die Änderungen zuvor abgespeichert werden sollen. In diesem Fall kann wieder ein Projektname eingegeben werden, und die alten Eingaben werden unter diesem gesichert.

Ebenso kann ein eingegebenes oder geladenes Projekt als PostScript-Datei abgespeichert werden, um es später auf einem PostScript-fähigen Drucker ausdrucken zu lassen. Hierzu ist der Punkt "Print" im Generalmenü vorgesehen. Erneut erfolgt eine Angabe des gewünschten Dateinamens unter dem die PostScript-Datei gesichert werden soll. Diese Datei wird mit der Extension ".PS" versehen.

Eine ganze Reihe von Hilfsmitteln, wie ein Gitter mit frei definierbarer Größe, abgerundete Ecken bei Kanten, verschiedene Grundgrößen von Stellen und Transitionen, rechtwinklige Kanteneingabe, stehen dem Benutzer im Generalmenü zur Verfügung.

Diese und alle anderen Menüs werden in Kapitel 5.3 Punkt für Punkt erläutert.

5.2. Bedienung den Texteditors

Den Stellen, Transitionen und dem Petrinetzwerk selbst können Texte zugeordnet werden, welche ihre Eigenschaften näher spezifizieren sollen.

Bei Stellen ist dies ihre Kapazität und die Art der zu speichernden Marken. Bei den Transitionen gibt es Texte für Schaltbedingungen, Verzögerungen und für die Schaltprozedur. Das Petrinetzwerk kann mit einem Text eine Anfangsbelegung zugewiesen bekommen.

Alle diese Texte beziehen sich dabei auf die NSL-Sprache (siehe Kapitel 2.2). Die Texte können mit einem Texteditor eingegeben und bearbeitet werden. Danach erfolgt eine Syntaxprüfung bezüglich der NSL-Grammatik (siehe Kapitel 5.3.2.4)

Um die Bearbeitung dieser Texte komfortabel zu gestalten, unterstützt der Texteditor eine Reihe von speziellen Tasten, im folgenden als Editiertasten bezeichnet. Sie dienen dazu, den Cursor auf dem Eingabebereich zu verschieben und zu positionieren sowie Buchstaben und Ziffern einfügen und löschen zu können.

Grundsätzlich kann die Texteingabe mit der ESC-Taste abgeschlossen werden. Diese Taste und alle weiteren Editiertasten sind mit ihrer Funktion in Bild 5.2 aufgeführt.

ESC	:	Textbearbeitung beenden
RETURN	:	naechste Zeile
INSERT	:	ein fuegen/Ueberschreiben
BACKSPACE	:	voriges Zeichen loeschen
DELETE	:	Zeichen loeschen
CTL-Y	:	Zeile loeschen
Pfeil-Links	= CTL-S	: ein Zeichen nach links
Pfeil-Rechts	= CTL-D	: ein Zeichen nach rechts
Pfeil-Hoch	= CTL-E	: eine Zeile nach oben
Pfeil-Runter	= CTL-X	: eine Zeile nach unten
BEGIN	:	zum Zeilenanfang
END	:	zum Zeilenende
CTL-BEGIN	:	zum Beginn der Seite
CTL-END	:	zur Ende der Seite
PAGE-UP	:	vorige Seite
PAGE-DOWN	:	naechste Seite

Bild 5.2: Unterstützte Editiertasten

5.3. Erläuterung der einzelnen Menüfunktionen

Um die eingegebenen Objekte weiter bearbeiten zu können, sind eine Reihe von Menüs vorgesehen. Hierbei kann das Generalmenü durch Drücken der Maustaste in der oberen Menüleiste angewählt werden. Die anderen Menüs können eröffnet werden, in dem man den Mauszeiger auf der jeweiligen Objektart plziert und die Maustaste betätigt. Das Hintergrundmenü kann demzufolge auf dem leeren Hintergrund und das Stellenmenü auf einer Stelle angewählt werden.

Die Funktionen dieser Menüs sind nun im einzelnen beschrieben, wobei die programmtechnischen Einzelheiten in Kapitel 4 erläutert sind.

5.3.1. Funktionen des Generalmenüs

Nach Anwahl dieses Menüs durch Betätigen der Maustaste innerhalb der oberen Menüleiste erscheint das in Bild 5.3.1 wiedergegebene Menü. Es dient dem Einstellen von Programmparametern sowie allgemeinen Funktionen wie Laden/Abspeichern und Drucken.

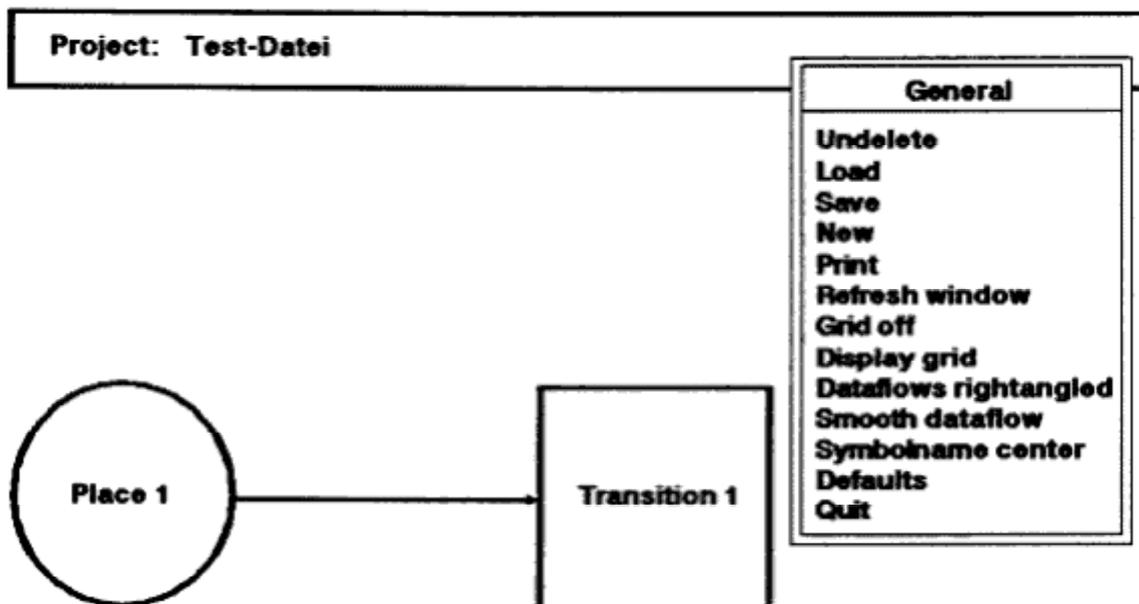


Bild 5.3.1: Das Generalmenü

5.3.1.1. "Undelete"

Die "Undelete"-Funktion macht eine zuvor erfolgte Löschoperation eines Objektes rückgängig. Erst wenn ein Objekt gelöscht wird, ist dieser Menüpunkt zulässig und damit anwählbar. Wird er daraufhin angewählt, erscheint das zuvor gelöschte Objekt mit allen Eigenschaften wie Größe, Text und so weiter an seiner alten Position. Werden zwei Objekte hintereinander gelöscht, so kann immer nur das letzte wiederhergestellt werden. Gültige Objekte sind hierbei Stellen, Transitionen, Kanten, Verbindungspunkte und Hintergrundtexte.

5.3.1.2. "Load"

Die "Load"-Funktion dient dem Laden eines bereits vorher eingegebenen Petrinetzes mit allen seinen Angaben und aktuellen Programmparametern, wie etwa Gitter an/aus, Gittergröße, Farben et cetera.

Nach der Anwahl dieses Menüpunktes erscheint ein Eingabefenster für den Dateinamen. Die Eingabe kann durch die Editiertasten von Bild 5.2 verbessert werden. Wurde vorher bereits eine Datei gesichert oder geladen, so wird der alte Dateiname als Vorgabe angezeigt. Wird als erstes ein Buchstabe oder eine Ziffer eingegeben, wird die gesamte Vorgabe gelöscht, und es kann ein neuer Name eingegeben werden. Im Falle, daß zuerst eine Editiertaste gedrückt wird, kann der alte Name weiter bearbeitet werden.

Soll der Ladevorgang abgebrochen werden, kann hierzu die ESC-Taste benutzt werden. Wird die Namenseingabe mit Return abgeschlossen, so wird das gesamte Projekt unter dem angegebenen Dateinamen mit der Extension ".PNE" in dem unter dem Punkt "Default" angegebenen Verzeichnis gesucht.

Ist eine Datei dieses Namens vorhanden, wird sie geladen. Dabei wird kontrolliert, ob die Datei derjenigen Syntax entspricht, welche der Editor beim Sichern einer Datei benutzt. Wird keine Datei dieses Namens oder eine Datei mit unterschiedlicher Syntax gefunden, wird der Ladevorgang mit einer entsprechenden Fehlermeldung abgeschlossen. Ansonsten werden alle abgespeicherten Parameter, wie Farben und so weiter, neu eingestellt und das Petrinetz gezeichnet.

5.3.1.3. "Save"

Mit der "Save"-Funktion kann ein gerade erstelltes Petrinetz gesichert werden. Hierbei werden alle seine Angaben und aktuellen Programmparametern abgespeichert. Dazu gehören Gitter an/aus, Gittergröße, Farben, Default-Größe von Stellen und Transitionen, Projektnamen und -datum, wie sie zuvor im Generalmenü eingestellt wurden.

Soll der Sicherungsvorgang während der Eingabe des Dateinamens abgebrochen werden, kann hierzu wieder die ESC-Taste gedrückt werden. Wird die Namenseingabe mit Return abgeschlossen, wird das gesamte Projekt unter dem angegebenen Dateinamen mit der Extension ". PNE" gesichert. Ist bereits eine Datei mit gleichem Namen vorhanden, erscheint ein weiteres Fenster, und der Benutzer hat die Möglichkeit, den Vorgang abubrechen oder die alte Datei zu überschreiben.

Anstelle des Anwählens der Auswahlpunkte "YES" oder "NO" können auch die "Y"- oder die Return-Taste zur Bestätigung, beziehungsweise die "N"- oder die ESC-Taste zur Verneinung eingegeben werden.

5.3.1.4. "Hex"

Um ein neues Projekt anzufangen und alle bereits eingegebenen Objekte zu löschen, muß dieser Menüpunkt angewählt werden.

Sind bereits Objekte eingegeben und seit der letzten Änderung nicht gesichert worden, erfolgt eine Abfrage, ob das derzeitige Projekt zuvor gesichert werden soll. Falls eine positive Antwort erfolgt, wird das gesamte Projekt unter einem einzugebenden Namen gesichert, wie im Punkt 5.3.1.2 beschrieben, und danach alle Objektdaten gelöscht. Im anderen Fall werden sofort alle Daten gelöscht.

5.3.1.5. "Print"

Diese Funktion dient dem Sichern einer PostScript-Datei, welche später an ein PostScript-fähiges Ausgabegerät weitergegeben werden kann.

Es erfolgt die Eingabe des Dateinamens, an welchen die Endung ". PS" hinzugefügt wird. Der gesamte Vorgang kann während der Eingabe mit der ESC-Taste abgebrochen werden. Ansonsten wird der Bildschirminhalt in eine PostScript-Datei übertragen.

5.3.1.6. "Refresh window"

Der Aufruf dieser Funktion baut den gesamten Bildschirminhalt mit allen Objekten und Texten neu auf. Dies ist für den Fall vorgesehen, daß sich einzelne Punkte auf dem Bildschirm befinden, welche nicht zum Petrinetz gehören und gelöscht werden sollen (zum Beispiel durch Interruptprogramme wie Uhren et cetera).

5.3.1.7. "Grid on/off"

Damit sich die Objekte auf dem Bildschirm leicht ausrichten lassen, kann man ein Gitter einschalten. Dadurch finden alle Plazierungen von Objekten immer auf dem nächsten Gitterpunkt statt, während der Mauszeiger nach wie vor alle Punkte anfahren kann. Beim Drücken der Maustaste werden die Koordinaten auf den nächsten Gitterpunkt abgeglichen.

Die entsprechende Gittergröße wird im Punkt "Defaults" unter "Grid size" eingetragen (siehe 5.3.1.12).

5.3.1.8. "Display/Hide grid"

Entsprechend dem vorigen Menüpunkt läßt sich hier entscheiden, ob die Gitterpunkte angezeigt werden sollen oder nicht.

Die Gitterpunkte werden nur bei zuvor eingeschaltetem Gitter angezeigt (siehe 5.3.1.7).

5.3.1.9. "Dataflows right/normal angled"

In der Grundeinstellung folgt beim Zeichnen von Kanten das Ende der Kante immer dem Mauszeiger. Oftmals ist es jedoch aus optischen Gründen sinnvoll, nur rechtwinklige Kanten zuzulassen. Nach dem Anwählen dieser Funktion werden alle Kanten rechtwinklig gezeichnet, indem die Kanten entweder vertikal oder horizontal gezeichnet werden, jeweils entlang der längsten Strecke.

5.3.1.10. "Smooth/Unsmooth data flow"

Um Verbindungspunkte werden normalerweise Kreise gezeichnet, welche den Bereich markieren, in dem die Punkte mit dem Mauszeiger angewählt werden können.

Die Kreise können mit dieser Funktion in abgerundete Ecken umgewandelt werden und umgekehrt.

Sowohl der Kreisradius als auch die Größe der abgerundeten Ecken können unter "Defaults" eingestellt werden (siehe 5.3.1.12).

5.3.1.11. "Symbolname bottom/center"

Diese Funktion bestimmt, ob die Symbolnamen in der Mitte der Symbole oder unterhalb der Symbole angebracht werden. Dies betrifft Stellen und Transitionen gleichermaßen.

5.3.1.12. "Defaults"

Nach Anwahl dieses Menüpunktes erscheint ein großes Eingabefenster, in welchem die folgenden Vorgaben für das weitere Zeichnen gemacht werden können:

- sämtliche Farben sowohl der Objekte als auch der Menüs,
- Standard-Radius der Stellen,
- Standard-Länge und -Breite der Transitionen,
- Größe der abzurundenden Ecken bei der "Smooth"-Funktion,
- Radius der Verbindungspunkte,
- Länge der Pfeilspitzen der Kanten,
- Winkel der Pfeilspitzen der Kanten (in Grad),
- Gittergröße,
- Verzeichnisname zum Laden und Speichern aller Dateien,
- eigener Projektname und Projektdatum.

Die Eingabefelder können mit Pfeil-Hoch, Pfeil-Runter, beziehungsweise der Tab- und Return-Taste gewechselt werden. Übernommen werden die Änderungen durch Drücken der ESC-Taste.

5.3.1.13. "Quit"

Als letzter Menüpunkt kann an dieser Stelle das Programm nach Bestätigung einer Sicherheitsabfrage verlassen werden. Sind seit dem letztem Laden oder Speichern weitere Änderungen an dem Petrinetz vorgenommen worden, erfolgt zuvor eine weitere Abfrage, ob die Änderungen gesichert werden sollen.

5.3.2. Funktionen den Hintergrundmenüs

Dieses in Bild 5.3.2 wiedergegebene Menü kann auf dem leeren Hintergrund aktiviert werden. Es dient dem Hinzufügen von Stellen, Transitionen und Kanten sowie der Eingabe eines Setup-Textes.

Die eingegebenen Objekte können dann später über eigene Menüs weiter bearbeitet werden.

5.3.2.1. "Add place"

An der aktuellen Mausposition wird eine neue Stelle hinzugefügt. Der Radius der Stelle entspricht der Standardgröße, wie sie im Generalmenü unter "Defaults" oder im Stellenmenü unter "Take as default" eingestellt werden kann. Weiterhin erscheint ein Fenster zur Eingabe des Stellennamens. Wird hier ein Leerstring eingegeben, so erhält die Stelle den Namen "Place n". Hierbei ist n eine natürliche Zahl, welche fortlaufend numeriert wird. Falls ein eigener Name eingegeben wird, so darf dieser Name nicht schon an eine andere Stelle vergeben worden sein, sonst erscheint eine Fehlermeldung mit anschließender Neueingabe.

5.3.2.2. "Add transition"

Eine neue Transition wird an der aktuellen Position der Maus hinzugefügt. Hierbei können Länge und Breite im Generalmenü unter "Defaults" oder im Stellenmenü unter "Take as default" eingestellt werden. Auch hier erscheint ein Fenster zur Eingabe des Transitionsnamens.

Bei Eingabe eines Leerstrings erhält die Transition den Namen "Transition m", wobei m eine natürliche Zahl ist, welche fortlaufend durchnummeriert wird. Falls ein eigener Name eingegeben wird, darf dieser Name nicht schon an eine andere Transition vergeben worden sein. Andernfalls erscheint, analog zu den Stellen, eine Fehlermeldung, und eine neue Namenseingabe wird erforderlich.

5.3.2.3. "Add text"

Mit dieser Funktion läßt sich auf dem Hintergrund an beliebiger Position Text hinzufügen. Damit können zum Beispiel Kanten benannt oder Projekterläuterungen angebracht werden.

Um den Text wird ein Rahmen in Form eines Rechteckes gezeichnet, und dieses mit seiner linken oberen Ecke an der aktuellen Hausposition in den Hintergrund eingefügt. Die Eingabe erfolgt über den in Kapitel 5.2 erläuterten Texteditor.

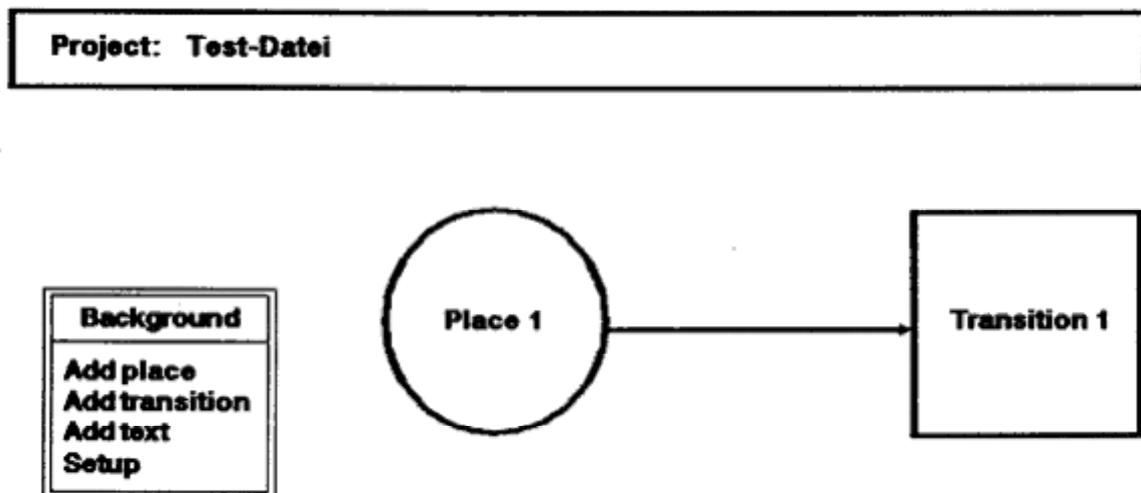


Bild 5.3.2: Das Hintergrundmenü

5.3.2.4. "Setup"

Die Initialisierung des Petrinetzes geschieht mit dieser Funktion, indem die Anfangsbelegung des Netzes mittels eines Textes beschrieben wird. Die Eingabe des Textes läuft über den Texteditor von Kapitel 5.2..

Danach erscheint ein Eingabefenster, in welchem der Benutzer bestimmt, ob der gerade eingegebene Text verworfen oder abgespeichert wird.

Falls er abgespeichert werden soll, erfolgt vorher eine Prüfung auf Übereinstimmung mit den Regeln der NSL-Sprache. Wird hierbei eine Diskrepanz gefunden, erscheint eine Fehlermeldung. Der Cursor befindet danach hinter dem Wort, welches den Fehler hervorgerufen hat. Der Text kann nun verändert und erneut geprüft werden.

Soll der Text verworfen werden, erfolgt keine weitere Syntaxprüfung.

5.3.3. Funktionen des Stellenmenüs

Dieses Menü erscheint durch Drücken der Maustaste, wenn sich der Mauszeiger auf einer Stelle befindet. Es dient dem Bearbeiten von Eigenschaften der betreffenden Stelle.

In Bild 5.3.3 sind die möglichen Funktionen aufgeführt.

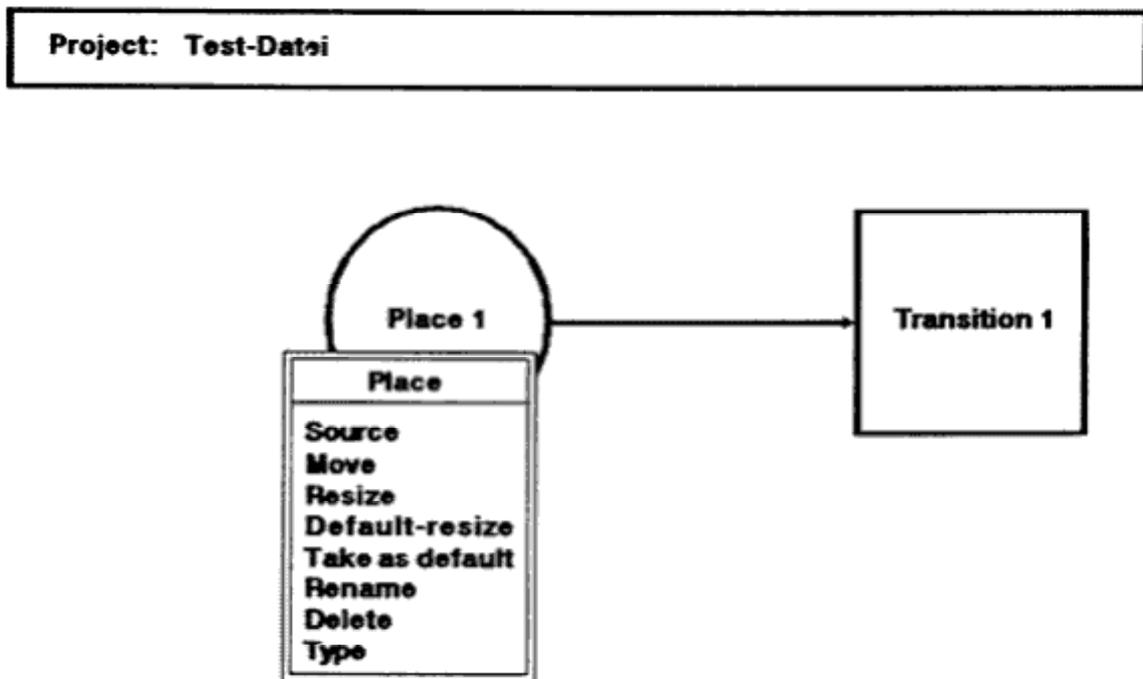


Bild 5.3.3: Funktionen des Stellenmenüs

5.3.3.1. "Source"

Mit dieser Funktion wird das Zeichnen einer Kante initiiert. Die angewählte Stelle ist dabei der Beginn der Kante. Ab nun folgt das Ende der Kante dem Mauszeiger. Während des Zeichnens können Verbindungspunkte auf dem Hintergrund eingefügt werden. Das Zeichnen der Kanten wird schließlich auf einer Transition durch den Menüpunkt "Destination" abgeschlossen.

5.3.3.2. "Nova"

Nach der Anwahl dieses Punktes kann die angewählte Stelle mit der Maus verschoben werden. Alle damit verbundenen Kanten werden dabei automatisch mitbewegt (Gummibandeffekt). Ist die gewünschte neue Position erreicht, wird die Stelle nach dem Drücken der Maustaste dort plziert.

5.3.3.3. "Resize"

Um die Größe eines Stellensymboles zu verändern, wird dieser Menüpunkt benutzt. Hierbei wird der Radius der Stelle je nach Mausposition verändert und durch Drücken der Maustaste übernommen. Dabei wird der Radius immer so gewählt, daß sich der Mauscursor auf dem Kreis selbst befindet.

5.3.3.4. "Default-resize"

Diese Funktion setzt die Größe des angewählten Stellensymboles auf den Standardwert zurück. Dieser kann durch "Take as default" oder im Generalmenü unter "Defaults" gesetzt werden.

5.3.3.5. "Take as default"

Mit dieser Funktion wird die Größe der angewählten Stelle als Standardgröße übernommen. Alle neuen Stellen erhalten als Vorgabe diese Größe.

5.3.3.6. "Rename"

Um eine angewählte Stelle umzubenennen, wird die "Rename"-Funktion eingesetzt. Die Eingabe des neuen Namens kann mit der ESC-Taste abgebrochen und mit der Return-Taste abgeschlossen werden. Auch hierbei wird der alte Name gelöscht, wenn als erstes Zeichen ein Buchstabe oder eine Ziffer eingegeben wird. Erfolgt hingegen als erstes Zeichen die Betätigung einer Editiertaste, kann der alte Name weiter bearbeitet werden.

5.3.3.7. "Delete"

Soll eine Stelle gelöscht werden, kann dies durch die "Delete"-Funktion erfolgen. Die Stelle verschwindet dann mit allen ihren Kanten vom Bildschirm; sie kann aber später durch die "Undelete"-Funktion im Generalmenü wieder zurückgeholt werden. Wird jedoch zwischenzeitlich ein anderes Objekt gelöscht, ist die alte Stelle unweigerlich gelöscht.

5.3.3.8. "Type"

Um einen Stellentyp festzulegen, kann hier der Stelle ein Text zu seiner Beschreibung zugeordnet werden.

Die Eingabe des Textes geschieht wieder mit Hilfe des in Kapitel 5.2 beschriebenen Texteditors. Danach kann der Text verworfen oder abgespeichert werden. Vor dem Abspeichern erfolgt eine Prüfung, ob der Text der Type-Syntax von Stellen entspricht. Im Falle eines Fehlers erscheint eine Fehlermeldung und der Cursor wird hinter das falsche Wort gesetzt, um den Text erneut zu verbessern und abzuspeichern.

Sollte der Text verworfen werden, erfolgt keine Syntaxprüfung.

5.3.4. Funktionen des Transitionenmenüs

Dieses Menü kann durch das Drücken der Maustaste innerhalb einer Transition eröffnet werden. Hierin sind die Funktionen enthalten, welche dem Bearbeiten von Transitionseigenschaften dienen.

Bis auf die letzten drei Textfunktionen entspricht es dem Stellenmenü.

In Bild 5.3.4 sind die möglichen Funktionen des Transitionenmenüs aufgeführt.

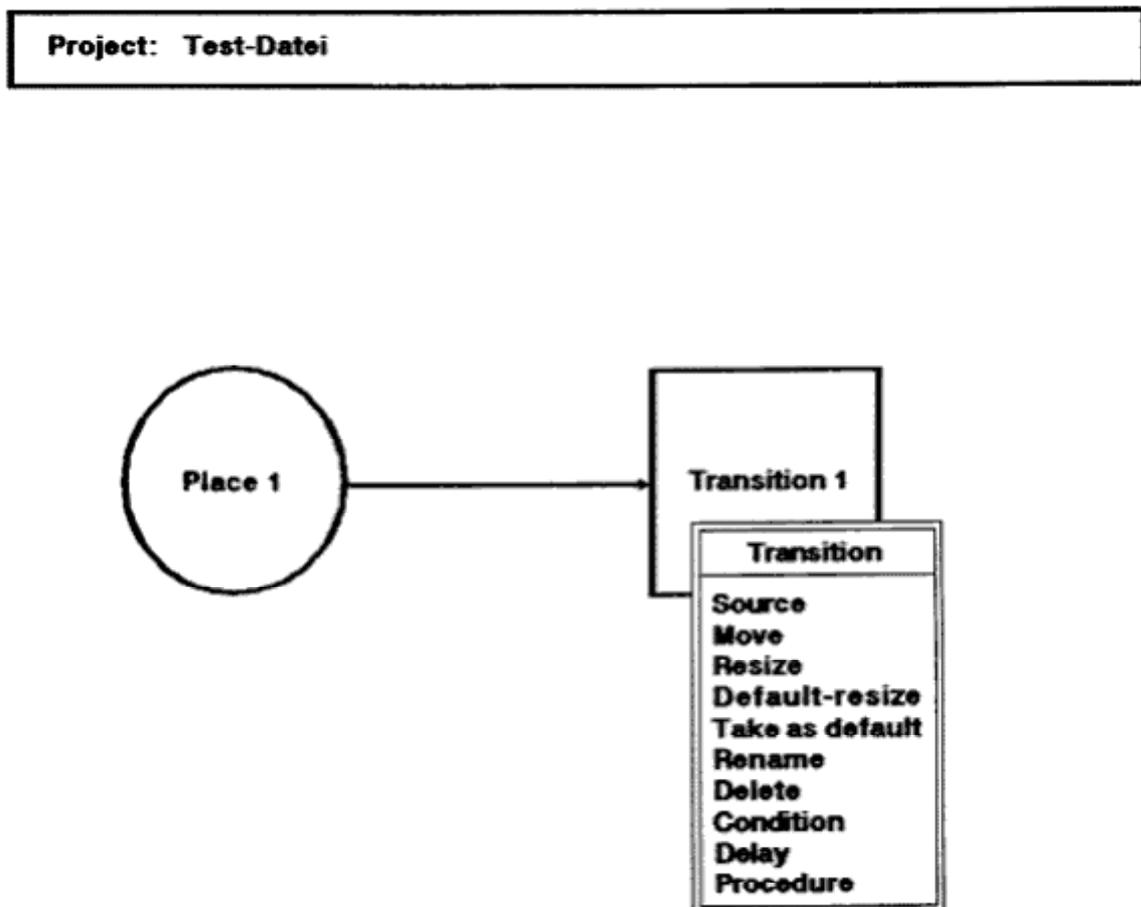


Bild 5.3.4: Das Transitionenmenü

5.3.4.1. "Source"

Das Zeichnen einer Kante wird hiermit gestartet, analog der "Source"-Funktion von Stellen. Die angewählte Transition ist der Beginn der Kante. Das Zeichnen der Kante wird entsprechend auf einer Stelle durch den Punkt "Destination" abgeschlossen.

5.3.4.2. "Move"

Nach der Anwahl dieses Punktes kann die Transition mit der Maus verschoben und durch Drücken der Maustaste dort endgültig plziert werden.

5.3.4.3. "Resize"

Hiermit werden Größe und Länge der Transition je nach Mausposition verändert. Der Mauszeiger befindet sich hierbei immer auf einer Ecke des Rechteckes der Transition. Die aktuellen Werte werden durch Drücken der Maustaste übernommen.

5.3.4.4. "Default-resize"

Die Größe der angewählten Transition wird auf den Standardwert zurückgesetzt. Dieser kann durch "Take as default" oder im Generalmenü unter "Defaults" eingestellt werden.

5.3.4.5. "Take an default"

Die aktuelle Größe der Transition wird als Standardgröße für alle weiteren Transitionen übernommen.

5.3.4.6. "Rename"

Hiermit kann die angewählte Transition umbenannt werden. Die Namenseingabe kann entsprechend der gleichen Funktion im Stellenmenü bearbeitet oder abgebrochen werden.

5.3.4.7. "Delete"

Diese Funktion löscht die aktuelle Transition. Genau wie bei gelöschten Stellen kann die Transition durch "Undel ete" im Generalmenü zurückgeholt werden, solange kein anderes Objekt gelöscht wurde.

5.3.4.8. "Condition"

Einer Transition kann durch diese Funktion ein Text zugeordnet werden, welcher die Bedingungen zum Schalten der Transition beschreibt.

Der Text wird mittels des in Kapitel 5.2 beschriebenen Editors bearbeitet. Nach der Abfrage, ob der Text verworfen oder abgespeichert werden soll, wird dieser beim Abspeichern wieder auf seine syntaktische Richtigkeit bezüglich der NSL-Sprache überprüft. Wird hierbei ein Fehler gefunden, erscheint eine Fehlermeldung. Danach wird der Cursor hinter das Wort gesetzt, das die Fehlermeldung hervorrief. Der Text kann nun verändert und erneut geprüft werden.

5.3.4.9. "Delay"

Hier kann zu der angewählten Transition ein Text eingegeben werden, welcher die Verzögerungszeit festlegt. Nach der Texteingabe erfolgt wieder eine Syntaxprüfung mit eventueller Fehlerkorrektur.

5.3.4.10. "Procedure"

Die "Procedure"-Funktion ordnet der jeweils angewählten Transition einen Text zu, der das Schaltverhalten definiert. Auch hier wird eine Syntaxprüfung des Textes mit anschließender Cursorpositionierung im Fehlerfall vorgenommen.

5.3.5. Funktionen des Kantenmenüs

Wird eine Kante oder eine ihrer Verbindungspunkte mit dem Mauszeiger angewählt, so erhält man das in Bild 5.3.5 wiedergegebene Menü.

Einige der Menüeigenschaften sind nur auf den Verbindungspunkten aktivierbar, andere auf den Kanten selber. Die jeweils gültigen Funktionen sind auf dem Bildschirm im Menü Fett-gedrukt hervorgehoben, die ungültigen erscheinen Fein-gedrukt und können nicht angewählt werden, so daß eine Fehlfunktion ausgeschlossen ist.

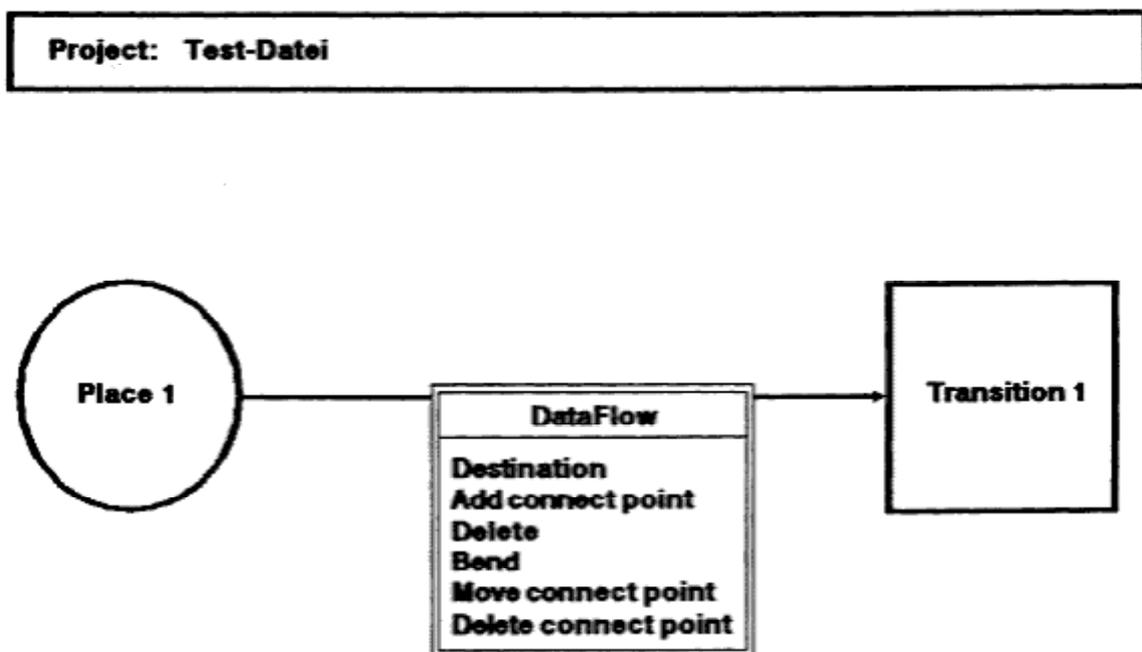


Bild 5.3.5: Das Kantenmenü

5.3.5.1. "Destination"

Dieser Menüpunkt ist während des Zeichnens von Kanten verfügbar. Er kann innerhalb einer Transition oder Stelle angewählt werden, je nachdem, ob die Kante auf einer Stelle oder einer Transition seinen Anfang hat. Das angewählte Objekt wird dann zum Endpunkt der Kante.

5.3.5.2. "Add connect point"

Beim Zeichnen der Kanten kann hiermit ein Verbindungspunkt auf dem Hintergrund gesetzt werden.

Soll ein Verbindungspunkt in eine bereits bestehende Kante hinzugefügt werden, ist hierfür die "Bend"-Funktion vorgesehen (siehe 5.3.5.4).

5.3.5.3. "Delete"

Diese Funktion löscht die angewählte Kante mit allen ihren Verbindungspunkten.

Solange keine weiteren Objekte gelöscht werden, kann sie durch die "Undel ete"-Funktion im Generalmenü wiederhergestellt werden.

5.3.5.4. "Bend"

Mit dieser Funktion wird ein weiterer Verbindungspunkt in eine bestehende Kante eingefügt. Sie ist nur auf einer Kante und nicht auf ihren Verbindungspunkten verfügbar. Der neue Verbindungspunkt kann mit der Maus an seine endgültige Position verschoben werden.

5.3.5.5. "Move connect point"

Diese Funktion kann nur auf einem Verbindungspunkt aktiviert werden.

Hiermit kann ein bestehender Verbindungspunkt mittels der Maus verschoben werden. Beim Erreichen der gewünschten Position ist wieder die Maustaste zu drücken.

5.3.5.6. "Delete connect point"

Auch hier muß sich der Mauszeiger auf einem bestehendem Verbindungspunkt befinden, um diese Funktion aktivieren zu können. Der betroffene Verbindungspunkt wird gelöscht. Er kann jedoch, falls keine weiteren Objekte zwischenzeitlich gelöscht werden, mit Hilfe der "Undel ete"-Funktion im Generalmenü zurückgeholt werden.

5.3.6. Funktionen des Hintergrundtextmenüs

Dieses Menü ist auf dem Hintergrund zu erreichen und dient der nachträglichen Beschriftung von Petrinetzen.

Um die Darstellung des eigentlichen Textes im Petrinetz erfolgt die Zeichnung eines Rahmens, welcher mit seiner rechten oberen Ecke an der Position des Mauszeigers eingefügt wird. Durch Auswahl einer geeigneten Farbe (schwarz) kann der Rahmen jedoch unterdrückt werden, und damit beispielsweise Kanten mit Namen versehen werden.

Die Bearbeitung des Textes erfolgt jeweils unter Verwendung des in Kapitel 5.2 beschriebenen Texteditors.

In Bild 5.3.6.1 sind die vier Funktionen des Menüs zu erkennen.

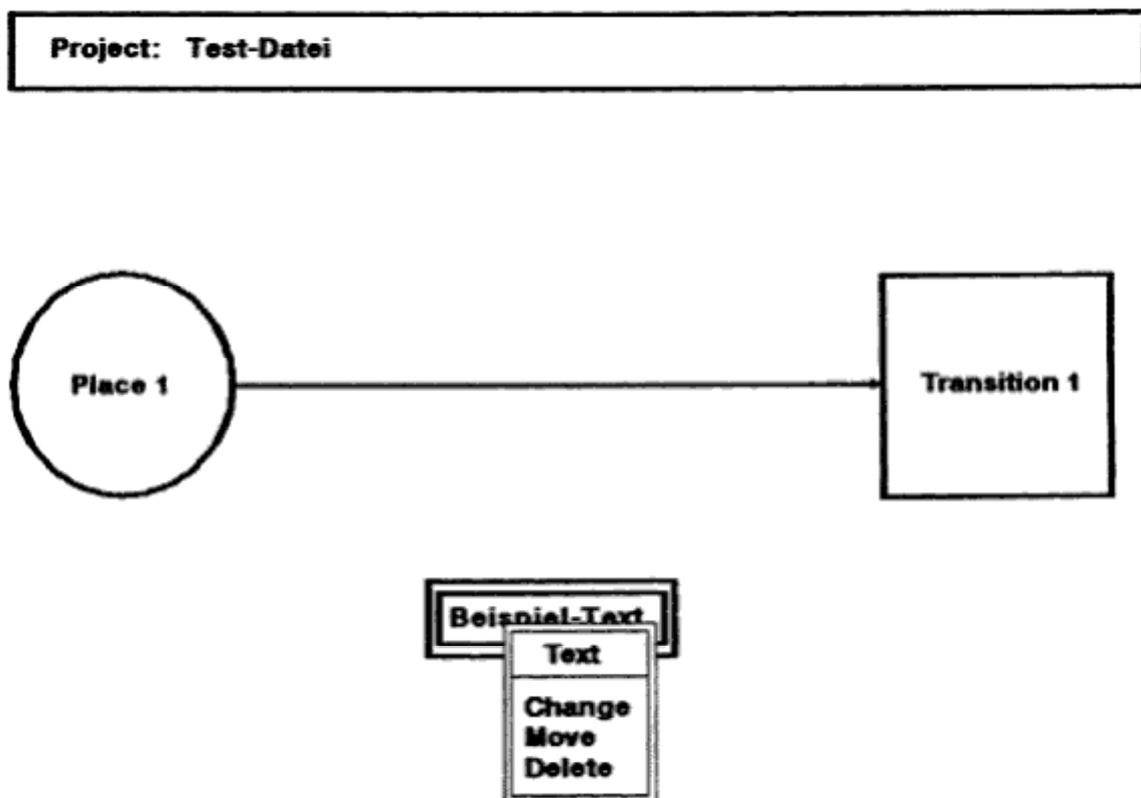


Bild 5.3.6.1: Das Hintergrundtextmenü

5.3.6.1. "Change"

Ein eingegebener Text wird mit Hilfe dieser Funktion weiter bearbeitet und dann an der alten Position erneut gezeichnet.

5.3.6.2. "Move"

Möchte man einen Hintergrundtext verschieben, muß diese Funktion eingesetzt werden. Der gesamte Text wird mit seinem Rahmen verschoben. Aus Zeitgründen wird jedoch während des Verschiebens nur der Rahmen gezeichnet. Bei Erreichen der gewünschten Position ist die Maustaste zu drücken, und der Text wird mitsamt Rahmen dort angebracht.

5.3.6.3. "Delete"

Mit dieser Funktion wird ein Text gelöscht. Wie bei anderen Objekten auch kann er durch die "Undo"-Funktion im Generalmenü zurückgeholt werden, solange kein anderes Objekt gelöscht wurde.

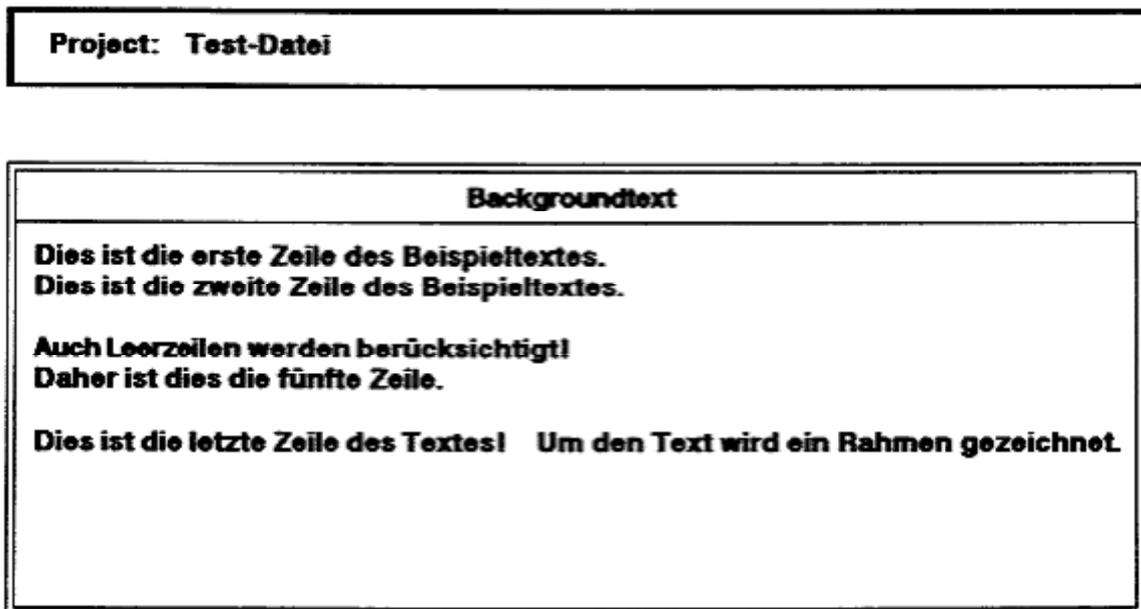


Bild 5.3.6.2: Beispiel einer Texteingabe für den Hintergrundtext

6. Programmiererweiterungen

6.1. Erweiterung des Menüs

Aufgrund seiner Konzeption sind Programmänderungen an dem Petrinetzeditor sehr leicht vorzunehmen. Bild 6.1.1 veranschaulicht die dafür notwendigen Schritte. Zur näheren Erläuterung folgt ein Beispiel, in welchem der Editor um eine neue Funktion erweitert wird. Die dafür erforderlichen Veränderungen werden im einzelnen beschrieben.

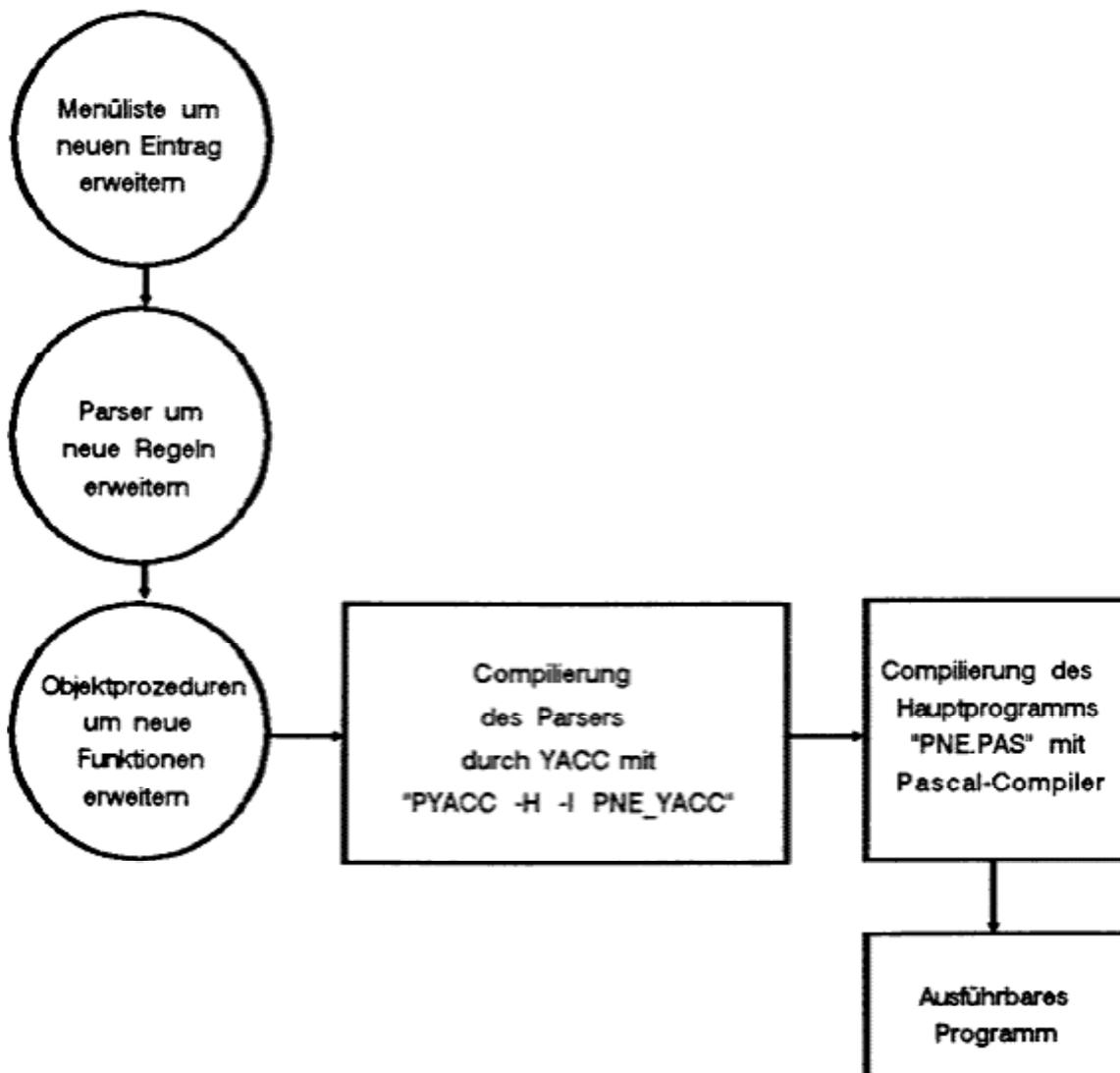


Bild 6.1.1: Vorgehensweise bei Programmiererweiterungen

Als Beispiel diene eine Funktion, welche alle Hintergrundtexte des Petrinetzes ausblendet, ohne sie zu löschen.

Da sich diese Funktion nicht auf ein spezielles Objekt, sondern auf eine ganze Objektart, die Hintergrundtexte, bezieht, erfolgt ihre Einfügung in das General-Menü. Als Eintragsname wird "Hide backgroundtext" gewählt, welcher an der vorletzten Position in das Menü hinzugefügt wird. Die Funktion sei weiterhin immer verfügbar ("EntryActive := true") und erhalte das Token "_HIDE TEXT" zugewiesen.

Hierzu müssen die Programmzeilen

```
General Menu^. EntryName [12] := 'Hide backgroundtext' ;  
General Menu^. EntryActive [12] := true;  
General Menu^. EntryToken [12] := _HIDE TEXT;
```

in der "INIT.PAS"-Datei innerhalb der Initialisierung des General-Menüs ergänzt werden. Da sich die Anzahl der Menüpunkte um eins erhöht, wird die Variable "General Menu^. Number" von dreizehn auf vierzehn gesetzt.

Die komplette Initialisierung des Menüs ist in Bild 6.1.2 abgebildet.

Die Berechnung der neuen Länge und einer eventuell geänderten Breite des Menüfensters übernimmt die Funktion "MenuOpen", wenn das Menü gezeichnet wird. Der Programmierer braucht diese Angaben nicht neu zu berechnen.

Damit ist die neue Funktion vollständig in das General-Menü integriert.

Als nächstes erfolgt die Erweiterung des Parsers.

```
New (General Menue);                                { General -Menue      }

General Menue^. Headl i ne                          := ' General ' ;
General Menue^. Number                              := 14;

General Menue^. EntryName [ 0]                      := ' Undel ete' ;
General Menue^. EntryActi v [ 0]                   := fal se;
General Menue^. EntryToken [ 0]                    := _UNDELETE;

General Menue^. EntryName [ 1]                      := ' Load' ;
General Menue^. EntryActi v [ 1]                    := true;
General Menue^. EntryToken [ 1]                    := _LOAD;

General Menue^. EntryName [ 2]                      := ' Save' ;
General Menue^. EntryActi v [ 2]                    := true;
General Menue^. EntryToken [ 2]                    := _SAVE;

General Menue^. EntryName [ 3]                      := ' New' ;
General Menue^. EntryActi v [ 3]                    := true;
General Menue^. EntryToken [ 3]                    := _NEW;

General Menue^. EntryName [ 4]                      := ' Print' ;
General Menue^. EntryActi v [ 4]                    := true;
General Menue^. EntryToken [ 4]                    := _PRINT;

General Menue^. EntryName [ 5]                      := ' Refresh wi ndow' ;
General Menue^. EntryActi v [ 5]                    := true;
General Menue^. EntryToken [ 5]                    := _REFRESHWI NDOW;

General Menue^. EntryName [ 6]                      := ' Grid off' ;
General Menue^. EntryActi v [ 6]                    := true;
General Menue^. EntryToken [ 6]                    := _GRID;

General Menue^. EntryName [ 7]                      := ' Di splay gri d' ;
General Menue^. EntryActi v [ 7]                    := true;
General Menue^. EntryToken [ 7]                    := _HI DEGRI D;

General Menue^. EntryName [ 8]                      := ' Datafl ows ri ght-angl ed' ;
General Menue^. EntryActi v [ 8]                    := true;
General Menue^. EntryToken [ 8]                    := _DFRI GHTANGLED;

General Menue^. EntryName [ 9]                      := ' Smooth datafl ows' ;
General Menue^. EntryActi v [ 9]                    := true;
General Menue^. EntryToken [ 9]                    := _SMOOTH;

General Menue^. EntryName [10]                      := ' Symbol name: bottom' ;
General Menue^. EntryActi v [10]                    := true;
General Menue^. EntryToken [10]                    := _SYMBOLNAMECENTER;

General Menue^. EntryName [11]                      := ' Defaul ts' ;
General Menue^. EntryActi v [11]                    := true;
General Menue^. EntryToken [11]                    := _DEFAULTS;

General Menue^. EntryName [12]                      := ' Hi de backgroundtext' ;
General Menue^. EntryActi v [12]                    := true;
General Menue^. EntryToken [12]                    := _HI DETEXT;

General Menue^. EntryName [13]                      := ' Qui t' ;
General Menue^. EntryActi v [13]                    := true;
General Menue^. EntryToken [13]                    := _QUIT;
```

Bild 6.1.2: Die Erweiterung des General-Menus

6.2. Erweiterung des Parsers

Für die Verwaltung des neuen Menüpunktes ist eine entsprechende Ergänzung der Grammatik, welche den Parser bestimmt, erforderlich. Die Datei, in welcher sich die Regeln für den Parser befinden, trägt den Namen "PNE_YACC.P". In ihr sind die folgenden Änderungen vorzunehmen:

Zunächst wird das neue Token "_HI DETEXT" in dem Deklarationsteil vereinbart:

```
%Token _HI DETEXT.
```

Nun werden die Regeln für diesen Menüpunkt hinzugefügt. Dafür muß die bereits existierende Produktionen "General Exec" geändert werden. Die Regel wird um die Substitution

```
| _HI DETEXT Hi deTextOp
```

erweitert. In Bild 6.2.1 ist die vollständige "General Exec"-Produktion aufgeführt.

```
General Exec      :  _UNDELETE      Undel eteOp
                   |  _LOAD        LoadOp
                   |  _SAVE        SaveOp
                   |  _NEW         NewOp
                   |  _PRI NT      Pri ntOp
                   |  _REFRESHWI NDOW RefreshWi ndowOp
                   |  _GRI D       Gri dOp
                   |  _HI DEGRI D  Hi deGri dOp
                   |  _DFRI GHTANGLED DFRI ghtAngl edOp
                   |  _SMOOTH      SmoothOp
                   |  _SYMBOLNAMECENTER Symbol NameCenterOp
                   |  _DEFAULTS    Defaul tsOp
                   |
                   |  _HI DETEXT    Hi deTextOp
                   |
                   |  _QUI T        Qui top
                   |  _MENUENOTOK
                   {
                   Begin { General Exec }
                   MenueCl ose (General Menue, MouseXMenue, MouseYMenue)
                   End; { General Exec }
                   }
                   ;
```

Bild 6.2.1: Die neue "GeneralExec"-Produktion

Das Nonterminal-Symbol "Hi deTextOp" ist eine neue Produktion, die in Bild 6.2.2 zu sehen ist:

```
Hi deTextOp      :  
                  {  
                    Begin { Hi deTextOp }  
                      MenueClose (General Menue, MouseXMenue, MouseYMenue);  
                      Hi deTextProc;  
                    End; { Hi deTextOp }  
                  ;
```

Bild 6.2.2: Die Produktion "HideTextOp"

Entsprechend ihrer Position im General-Menü wird sie zwischen "DefaultOp" und "QuitOp" in die Datei eingebunden. Im Pascalteil der Regel befindet sich der Aufruf der Prozedur "HideTextProc". Diese Prozedur wird nach der Erweiterung des Parsers zu den Objektprozeduren hinzugefügt.

6.3. Erweiterung der Objektprozeduren

Da die "Hide backgroundtext"-Funktion in dem General-Menü zu finden ist, erfolgt die Deklaration der Prozedur "HideTextProc" in der Datei "GENERAL.PAS".

Die gesamte Prozedur ist in Bild 6.3 wiedergegeben und wird zwischen den Prozeduren "DefaultProc" und "QuitProc0" eingefügt. Wenn sie vom Parser aufgerufen wird, setzt sie die Farben für den Hintergrundtext und seinen Rahmen auf null (schwarz) und baut den gesamten Bildschirminhalt neu auf. Dadurch sind die Hintergrundtexte nicht mehr zu erkennen. Sollen sie wieder zu sehen sein, können die beiden Farben mit der "Default"-Funktion geändert werden (siehe 5.3.1.12).

Damit sind alle Programmänderungen vorgenommen worden, und das Programm kann nun neu compiliert werden.

```
Procedure HideTextProc;
{
  Blendet Hintergrundtexte ein/aus
}

Begin { HideTextProc }
  ChangeFlag := true;           { Petri netz wurde veraendert           }
  ColorBTF   := 0;             { Farbe BackgroundText-Rahmen auf schwarz }
  ColorBTT   := 0;             { Farbe BackgroundText-Text auf schwarz  }

  RefreshWindow;               { Fenster neu aufbauen           }
End; { HideTextProc }
```

Bild 6.3: Die Prozedur "HideTextProc"

6.4. Erstellung des ausführbaren Programms

Die Erstellung des ausführbaren Programmes besteht aus zwei weiteren Schritten:

- Compilierung des Parsers mit YACC, und
- Compilierung des Hauptprogrammes mit Turbo-Pascal.

Um die Grammatik des Parsers in ein Pascalprogramm zu übersetzen, ist folgender Aufruf zu benutzen:

```
"PYACC -H -I PNE YACC".
```

Der Compilergenerator YACC erstellt dann aus der Datei "PNE_YACC.P" die Pascaldatei "PNE_YACC.PAS". Dabei wird ebenfalls die Datei "PNE_YACC.H" sowie die Datei "PNE_YACC.I" erzeugt.

In ersterer befinden sich die Konstantendeklaration der Token und in der zweiten Datei einige YACC-spezifische Informationen, wie etwa Anzahl der Token und Regeln, Ausnutzung des zur Verfügung stehenden Tabellenplatzes innerhalb von YACC, und so weiter.

Schließlich wird die Datei "PNE.PAS" mit Turbo-Pascal 5.5 compiliert. Dabei müssen die Option "Var-string checking" auf "Relaxed" und die Option "Boolean evaluation" auf "Short circuit" stehen.

Die dabei benötigten Dateien lauten:

- PNE.PAS { Hauptprogramm, bindet alle nötigen Dateien ein }
- VARIABLE.PAS { Initialisierung der globalen Variablen }
- PNE_YACC.PAS { Parsersteuerung }
- SCANNER.PAS { Routinen für den Scanner }
- SCREEN.PAS { Verwaltung des Bildschirms }
- DISK.PAS { Verwaltung der Datei-Ein- und Ausgabe }
- GENERAL.PAS { Verwaltung von allgemeinen Routinen }
- PLACE.PAS { Verwaltung von Places }
- TRANS.PAS { Verwaltung von Transitionen }
- DATAFLOW.PAS { Verwaltung von DataFlows }
- BGTEXT.PAS { Verwaltung von BackgroundText }
- INIT.PAS { Initialisierung des Programmes }

Alle notwendigen Dateien werden von "PNE. PAS" selbstständig eingebunden, und als Ergebnis erhält man die ausführbare Datei "PNE. EXE".

7. Zusammenfassung und Ausblick

Aufgabe der vorliegenden Arbeit ist die Entwicklung eines syntaxgesteuerten graphischen Petrinetzeditors, der auf einem Personal-Computer mit Mausunterstützung implementiert wurde.

Im Rahmen der Aufgabenstellung wurden hierzu zunächst Petrinetze allgemein untersucht.

Petrinetze sind ein wichtiges Modell zur Beschreibung von Systemen. Entsprechend ihrer Interpretation werden sie von der Petrinetztheorie in verschiedene Netzklassen eingeteilt. Zwei wichtige Vertreter von Netzklassen sind die Prädikat/Transitionsnetze und die NSL-Netze. Im Vergleich mit der Automatentheorie besitzen Petrinetze wichtige Vorteile. Insbesondere ermöglichen sie die Analyse unabhängiger und nebenläufiger Vorgänge. Verdeutlichen kann man sich den Begriff des Petrinetzes durch sein Schaltverhalten innerhalb des Markenspieles. Die graphische Darstellung von Petrinetzen erfolgt für Stellen als Kreise, für Transitionen als Rechtecke und für Kanten als Pfeile.

Für die Syntaxsteuerung wurden Wirkungsweise und Vorteile von Parsersteuerungen diskutiert.

Bei der Parsersteuerung hat der Parser die Aufgabe, den Programmverlauf an Hand einer Grammatik zu bestimmen. Um einen solchen Parser zu erzeugen, wird der Compilergenerators YACC benutzt. Er generiert aus einer BNF-ähnlichen Grammatik einen Parser in Form eines Pascalprogrammes.

Auf diesen Grundlagen aufbauend, wurde der Editor implementiert. Er ist in fünf Teile gegliedert: Scanner, Parser, Bildschirmsteuerung, Objektprozeduren, Datenübertragungsprozeduren und die Druckprozedur. Die Steuerung des Editors erfolgt durch den von YACC erstellten Parser. Dieser nimmt Benutzereingaben unter Zuhilfenahme eines Scanners entgegen. Ein Teil der lexikalischen Analyse wurde dabei vom Parser auf den Scanner übertragen, der dadurch beispielsweise in der Lage ist, einen Großteil der Menüverwaltung zu übernehmen. Mit dem Benutzer tritt der Parser durch Aufruf von entsprechenden Prozeduren in einen interaktiven Dialog. Diese Prozeduren verwalten das Zeichnen der Objekte, das Laden und Speichern von Dateien und so weiter und spiegeln für den Benutzer die Reaktion des Parsers wieder. Als Beispiel für einen typischen Programmverlauf wurde das Hinzufügen einer Stelle durchgespielt.

Der Anwendung des Editors, angefangen von dem Start über die Bedienung des integrierten Texteditors bis zu der Beschreibung aller zur Verfügung stehenden Menüfunktionen, wurde ein umfangreiches Kapitel gewidmet.

Der Benutzer erstellt Petrinetze dabei am Bildschirm unter Zuhilfenahme einer Maus. Durch Betätigen der linken Maustaste können Menüs eröffnet werden, welche die Generierung und Bearbeitung von Objekten erlauben, wie Stellen, Transitionen und Kanten. Zur Beschriftung des Petrinetzes ist die Möglichkeit vorgesehen, Texte an beliebiger Stelle in das Petrinetz einzufügen. Im Hinblick auf den NSL-Compiler können den Objekten Textblöcke zugeordnet werden, um die Objekteigenschaften mit Hilfe der NSL-Sprache näher zu spezifizieren. Die Textblöcke werden mit dem Texteditor erzeugt und verändert. Dabei werden sie bereits auf ihre syntaktische Korrektheit bezüglich der NSL-Syntax geprüft. Im Falle eines Syntaxfehlers erfolgt die Positionierung des Cursors dann automatisch an der entsprechenden Stelle im Textblock, um den Text erneut bearbeiten zu können. Der Ausdruck eines Petrinetzes geschieht mit Hilfe einer PostScript-Datei.

Zur Unterstützung der Programmwartung erfolgte die Deskription von Programmweiterungen an Hand eines ausführlichen Beispiels. Zuerst wird das Menü erweitert, dann die Grammatik, um schließlich noch die entsprechende Objektprozedur einzufügen. Danach wird der Parser mit YACC generiert und die "PNE.PAS"-Datei mit TurboPascal compiliert.

Insgesamt bildet der Editor ein abgeschlossenes Werk. Zur Weiterentwicklung bieten sich einige Funktionen an, deren Realisierung sich in der Zukunft empfiehlt:

- Implementierung von Inspektionskanten.
Inspektionskanten werden innerhalb der NSL-Netze zum lesenden Zugriff auf Marken benutzt.
- Unterstützung einer sogenannten Log-Datei.
In dieser Datei werden alle Benutzereingaben dokumentiert und während des Programmlaufes abgespeichert.

- Erweiterung des Eingabebereiches über den Bildschirmausschnitt hinaus durch Einführung eines beweglichen Sichtfensters (Scrolling).
- Einführung einer Block-Funktion, welche es gestattet, mehrere Stellen und Transitionen zu einem Block zusammenzufassen. Mit diesem Block können dann verschiedene Operationen durchgeführt werden, wie zum Beispiel Kopieren und Verschieben der Objekte.
- Unterstützung der Verfeinerung und Einbettung von Petrinetzen durch Implementierung verschiedener unabhängiger Ebenen, in welchen jeweils ein autarkes Petrinetz entworfen werden kann.
- Entwicklung eines Filters, welcher die abgespeicherten Petrinetzdateien in Dateien für den NSL-Compiler umwandelt, indem er die nötigen Informationen aus der Petrinetzdatei extrahiert und in die Syntax der NSL-Sprache umformt.
- Interaktive Simulation eines Markenspieles für das eingegebene Petrinetz auf dem Bildschirm.

Die Verwirklichung dieser Funktionen wird durch den Einsatz der Parsersteuerung und die daraus resultierende Programmstruktur wesentlich erleichtert. Die dafür nötigen Grundlagen finden sich eingehend und vollständig dokumentiert.

8. Literatur

- /AHOS88/ Aho Alfred V., Sethi Ravi, Ullman Jeffrey D.
Compilerbau
Addison-Wesley Publishing Company, 1988
- /JOHN75/ Johnson S. C.
YACC - yet another compiler compiler
Computing Science Technical Report 32
- /KROE89/ Krönert Günther, Laubner Georg, Mannes Hans-G.
Spezifikation, Prototyping und Implementierung von
interaktiven Systemen unter Verwendung von
attributiven Grammatiken
Informatik Spektrum 1989, S. 225 ff.
- /REI S82/ Reisig Wolfgang
Petri-Netze - Eine Einführung
Springer Verlag Berlin, 1982
- /REI S85/ Reisig Wolfgang
Systementwurf mit Netzen
Springer Verlag Berlin, 1985
- /WI RT83/ Wirth Niklaus
Algorithmen und Datenstrukturen
B.G. Teubner Verlag Stuttgart, 1983
- /WI RT85/ Wirth Niklaus
Systematisches Programmieren
B.G. Teubner Verlag Stuttgart, 1985
- /WI RT86/ Wirth Niklaus
Compilerbau
B.G. Teubner Verlag Stuttgart, 1986
- /ZUSE80/ Zuse Konrad
Petri-Netze aus der Sicht des Ingenieurs
Vieweg Verlag Braunschweig, 1980