Compilerentwicklung in Miranda – ein Praktikum in funktionaler Programmierung

Andreas Maaßen
Lehrstuhl für Informatik II, RWTH Aachen
Ahornstraße 55, D-5100 Aachen
maassen@informatik.rwth-aachen.de

Juli 1990

Zusammenfassung

Im Sommersemester 1990 führte der Autor ein Softwarepraktikum für Studenten des 4.Fachsemesters Informatik durch. In diesem Praktikum wurde – was für das Grundstudium Informatik in Aachen ein Novum war – keine imperative Programmiersprache, sondern die funktionale Sprache Miranda[1] verwendet. Gegenstand des Praktikums war die Entwicklung eines Compilers für eine kleine selbstdefinierte Programmiersprache. Dieser Bericht beschreibt die Durchführung des Praktikums und die dabei gemachten Erfahrungen, außerdem enthält er die vollständigen Aufgabenstellungen des Praktikums und Musterlösungen dazu.

[1]Miranda ist eingetragenes Warenzeichen der Firma Research Software Ltd.

1 Einführung

Die vorliegende Arbeit ist eine Beschreibung des Softwarepraktikums ***Funktionales Programmieren in Miranda***, das im Sommersemester 1990 vom Lehrstuhl für Informatik II durchgeführt wurde. Zielgruppe für dieses Praktikums waren Studenten des 4. Fachsemesters Informatik, für die der Studienplan ein Softwarepraktikum als Pflichtveranstaltung vorsieht. Die Studenten hatten in den vorhergehenden Semestern aus dem Bereich der Informatik einführende Vorlesungen zu den Themen

- Programmierung
- Algorithmen und Datenstrukturen
- Aufbau von Rechenanlagen
- Automatentheorie und Formale Sprachen

gehört. Als erste Programmiersprache im Rahmen des Studiums hatten sie im ersten Semester Modula-2 kennengelernt und diese Sprache im Rahmen der Übungen zu zwei Vorlesungen auch praktisch eingesetzt. Ein großer Teil der Praktikumsteilnehmer hatte überdies einen Kursus in FORTRAN V absolviert, und viele hatten Vorkenntnisse in Pascal bzw. Turbo-Pascal. Die meisten Teilnehmer kannten bis dahin als einziges Betriebssystem MS-DOS bzw. vergleichbare Systeme.

Zum ersten Mal wurde im Studiengang Informatik der RWTH Aachen eine funktionale Programmiersprache für eine Veranstaltung des Grundstudiums verwendet. Bis dahin waren Vorlesungen und Kurse über applikative Sprachen nur für das Hauptstudium angeboten worden, wobei die Studenten gewöhnlich erst eine Vorlesung über die mathematischen Grundlagen und Implementierungstechniken für funktionale Sprachen hörten und im Anschluß daran einen Kursus in einer funktionalen Programmiersprache absolvierten. Diese theoretischen Vorkenntnisse fehlten den Praktikumsteilnehmern.

2 Organisation

Das Praktikum zerfiel in zwei Phasen:

- einen einwöchigen Steilkurs in der vorlesungsfreien Zeit, in dem die Programmiersprache Miranda gelernt wurde;
- eine Folge von 7 Praktikumsaufgaben, die über das Semester verteilt gestellt wurden.

Das Miranda-System, mit dem im Praktikum gearbeitet wurde, läuft unter UNIX[2]. Dieses Betriebssystem war den meisten Praktikumsteilnehmern noch unbekannt, als bekannt konnte lediglich MS-DOS (bzw. vergleichbare Betriebssysteme) vorausgesetzt werden. Daher wurde eine schriftliche Kurzeinführung in UNIX, den Editor vi und das Dateiübertragungsprogramm ftp ausgearbeitet und den Praktikumsteilnehmern vorab zugeschickt. Zusätzlich wurden einige ergänzende Erläuterungen in der ersten Unterrichtsstunde des Miranda-Kurses gegeben. Außerdem erhielt jeder vorab eine schriftliche, um ein Stichwortverzeichnis ergänzte Version des Online-Handbuches aus dem Miranda-System.

Der Steilkurs wurde innerhalb einer Woche als ganztägige Veranstaltung durchgeführt. Dabei wurde morgens und am frühen Nachmittag jeweils eine Doppelstunde Unterricht gehalten, die übrige Zeit des Tages war jeweils für praktische Übungen am Rechner vorgesehen. Für das Unterrichten einer konventionellen Programmiersprache wäre eine solche Vorgehensweise kaum praktikabel gewesen, da man zu Anfang des Kurses gewiß Schwierigkeiten gehabt hätte, den Kursteilnehmern soviel von der Programmiersprache zu vermitteln, daß sie lauffähige Programme hätten schreiben können. Miranda bietet in dieser Hinsicht große Vorteile, da das System interaktiv ist und Miranda-Programme im Prinzip einen sehr einfachen Aufbau haben.
Als Übungsmaterial erhielten die Kursteilnehmer eine Liste mit Programmieraufgaben unterschiedlichen Schwierigkeitsgrades. Die Bearbeitung dieser Aufgaben war jedoch keine Pflicht, und sie standen in keiner Beziehung zum eigentlichen Praktikum.

Für die Dauer des Steilkurses konnten erfreulicherweise soviele Rechner-Arbeitsplätze zur Verfügung gestellt werden, daß jeder Kursteilnehmer an einer eigenen Konsole arbeiten konnte.

Das Praktikum selber umfaßte die Bearbeitung von 7 Programmieraufgaben, die über das Semester verteilt gestellt wurden. Die Bearbeitungszeit für eine Aufgabe betrug durchschnittlich drei Wochen, was bei einer Semesterdauer von ungefähr 12 Wochen natürlich nur dadurch zu realisieren war, daß sich die Bearbeitungszeiten der einzelnen Aufgaben überlappten. Obwohl das Praktikum als Projektpraktikum durchgeführt wurde und die einzelnen Aufgaben aufeinander aufbauten, konnte diese Organisationsform problemlos gewählt werden, da das interaktive Miranda-System das Entwickeln und Testen von Modulen auch (fast) ohne Verwendung von Testtreibern und -stummeln unterstützt. Durch die Überlappung der Bearbeitungszeiten wurde den Praktikumsteilnehmern eine vergleichsweise flexible Einteilung ihrer Arbeitszeit ermöglicht.

Die Bearbeitung der Aufgaben erfolgte in Zweiergruppen. Kriterium für den Erwerb des Praktikumsscheines war die Testierung aller 7 Aufgaben; eine Klausur wurde nicht geschrieben.

[2]Derzeit wird Miranda nur für UNIX-Systeme angeboten, und Research Software Ltd. plant nach eigener Auskunft auch nicht, Miranda für andere Betriebssysteme anzubieten.

3 Wahl der Aufgabenstellung

Bei der Wahl der Aufgabenstellung waren drei Gesichtspunkte von Bedeutung:

- Das Praktikum sollte möglichst als Projektpraktikum durchgeführt werden, damit die Praktikumsteilnehmer insbesondere auch die sinnvolle Verwendung des Modulkonzeptes von Miranda kennenlernen und einen Eindruck von der Entwicklung eines Miranda-Programms nicht-trivialen Umfangs bekommen.
- Entsprechend dem wichtigsten Einsatzgebiet applikativer Programmiersprachen sollte die Aufgabenstellung aus dem Bereich der symbolischen Datenverarbeitung stammen.
- Die Aufgabenstellung sollte so geartet sein, daß für ihre Lösung möglichst viele Konzepte der funktionalen Programmierung in sinnvoller Weise einsetzbar waren. Zu diesen Konzepten wurden hier vor allem die folgenden gezählt:
  - Algebraische Datentypen in Verbindung mit rekursiven Funktionen
  - Funktionen höheren Typs, insbesondere in der Verwendung als Werkzeuge
  - Lazy Evaluation
  - Polymorphie

Als Gegenstand des Praktikums wurde die Übersetzung einer selbstdefinierten kleinen Programmiersprache in Code für eine ebenfalls selbstdefinierte abstrakte Maschine, die wiederum in Miranda zu implementieren war, gewählt. Dieses Thema entsprach (fast) allen obengenannten Anforderungen:

- Der Compilerbau gehört zweifelsohne in den Bereich der symbolischen Datenverarbeitung.
- Algebraische Datentypen konnten in ihren typischen Anwendungen, nämlich zur Implementierung von Bäumen (Syntaxbäumen) und varianten Verbunden (Befehle der abstrakten Maschine) verwendet werden. Rekursive Funktionen auf Bäumen sind ein geeignetes Mittel, um die Erzeugung des Maschinencodes zu einem Syntaxbaum zu beschreiben.
- Parser sind Funktionen, die Zeichenketten (Programme in Text-Darstellung) auf Syntaxbäume abbilden. Die von Reade [Reade] vorgeschlagene Technik, Parser mit Hilfe geeigneter Werkzeuge induktiv über den Aufbau der rechten Regelseiten kontextfreier Grammatiken zu programmieren, bot ein gutes Beispiel für die sinnvolle Verwendung von Funktionen höheren Typs als Werkzeug zur Erzeugung von Parsern.

- Das Zusammenspiel von Scanner und Parser, bei dem letzterer ja aus Gründen der Speicherplatzeffizienz erst bei Bedarf das nächste Token vom Scanner anfordern soll, bot ein gutes Beispiel für die eleganten Möglichkeiten des "Verklebens" von Programmen und der Speicherverwaltung, die durch Lazy Evaluation verfügbar werden. In diesem Zusammenhang sei auch auf eine Arbeit von Hughes [Hughes] hingewiesen, der diesen Gesichtspunkt ausführlich erörtert.

Die nachfolgenden Abschnitte geben die Aufgabenstellungen des Praktikums wieder. Angefügt sind jeweils Listings der zu den einzelnen Aufgaben vorgegebenen Module und Programmstücke.

Warnung! Bei einigen hier wiedergegebenen Miranda-Programmen wurden aus drucktechnischen Gründen die Zeilenumbrüche geringfügig verändert. Da Einrückungen in Miranda semantisch relevant sind, kann es sein, daß einige der Programme mit den hier angegebenen Umbrüchen nicht übersetzbar sind. Die nötigen Anpassungen sind jedoch, wie gesagt, gering.

4 Aufgabe 1: Lexikalische Analyse

4.1 Problembeschreibung

Für die Übersetzung eines Programms in eine andere Sprache ist es sinnvoll, als erstes die Eingabe, die i. allg. als unstrukturierte Zeichenfolge (ASCII-Datei) vorliegt, in ihre sinntragenden Einheiten, die sogenannten ***Lexeme***, zu unterteilen. Lexeme sind z. B. Schlüsselwörter, Zahlen, frei gewählte Bezeichner und Operationssymbole (auch zusammengesetzte). Leerzeichen, Kommentare und Umbruchzeichen sind keine Lexeme, denn sie haben keine eigene Bedeutung in der Programmiersprache, sondern dienen nur der Strukturierung des Textes bzw. seiner Lesbarkeit und Verständlichkeit.

Diese Strukturierung des Quellprogramms in seine sinntragenden Einheiten nennt man ***lexikalische Analyse***, und den entsprechenden Teil des Compilers bezeichnet man als *Scanner*.

Die Entscheidung, an welcher Stelle ein Lexem aufhört und das nächste beginnt, ist unter Umständen nicht trivial. So kann man z. B. die Zeichenfolge "<=" wahlweise als ein Lexem, das den Kleiner-gleich-Operator darstellt, oder als Folge zweier Lexeme, die den Kleiner-Operator und den Gleichheitsoperator darstellen, interpretieren. Hier gilt i. allg.[3] das ***principle of the longest match*** , d. h. man strukturiert den Text, indem man – beginnend mit dem ersten Zeichen des Textes – Lexeme maximaler Länge bildet. Dementsprechend faßt man die Zeichenfolge "<=" gewöhnlich als ein Lexem auf.

Lexeme, die aus der Sicht der kontextfreien Syntax austauschbar sind, faßt man in Klassen zusammen. Z. B. bilden alle frei gewählten Bezeichner eine Klasse, ferner alle

[3]Eine Ausnahme von dieser Regel ist z. B. FORTRAN

Literale eines Typs (in unserem Beispiel gibt es nur die Klasse der ganzen Zahlen), und die relationalen sowie die arithmetischen Operatoren gleicher Priorität bilden ebenfalls jeweils eine Klasse. Jedes Schlüsselwort bestimmt eine eigene Klasse, die jeweils ein Element enthält.

Der Scanner gibt allerdings keine Lexeme – also Zeichenketten – aus, sondern ***Symbole***, die aus einem sogenannten ***Token*** und einem ***Attribut*** bestehen. Ein Token ist eine Kennung, die anzeigt, zu welcher Klasse von Lexemen ein Objekt gehört. Das Attribut gibt Auskunft darüber, welchen Wert das Lexem darstellt (welche Zahl, welchen Bezeichner, welchen Operator). Bei Lexemklassen, die nur ein Element enthalten, enthält das Symbol kein Attribut.

Die Beschreibung der Syntax einer Programmiersprache besteht meist aus drei Teilen:

- Die ***Mikrosyntax*** beschreibt den Aufbau der Lexeme[4].

- Die ***kontextfreie Syntax*** definiert die kontextfrei beschreibbaren Teile der Syntax, abstrahiert dabei aber vom Aufbau der Lexeme. Diese werden als Terminalsymbole der kontextfreien Grammatik aufgefaßt.

- Die kontextfreie Syntax wird ergänzt durch ***kontextsensitive Regeln***, in denen kontextfrei nicht beschreibbare Eigenschaften der Programmiersprache formuliert werden. Ein Beispiel hierfür ist die Forderung, daß Bezeichner vor ihrer Verwendung deklariert werden müssen.

In Büchern über Programmiersprachen werden Mikrosyntax und kontextfreie Syntax meist in einer einzigen kontextfreien Grammatik zusammengefaßt, und die kontextsensitiven Regeln werden oft nur informell angegeben.

4.2 Die Syntax der Sprache PSA

Die Syntax einer kleinen Programmiersprache PSA sei durch die folgenden Regeln beschrieben:

4.2.1 Kontextfreie Syntax

Terminalsymbole erscheinen in Fettschrift, Nichtterminalsymbole in Schrägschrift, metasprachliche Symbole der EBNF in normaler Schrift:

[4]Dieser Teil der Syntax wird vom Scanner behandelt.

Program	::=	*Declaration** *Statement_Sequence* **.**
Declaration	::=	**var Variable Variable* ;**
Statement_Sequence	::=	*Statement**
Statement	::=	**Variable :=** *Expression* **if** *BExpression* **then** *Statement_Sequence* **else** *Statement_Sequence* **end** **while** *BExpression* **do** *Statement_Sequence* **end**
Expression	::=	**Variable** **Number** (*Expression* **ArithOp** *Expression*)
BExpression	::=	*Expression* **RelOp** *Expression*

4.2.2 Mikrosyntax

- Eine Variable (**Variable**) besteht aus einem Buchstaben, gefolgt von beliebig vielen Buchstaben und Ziffern. Schlüsselwörter dürfen nicht als Variablen verwendet werden.
- Eine Zahl (**Number**) besteht aus einer nicht-leeren Folge von Ziffern.
- Als arithmetische Operatoren (**ArithOp**) sind zugelassen:
  `+ - * /`
- Als relationale Operatoren (**RelOp**) sind zugelassen:
  = /= < <= > >=

4.2.3 Kontextsensitive Regeln

- Keine Variable darf mehr als einmal im Deklarationsteil vorkommen.
- Jede im Anweisungsteil verwendete Variable muß im Deklarationsteil deklariert sein.

4.3 Aufgabenstellung

Programmieren Sie einen Scanner für PSA-Programme. In dem Verzeichnis

`/home/alpha/maassen`

finden Sie eine Datei `psa_lex.m`, die die Definition des Typs der Token und die Spezifikation einer Funktion `lex` enthält. Implementieren Sie diese Funktion. Der Scanner

soll im *panic mode* arbeiten, d. h. er soll beim ersten Fehler mit einer entsprechenden Meldung abbrechen. In dem gleichen Verzeichnis finden Sie auch die Datei `p1.psa`, die ein Beispielprogramm in der Sprache PSA enthält.

4.4 Vorgegebene Dateien

4.4.1 Die Datei `psa_lex.m`

```
|| Definition der Symbolklassen und des Scanners
|| fuer die Modellsprache PSA

%export string arithop relop token lex

string == [char]

arithop ::= Plus | Minus | Times | Div

relop ::= EQ | NE | LT | LE | GT | GE

token ::=
          ArithOp arithop | AssignOp | ClosePar | DoSym |
          ElseSym | EndSym | FullStop | Ident string |
          IfSym | Num num |OpenPar | RelOp relop | Semicolon |
          ThenSym | VarSym | WhileSym

lex :: [char] -> [token]
```

4.4.2 Die Datei `p1.psa`

```
var x;
var y z;
x := 0
while x < 5 do
        x := (x + 2)
end
if x >= 1 then
        y := 3
else
        z := 2
end.
```

5 Aufgabe 2: Syntaktische Analyse

5.1 Problembeschreibung

5.1.1 Die syntaktische Analyse

Die erste Phase eines Compilers – die lexikalische Analyse – wurde in der vorhergehenden Aufgabe behandelt. An sie schließt sich die ***syntaktische Analyse*** an. In dieser Phase wird der Ableitungsbaum des Programms entsprechend der kontextfreien Syntax aufgebaut. Dies dient zwei Zielen:

- Es wird überprüft, ob das Quellprogramm den kontextfrei beschreibbaren Regeln für den Aufbau von Programmen genügt.
- Der Syntaxbaum ist die Grundlage der weiteren Arbeit des Compilers. An ihm werden die noch fehlenden Überprüfungen der kontextsensitiven Bedingungen für den Programmaufbau durchgeführt, und auch die Erzeugung des Zielcodes wird induktiv über den Aufbau des Syntaxbaumes beschrieben.

Der Teil des Compilers, der diesen Syntaxbaum aufbaut, heißt *Parser*.

Statt des Syntaxbaums, der der sogenannten ***konkreten Syntax*** des Programms, also der Syntax des Quelltextes, entspricht, baut der Parser normalerweise einen ***abstrakten Syntaxbaum*** auf, der von allem, was nur der besseren Lesbarkeit oder der Textstrukturierung dient, abstrahiert. Zum Beispiel sind an dem Programmstück

```
while i < 10 do i := i * i end
```

eigentlich nur drei Dinge wichtig:

- Es handelt sich um eine While-Anweisung;
- das zu testende Prädikat ist `i < 10`;
- der Schleifenkörper besteht aus der Anweisung `i := i * i`.

Dementsprechend kommen in einem abstrakten Syntaxbaum die Terminalsymbole `while`, `do` und `end` nicht vor, sondern nur eine Markierung, die anzeigt, daß es sich um eine While-Schleife handelt, sowie die (abstrakten) Syntaxbäume des Prädikates und des Schleifenkörpers. Zur Darstellung von Syntaxbäumen in Miranda bieten sich algebraische Datentypen an. Das obengenannte Sprachkonstrukt While-Schleife würde dabei z. B. durch einen Konstruktor `While` eines algebraischen Datentyps `stmt_type` dargestellt, wobei `While` zwei Parameter bekäme: einen vom Typ der boole'schen Ausdrücke und einen vom Typ der Anweisungssequenzen.

5.1.2 Parser-erzeugende Werkzeuge

Ein Parser ist ein Programm (eine Funktion), das eine Liste von Token auf einen abstrakten Syntaxbaum abbildet.

Natürlich ist es wünschenswert, einen Parser modular aufzubauen, und zwar möglichst entsprechend der Struktur der kontextfreien Regeln, die die Programmstruktur beschreiben. Dies erhöht nicht nur die Übersichtlichkeit und Verständlichkeit des Programms, sondern erleichtert auch die Anpassung des Parsers an eventuelle Änderungen der Grammatik.

Diese Idee der Modularisierung motiviert die Bildung eines Begriffs, der hier *Teil-Parser* genannt werden soll. Ein Teil-Parser erzeugt den Syntaxbaum für ein Sprachkonstrukt der zu analysierenden Sprache, bildet also eine Liste von Token auf ein Stück Syntaxbaum (bzw. eine Fehlermeldung, falls der Strukturierungsversuch fehlschlägt) und den Teil der Tokenliste, der nicht für den Aufbau dieses Syntaxbaumes verbraucht wurde, ab. (Ein Parser ist also im wesentlichen ein spezieller Teil-Parser, bei dem man verlangt, daß nach Beendigung seines Laufes die Tokenliste ganz aufgebraucht ist.)

Dieser Ansatz läßt sich in Miranda praktisch unmittelbar in ein Programm übertragen, da Miranda Funktionen höheren Typs zuläßt: Man definiert elementare Parser, die einzelne Terminalsymbole erkennen, sowie Funktionen höheren Typs, die gegebene Parser entsprechend des Konstrukten zur Bildung rechter Regelseiten zu neuen Parsern zusammensetzten.

5.1.3 Verzahnung von lexikalischer und syntaktischer Analyse

Führt man die lexikalische und die syntaktische Analyse hintereinander aus, so ist dies ungünstig bzgl. der Speicherplatzeffizienz, denn man muß die Ausgabe des Scanners in einer temporären Datei, deren Größe im wesentlichen proportional zur Länge des Quelltextes ist, zwischenspeichern. Günstiger ist es, so zu programmieren, daß der Parser bei Bedarf neue Symbole anfordert und der Scanner diese erst bei Bedarf ermittelt.

Bei Compilern, die in imperativen Sprachen geschrieben werden, wird der Scanner daher meist als eine Prozedur `GetNextSymbol` realisiert, die vom Parser bei Bedarf aufgerufen wird und – wie ihr Name andeutet – das jeweils nächste Symbol ermittelt und abliefert.

Diese Verzahnung kann man in Miranda dank der lazy evaluation viel einfacher bekommen: Man programmiert den Scanner in natürlicher Weise als Funktion, die eine Quelldatei (also eine Liste von Zeichen) auf eine Liste von Symbolen abbildet. Ruft man später eine Hintereinanderausführung von Scanner und Parser auf, so sorgt die lazy evaluation des Parser-Aufrufs dafür, daß der Scanner – wiederum lazy – jeweils gerade soviele Symbole wie nötig ermittelt. Gegenüber der imperativen Lösungsvariante hat man hierbei zwei Vorteile:

- Der Scanner kann ohne Rücksicht darauf, in welcher Weise er später vom Parser verwendet wird, programmiert werden.
- Der Scanner kann auch ohne die Erstellung eines Testtreibers getestet werden.

5.2 Aufgabenstellung

In dem Verzeichnis

/home/alpha/maassen

finden Sie die Dateien `parser_tool.m` und `psa_parser.m`.

Die erstgenannte enthält einen generischen Miranda-Modul mit Werkzeugen zur Erzeugung von Teil-Parsern. Es handelt sich dabei im wesentlichen um eine Reihe von Funktionen, die vorhandene Teil-Parser entsprechend den Konstruktionen, mit denen man rechte Regelseiten in EBNF-Beschreibungen bilden kann, zu neuen Parsern zusammensetzen. Lediglich die Funktion `parse_star` ist noch nicht implementiert.

Die zweitgenannte enthält die Definition der abstrakten Syntaxbäume und einen Teil des Parsers für die Sprache PSA.

Vervollständigen Sie beide Module, sodaß ein funktionsfähiger Parser für PSA-Programme entsteht. Ebenso wie der Scanner soll auch der Parser im *panic mode* arbeiten.

In dem obengenannten Verzeichnis finden Sie in der Datei `p1.psa` ein kleines PSA-Programm, mit dem Sie ihren Parser testen können.

5.3 Vorgebene Dateien

5.3.1 Die Datei parser_tool.m

```
|| Dieser Modul stellt Parsergeneratoren fuer einige gaengige
|| EBNF-Konstrukte zur Verfuegung
|| Parameter fuer diesen Modul sind der Typ der Token und der
|| zugehoerige Scanner

%export parser_type parser parser_part parser_try parse_lit
parse_alt parse_seq parse_star modify

%free {token :: type; scanner :: [char] -> [token];}

|| Ein Parser ist ein spezieller Teilparser, der nur solche Ergebnisse
|| akzeptiert, die die Eingabe restlos verbrauchen.

parser_type * == [char] -> *
```



```
parser :: (parser_part *) -> parser_type *

parser p input =
        check ((p.scanner) input)
        where
        check (Parsed (t, [])) = t
        check (Parsed (t, anything_else)) =
                error "More input than expected"
        check Fail =
                error "Parsing terminated due to failure"

|| Ergebnis einer Teilparser-Anwendung ist entweder ein
|| erfolgreich gebildetes Objekt oder eine Fehlermeldung

parser_try * ::= Parsed * | Fail

|| Ein Teilparser bildet eine Tokenfolge auf einen Syntaxbaum
|| und eine restliche Tokenfolge ab, im Falle des Scheiterns
|| jedoch auf eine  Fehlermeldung

parser_part * == [token] -> parser_try (*, [token])

|| Fuer (parameterlose) Token gibt es einen einfachen Parsergenerator

parse_lit :: token -> (parser_part token)

(parse_lit tk) [] = Fail
(parse_lit tk) (h:t)
        = Parsed(h, t), tk = h
        = Fail, otherwise

|| Rechte Regelseiten koennen aus einer Alternative bestehen.
|| Diesem Konstrukt entspricht eine Funktion, die eine Liste von
|| Teilparsern gleichen Typs nacheinander testet und so einen neuen
|| Parser beschreibt.

parse_alt :: [parser_part *] -> parser_part *

parse_alt [] tokenlist = Fail
parse_alt (p:t) tokenlist
        = parser_try            , ok parser_try
        = parse_alt t tokenlist , otherwise
```

```
                where
                parser_try = p tokenlist
                ok (Parsed anything) = True
                ok Fail = False

|| Eine rechte Regelseite kann eine Sequenz von Symbolen sein.
|| Da man Parser unterschiedlichen Typs nicht in einer Liste
|| mischen kann, muss man sich mit einer Paarbildung behelfen.

parse_seq :: parser_part * -> parser_part ** -> parser_part (*, **)

(p1 $parse_seq p2) tokenlist =
        p2_after (p1 tokenlist)
        where
        p2_after Fail = Fail
        p2_after (Parsed (t1, rest1)) =
                (p2 $modify (pair t1)) rest1
                where
                pair x1 x2 = (x1, x2)

|| Eine spezielle Variante der Sequenz ist der Kleene'sche Stern.
|| Diese Funktion verwendet die vordefinierte Datenstruktur Liste.

parse_star :: parser_part * -> parser_part [*]

|| Mitunter ist es noetig, den Typ der geparsten Objekte zu
|| konvertieren.
|| Dies tritt zum einen auf, wenn man Uebergaenge von rechten zu
|| linken Regelseiten hat, zum anderen muss man die etwas
|| unglueckliche Tupelbildung bei parse_seq korrigieren.
|| Diese Konvertierung fuehrt die Funktion modify durch

modify :: parser_part * -> (* -> **) -> parser_part **

(p $modify f) tokenlist =
        modresult (p tokenlist)
        where
        modresult Fail = Fail
        modresult (Parsed (t, r)) = Parsed (f t, r)
```

5.3.2 Die Datei psa_parser.m

```
|| Dieser Modul enthaelt die Beschreibung der Grammatik fuer PSA
```



```
|| und den zugehoerigen Parser. Er exportiert die Token und den Typ
|| der Syntaxbaeume von PSA, ausserdem den Scanner und den
|| Parser fuer PSA.

|| Er stuetzt sich auf die Symbolklassenbeschreibung und den
|| Scanner aus dem  Modul psa_lex sowie den Modul parser_tool.

%include "psa_lex"

%include "parser_tool" {token==token; scanner=lex;}

%export token arithop relop string
        psa_parser program_type declaration_type stmt_seq_type
        stmt_type exp_type bexp_type


|| Der Parser heisst psa_parser, er nimmt als Eingabe eine
|| Zeichenfolge
psa_parser :: parser_type program_type
psa_parser = parser program_parser


|| Es folgt die Grammatik fuer PSA.
|| Zu jeder Regel der Grammatik werden die entsprechende
|| Typdeklaration fuer den abstrakten Syntaxbaum und der
|| zugehoerige Teilparser angegeben.


|| Program ::= Declaration* Statement_Sequence .
program_type ::= Program [declaration_type] stmt_seq_type
program_parser =
        ((parse_star decl_parser) $parse_seq stmt_seq_parser
         $parse_seq (parse_lit FullStop)) $modify mk_prog
        where
        mk_prog (decls, (stmts, fs)) = Program decls stmts

|| Declaration ::= var Variable Variable* ;
declaration_type == [token]
|| Dies muss natuerlich eine nicht-leere Liste von Bezeichnern sein
decl_parser = || Hier fehlt noch was

id_parser (Ident i : anything) = Parsed (Ident i, anything)
id_parser anything_else = Fail
```

```
|| Statement_Sequence ::= Statement*
stmt_seq_type == [stmt_type]
stmt_seq_parser = parse_star stmt_parser


|| Statement ::=  Variable := Expression |
||                if BExpression then Statement_Sequence
||                        else Statement_Sequence end |
||                while BExpression do Statement_Sequence end |
stmt_type ::=
        Assignment token exp_type |
        If bexp_type stmt_seq_type stmt_seq_type |
        While bexp_type stmt_seq_type

stmt_parser = parse_alt [assignment_parser, if_parser, while_parser]

assignment_parser = || Hier fehlt noch was

if_parser =
        ((parse_lit IfSym) $parse_seq bexp_parser $parse_seq
         (parse_lit ThenSym)  $parse_seq stmt_seq_parser $parse_seq
         (parse_lit ElseSym) $parse_seq stmt_seq_parser $parse_seq
         (parse_lit EndSym)) $modify mk_if_stmt
        where
        mk_if_stmt (i, (e, (t, (s1, (l, (s2, d)))))) = If e s1 s2

while_parser = || Hier fehlt noch was

|| Expression ::=
||        Variable |
||        Number |
||        (Expression ArithOp Expression)
exp_type ::=
      Variable token | Number token |
      Compound exp_type token exp_type

exp_parser =
        parse_alt [(id_parser) $modify Variable,
                (num_parser) $modify Number, compound_parser]

num_parser (Num n : anything) = Parsed (Num n, anything)
num_parser anything_else = Fail
```



```
compound_parser =
        ((parse_lit OpenPar) $parse_seq exp_parser $parse_seq
         (arithop_parser)  $parse_seq exp_parser $parse_seq
         (parse_lit ClosePar)) $modify mk_compound_exp
        where
        mk_compound_exp (k1, (e1, (ar, (e2, k2)))) = Compound e1 ar e2

arithop_parser (ArithOp op : anything) = Parsed (ArithOp op, anything)
arithop_parser anything_else = Fail

|| BExpression ::= Expression RelOp Expression
bexp_type == (exp_type, token, exp_type)

bexp_parser = || Hier fehlt noch was
```

6 Aufgabe 3: Semantische Analyse

6.1 Problembeschreibung

6.1.1 Ziele der semantischen Analyse

Programmiersprachen sind in der Regel nicht kontextfrei. Dies liegt daran, daß Eigenschaften wie etwa die Bedingung, daß verwendete Bezeichner deklariert werden müssen, nicht kontextfrei beschreibbar sind.

Trotzdem werden in der Regel kontextfreie Grammatiken oder äquivalente Formalismen (BNF, EBNF) zur Beschreibung von Programmiersprachen eingesetzt. Mit diesen Mitteln beschreibt man eine Obermenge der Programmiersprache, die noch syntaktisch eigentlich unzulässige Programme enthält. Diese werden im Anschluß an die syntaktische Analyse durch die *semantische Analyse* eliminiert.

Für diese Vorgehensweise gibt es im wesentlichen zwei Gründe:

- Kontextsensitive oder Typ-0-Grammatiken für Programmiersprachen sind zu unübersichtlich und unterstützen kein effizientes Parsen.
- Der kontextfreie Syntaxbaum eignet sich sehr gut als Grundlage für die Erzeugung des Zielcodes.

Die Aufgabe der semantischen Analyse besteht aber nicht nur in der Überprüfung von nicht kontextfrei beschreibbaren Eigenschaften einer Sprache, sondern sie soll auch einen Informationstransfer im Syntaxbaum durchführen. So muß z. B. bei der Erzeugung des Codes für die Anweisungsteile eines Programms nicht nur sichergestellt sein, daß die dort verwendeten frei gewählten Bezeichner auch deklariert sind, sondern es muß auch bekannt sein, welche Bedeutung sie haben: bei Variablen muß bekannt sein, welcher Speicherplatz ihnen zugeordnet ist, bei selbstdefinierten Prozeduren muß bekannt sein, wo im Zielprogramm der ihnen zugehörige Code liegt usw. Zu diesem Zwecke stellt man eine *Symboltabelle* auf; das ist eine Abbildung, die frei wählbaren Bezeichnern ihre Bedeutung zuordnet.

Als Mittel zur Formalisierung der semantischen Analyse haben sich *attributierte Grammatiken* bewährt; dies sind kontextfreie Grammatiken, deren Produktionen durch formale semantische Regeln ergänzt werden. Dieser Ansatz würde hier jedoch zu weit führen; im Rahmen dieses Praktikums werden wir eine ad hoc-Methode verwenden.

6.1.2 Semantische Analyse mit einer Symboltabelle

Die Struktur der Einträge für die Symboltabelle ist in unserem Fall sehr einfach, da frei gewählte Bezeichner in PSA nur ganzzahlige Variable bezeichnen können. Dementsprechend muß man zu einem Bezeichner nur eintragen, daß er deklariert wurde und welcher Speicherplatz ihm zugeordnet ist; Typinformationen u. ä. sind nicht nötig. Zur Implementierung einer Symboltabelle kann man ein Feld oder eine ähnliche Datenstruktur verwenden; in Miranda kommt natürlich auch eine Implementierung als Funktion in Frage.

Die Symboltabelle wird aus dem Deklarationsteil des Programms aufgebaut, indem man alle dort deklarierten Bezeichner einträgt und ihnen die für die Codeerzeugung nötigen Zusatzinformationen zuordnet; in unserem Fall werden den Bezeichnern lediglich Speicherplätze zugeordnet.

Mit Hilfe dieser Symboltabelle kann dann die gesamte semantische Analyse durchgeführt werden:

- Man überprüft für alle im Anweisungsteil verwendeten Bezeichner, ob sie in der Symboltabelle eingetragen sind. Ist dies der Fall, so ist das PSA-Programm syntaktisch korrekt.

- Die Symboltabelle enthält alle für die Codeerzeugung notwendigen Informationen über die benutzerdefinierten Bezeichner. Sie wird daher auch für den oben erwähnten Informationstransfer benutzt und für die Codeerzeugung aufbewahrt.

6.2 Aufgabenstellung

In dem Verzeichnis

`home/alpha/maassen`

finden Sie die Dateien `table_type` und `psa_analyser`. Die erstgenannte enthält die Spezifikation eines abstrakten Datentyps, der gleichfalls `table_type` heißt und der sich zur Implementierung einer Symboltabelle eignet. Die zweitgenannte enthält die Spezifikation einer Funktion `analyse`, die zu dem Syntaxbaum eines PSA-Programms die entsprechende Symboltabelle berechnet und die korrekte Deklaration der verwendeten Bezeichner überprüft. `analyse` soll mit einer Fehlermeldung abbrechen, falls das zu analysierende Programm syntaktisch nicht korrekt ist, d. h. falls in dem PSA-Programm, dessen abstrakter Syntaxbaum analysiert wird, eine der kontextsensitiven syntaktischen Bedingungen verletzt ist.

Vervollständigen Sie diese Programme.

6.3 Vorgegebene Dateien

6.3.1 Die Datei `table_type.m`

```
|| Dieser Modul exportiert eine Datenstruktur table_type, die sich zur
|| Darstellung von Funktionen mit endlichem Traeger eignet.

%export table_type table_entry empty_table insert_table read_table

|| Ergebnis eines lesenden Zugriffs kann entweder ein Eintrag sein
|| oder die Auskunft, dass unter diesem Index kein Wert eingetragen
|| ist.

table_entry ** ::= NoEntry | Entry **

|| table_type stellt die folgenden Operationen zur Verfuegung:
||      - Erzeugung einer leeren Tabelle
||      - Einfuegung eines Eintrages (ein eventuell vorhandener alter
||        Eintrag wird ueberschrieben)
||      - Lesender Zugriff auf eine Tabelle

abstype table_type * **
with    empty_table :: table_type * **
        insert_table :: * -> ** -> table_type * ** -> table_type * **
        read_table :: * -> table_type * ** -> table_entry **

table_type * ** == implementation * **

implementation * ** :: type
```

6.3.2 Die Datei `psa_analyser.m`

```
|| Dieser Modul stellt eine Funktion analyse zur Verfuegung, die zu
|| dem Syntaxbaum eines PSA-Programms die Symboltabelle aufbaut und
|| an Hand dieser Symboltabelle die kontextsensitiven Nebenbedingungen
|| der PSA-Syntax ueberprueft.

%export analyse table_type table_entry read_table

%include "psa_parser" -psa_parser

%include "table_type"

|| analyse liefert als explizites Ergebnis nur die Symboltabelle; die
|| syntaktische Korrektheit wird implizit ueberprueft in dem Sinne,
|| dass analyse bei einem nicht korrekt gebildeten Programm mit einer
|| Fehlermeldung aussteigt.
|| Als Speicherplaetze fuer die einzutragenden Bezeichner werden
|| fortlaufende Zahlen beginnend mit der 1 verwendet.

analyse :: program_type -> table_type token num
```

7 Aufgabe 4: Die abstrakte Maschine

7.1 Problembeschreibung

Als Zielmaschine für die Übersetzung der Programmiersprache PSA wählen wir keine reale Maschine, sondern eine sehr kleine abstrakte Maschine. Sie wird im folgenden *A-Maschine* genannt.

Die A-Maschine besteht aus drei Komponenten:

- einem Befehlszähler;
- einem Datenkeller, der als Hilfsmittel zur Auswertung von Ausdrücken dient;
- einem Hauptspeicher, in dem man Daten speichern kann.

Die A-Maschine hat weder einen Akkumulator noch spezielle Register (mit Ausnahme des Befehlszählers); Rechenoperationen werden direkt auf dem Datenkeller ausgeführt. Einträge des Datenkellers können sowohl ganze Zahlen als auch boole'sche Werte sein. (Bei realen Maschinen sind Datenkeller hardwaremäßig nicht vorhanden, werden aber im Hauptspeicher simuliert.)

Die Sprache, in der die A-Maschine programmiert wird, heißt *A-Code*. Ein Programm in dieser Sprache ist eine Liste von Maschinenbefehlen. Die Ausführung eines Maschinenprogramms beginnt mit der Ausführung des Befehls mit der Codeadresse 0 und läuft entsprechend der Wirkung der ausgeführten Befehle weiter, bis die Maschine anhält oder der Befehlszähler auf eine nicht vorhandene Codeadresse zeigt; im letzteren Fall bricht die Programmausführung mit einer Fehlermeldung ab. Bei Beginn einer Programmausführung ist der Datenkeller leer, und alle Zellen des Hauptspeichers sind in einem undefinierten Zustand.

Der Befehlssatz der A-Maschine umfaßt die folgenden Befehlstypen, deren Semantik jeweils umgangssprachlich angegeben wird:

- `Lod` n
  Der unter der Adresse n im Hauptspeicher abgelegte Wert wird auf den Datenkeller geladen. Ist die entsprechende Zelle des Hauptspeichers nicht belegt, so bricht die Programmausführung mit einer Fehlermeldung ab. Der Befehlszähler wird um 1 erhöht.

- `Sto` n
  Die auf der Spitze des Datenkellers liegende Zahl wird unter der Adresse n im Hauptspeicher abgelegt und vom Datenkeller gelöscht. Liegt keine Zahl auf dem Datenkeller, so bricht die Programmausführung mit einer Fehlermeldung ab. Der Befehlszähler wird um 1 erhöht.

- `Jmp` a
  Der Befehlszähler wird auf die Programmadresse a gesetzt.

- `Jmc` a
  Der auf dem Datenkeller liegende boole'sche Wert wird gelesen und vom Datenkeller entfernt. Ist der Wert FALSE, so wird der Befehlszähler auf a gesetzt, sonst wird er um 1 erhöht. Liegt kein boole'scher Wert auf dem Datenkeller, so bricht die Programmausführung mit einer Fehlermeldung ab.

- `Lit` z
  Die Zahl z wird auf den Datenkeller gelegt, und der Befehlszähler wird um 1 erhöht.

- `Ad`
  Die beiden oberen Einträge des Datenkellers werden durch ihre Summe ersetzt, und der Befehlszähler wird um 1 erhöht. Dabei wird angenommen, daß der zweite Operand auf der Spitze des Datenkellers liegt. Liegen keine zwei Zahlen auf dem Datenkeller, so bricht die Programmausführung mit einer Fehlermeldung ab.

  Entsprechendes gilt für die Befehle `Sb`, `Ml` und `Dv`, die die Subtraktion, die Multiplikation bzw. die ganzzahlige Division bezeichnen.

- **Eq**
  Die beiden oberen Einträge des Datenkellers werden heruntergenommen und durch einen booleschen Wert, der angibt, ob die beiden Werte gleich sind, ersetzt; der Befehlszähler wird um 1 erhöht. Es wird angenommen, daß der zweite Operand auf der Spitze des Datenkellers liegt. Liegen keine zwei Zahlen auf dem Datenkeller, so bricht die Programmausführung mit einer Fehlermeldung ab.

  Entsprechendes gilt für die Befehle `Ne`, `Gt`, `Ge`, `Lt` und `Le`, die die durch ihre Namen angedeuteten Beziehungen testen.

- **Halt**
  Die Maschine hält an.

7.2 Aufgabenstellung

Im Verzeichnis

`/home/alpha/maassen`

finden Sie die Dateien `store_type.m` und `machine.m`. Die erstgenannte exportiert einen abstrakten Datentyp gleichen Namens, der zur Implementierung des Hauptspeichers der A-Maschine geeignet ist. Die zweitgenannte enthält die Schnittstelle und einige Implementierungsvorgaben für einen Modul, der die A-Maschine implementiert. Vervollständigen Sie den Modul `machine.m`.

In der Datei `/home/alpha/maassen/p1.psx` finden Sie ein Beispielprogramm zum Testen der Maschine. Bei korrekter Arbeitsweise müssen Sie die folgende Ausgabe erhalten:

```
(1, 6)
(2, 3)
```

7.3 Vorgegebene Dateien

7.3.1 Die Datei `store_type.m`

```
|| In diesem Modul wird mit Rueckgriff auf den Modul "table_type" ein
|| Typ fuer die Darstellung des Hauptspeichers der abstrakten
|| Maschine implementiert.
|| Er unterscheidet sich von table_type dadurch, dass ein
|| Speicherabzug moeglich ist. Dies ist moeglich, weil als
|| Indizes nur natuerliche Zahlen vorkommen und der Speicher
|| sich merkt, wieviele seiner Zellen belegt sind.

%include "table_type"
```



```
%export
        table_entry store_type empty_store insert_store read_store
        display_store

abstype store_type
with    empty_store :: store_type
        insert_store :: num -> num -> store_type -> store_type
        read_store :: num -> store_type -> table_entry num
        display_store :: store_type -> [char]

store_type == (table_type num num, num)

empty_store = (empty_table, 0)

insert_store i x (tab, n)
        = (insert_table i x tab, n + 1), read_table i tab = NoEntry
        = (insert_table i x tab, n), otherwise

read_store i (table, n) = read_table i table

display_store (table, n) =
        display n table [1..]

display :: num -> table_type num num -> [num] -> [char]
display 0 table list = []
display (n + 1) table (h:t)
        = display (n + 1) table t, read_table h table = NoEntry
        = (show (h, entry)) ++ '\n' : (display n table t), otherwise
                where
                Entry entry = read_table h table
```

7.3.2 Die Datei `machine.m`

```
|| Dieser Modul realisiert eine abstrakte Stackmaschine mit
|| Hauptspeicher. Er stellt eine Funktion run zur Verfuegung,
|| die ein Maschinenprogramm aus einer Datei liest und
|| ausfuehrt, sowie den Typ der Maschinenbefehle, damit
|| man auch ausserhalb dieses Moduls Programme schreiben kann.

%export run cmd_type

|| Eine Maschine besteht aus einem Befehlszaehler, einem
```

```
|| Datenkeller und einem Hauptspeicher. Der Typ des
|| Hauptspeichers wird importiert.

%include "store_type"

machine_type == (num, [stack_entry], store_type)

|| Der Datenkeller kann als Eintraege Zahlen und boole'sche
|| Werte enthalten.

stack_entry :: type

|| Es gibt die folgenden Befehle:
|| Lesen und Beschreiben des Hauptspeichers, bedingten und
|| unbedingten Sprung,
|| arithmetische und relationale Operationen, Anhalten.

cmd_type ::=
             Lod num | Sto num | Jmp num | Jmc num |
             Lit num | Ad | Sb | Ml | Dv |
             Eq | Ne | Gt | Ge | Lt | Le | Halt

|| Ein Programmlauf fuehrt die Befehle eines Programms aus und
|| liefert als Ausgabe einen Speicherabzug der Maschine.

run :: [cmd_type] -> [char]
```

8 Aufgabe 5: Codegenerierung

8.1 Problembeschreibung

8.1.1 Codegenerierung

In den ersten drei Versuchen dieses Praktikums haben wir die sogenannte *Analysephase* eines Compilers behandelt. In dieser Phase wurde die Programmstruktur ermittelt und die Symboltabelle, die die für die Codeerzeugung notwendigen Angaben über die benutzerdefinierten Bezeichner enthält, angelegt.

An diese Phase schließt sich die *Synthesephase* an; in ihr wird das analysierte Quellprogramm in den Zielcode übersetzt. Für unsere einfache Programmiersprache läßt sich diese Phase in einem Praktikumsversuch erledigen.

Die Codegenerierung läßt sich in Miranda mit einer Reihe von Übersetzungsfunktionen induktiv über den Aufbau der abstrakten Syntaxbäume beschreiben. Dabei tritt

ein Problem auf: In den Sprungbefehlen müssen Codeadressen eingetragen werden, die von der Länge des durch andere Aufrufe von Übersetzungsfunktionen erzeugten Codes abhängen. Es gibt verschiedene Ansätze, dieses Problem zu lösen. Zwei sollen hier kurz diskutiert werden:

- Oft geht man so vor, daß man zunächst symbolische Adressen einträgt und diese später in einem zweiten Durchlauf durch echte Adressen ersetzt. Ein solches *Backpatching* läßt sich gerade in einer funktionalen Sprache sehr einfach beschreiben, da man wegen der lazy evaluation sogenannte *lazy dictionaries* implementieren kann, also "Wörterbücher", denen man auch dann schon Fragen stellen kann, wenn die entsprechenden Einträge noch nicht eingetroffen sind.
- Konzeptionell einfacher als das Backpatching ist ein Verfahren, das mit *Codelängenberechnung* arbeitet. Dabei wird zu jedem Stück erzeugtem Code auch seine Länge mit ausgegeben; das genügt, um die korrekten Adressen eintragen zu können.

Der Einfachheit halber werden wir hier mit der Codelängenberechnung arbeiten: jede Übersetzungsfunktion liefert als Ergebnis nicht nur ein Stück Maschinencode, sondern auch die Länge[5] dieses Codes.

8.1.2 Erzeugung ausführbarer Programme

Da wir keine reale Zielmaschine haben, können wir auch nicht ohne weiteres ausführbare Programme erzeugen, sondern müssen einen Umweg nehmen. Wir gehen folgendermaßen vor: Die Übersetzung eines PSA-Programms erzeugt zwei Dateien: eine enthält das Maschinenprogramm für die A-Maschine (pro Zeile ein Befehl), die zweite enthält einen "magic string", dessen Wirkung darin besteht, das Maschinenprogramm zu laden und von der A-Maschine ausführen zu lassen. Wir vereinbaren die folgenden Namenskonventionen:

- Der Name einer Quelldatei, die ein PSA-Programm enthält, muß die Endung `.psa` haben.
- Der PSA-Compiler akzeptiert bei seinem Aufruf eine Liste mit Namen von PSA-Quelldateien, wobei die Endung `.psa` weggelassen werden kann. Der Compiler legt den A-Code zu einem PSA-Programm in einer Datei ab, deren Name sich vom Namen der Quelldatei dadurch unterscheidet, daß die Endung `.psa` durch `.psx` ersetzt ist. Für jede Quelldatei, die aus irgendwelchen Gründen nicht gelesen werden kann bzw. für die kein Schreibrecht für die zugehörige `.psx`-Datei besteht, wird eine entsprechende Fehlermeldung auf die Standardfehlerausgabe geschrieben.

[5] Diese Längenberechnung ist für den Code der A-Maschine trivial, da jeder Befehl die Länge 1 hat. Trotzdem sollte man schon aus Effizienzgründen nicht den Miranda-Operator `#` verwenden.

- Der "magic string", der das A-Code-Programm auf der A-Maschine ausführen läßt, wird unter dem um die Endung `.psa` verkürzten Namen der Quelldatei abgelegt und ausführbar gemacht.

8.2 Semantik von PSA

Bisher haben wir nur die Syntax von PSA definiert, die Semantik aber noch nicht. Da die Syntax von PSA, insbesondere die verwendeten Schlüsselwörter und Operationszeichen, hinreichend suggestiv ist, soll hier auf eine formale Definition der Semantik von PSA verzichtet werden; es gelten – bis auf die nachfolgend aufgeführten Unterschiede – die Konventionen für die entsprechenden Sprachkonstrukte aus Modula-2.

- Der lesende Zugriff auf eine Variable, die nicht durch das PSA-Programm initialisiert wurde, löst einen Laufzeitfehler aus.
- Der Operator `/` bezeichnet die ganzzahlige Division, die in Modula-2 mit dem Schlüsselwort `DIV` bezeichnet wird.

8.3 Anleitung zur Codegenerierung

Die folgende Übersicht beschreibt informell Regeln für die Generierung des A-Codes zu einem PSA-Programm.

- Ein PSA-Programm wird übersetzt, indem man seinen Anweisungsteil mit Hilfe der aus seinem Deklarationsteil erzeugten Symboltabelle[6] übersetzt und den Befehl `Halt` anhängt.
- Eine Folge von Anweisungen wird übersetzt, indem man den Code für die einzelnen Anweisungen hintereinanderhängt.
- Eine Wertzuweisung wird übersetzt, indem man den Ausdruck auf der rechten Seite übersetzt und an diesen Code den Befehl `Sto` n anhängt, wobei n der Speicherplatz ist, der durch die Variable, der der Wert des Ausdrucks zuzuweisen ist, bezeichnet wird.
- Eine While-Schleife wird übersetzt, indem man die folgenden Programmstücke hintereinanderhängt:
  1. Code für den boole'schen Ausdruck
  2. `Jmc` "hinter das Ende dieser While-Schleife"
  3. Code für den Schleifenkörper

[6] Alle weiteren Übersetzungsfunktionen benutzen diese Symboltabelle. Dies wird im folgenden nicht mehr ausdrücklich erwähnt.

  4. `Jmp` "an den Anfang dieser While-Schleife"
- Eine If-Anweisung wird übersetzt, indem man die folgenden Programmstücke hintereinanderhängt:
  1. Code für den boole'schen Ausdruck
  2. `Jmc` "an den Anfang des ELSE-Falles"
  3. Code für den THEN-Fall
  4. `Jmp` "an das Ende dieser If-Anweisung"
  5. Code für den ELSE-Fall
- Ein zusammengesetzter Ausdruck (boole'sch oder arithmetisch) wird übersetzt, indem man die Codes für die beiden Operanden hintereinanderhängt und daran den Befehl anhängt, der dem Operator entspricht.
- Der Code für eine Konstante ist der entsprechende `Lit`-Befehl.
- Der Code für eine Variable ist der `Lod`-Befehl, der den Inhalt der Speicherzelle, die durch die Variable bezeichnet wird, auf den Datenkeller holt.

Für die Übersetzung von Ausdrücken können Sie sich auch die folgende Faustregel merken: Nach Ausführung des Codes zu einem Ausdruck liegt der Wert des Ausdrucks auf dem Datenkeller.

8.4 Aufgabenstellung

1. In dem Verzeichnis

   `/home/alpha/maassen`

   finden Sie eine Datei `psa_compiler.m`, die die Spezifikationen einer Funktion `compile` und einer Reihe von Übersetzungsfunktionen enthält.

   (a) Implementieren Sie `trans` und die zugehörigen anderen Übersetzungsfunktionen.

   (b) Implementieren Sie die Funktion `compile`. Gehen Sie dabei davon aus, daß sowohl die PSA-Quelldateien als auch die Dateien des PSA-Systems in dem gleichen Verzeichnis liegen und daß der Compiler nur von diesem Verzeichnis aus aufgerufen wird. Fangen Sie das Fehlen von Lese- bzw. Schreibrechten durch geeignete *eigene* Fehlermeldungen ab.

2. Richten Sie unter Verwendung eines "magic string" eine ausführbare Datei `psa` ein, die als Parameter eine (bis auf systembedingte Einschränkungen) beliebig lange Liste mit Namen von PSA-Programmen akzeptiert und diese in der für `compile` spezifizierten Weise compiliert. (Dieses Programm `psa` wird als Compiler aufgerufen.)

8.5 Vorgegebene Dateien

8.5.1 Die Datei psa_compiler.m

```
|| Dieser Modul enthaelt den eigentlichen Uebersetzer:
|| die Funktion compile.

%include "machine"

%include "psa_parser"

%include "psa_analyser"

%export cmd_type compile

compile :: [char] -> [sys_message]

|| compile akzeptiert einen Dateinamen mit und ohne die Endung ".psa";
|| fehlt die Endung ".psa", so wird sie automatisch ergaenzt.
|| compile legt das Programm fuer die A-Maschine in einer Datei
|| gleichen Namens, aber mit der Endung ".psx" ab; ausserdem wird
|| ein "magic string", der das gespeicherte A-Code-Programm
|| auf der A-Maschine ausfuehrt, unter dem
|| um die Endung ".psa" verkuerzten Dateinamen abgelegt.
||
|| Besteht fuer die angegebene Datei kein Leserecht bzw. besteht fuer
|| das Zielverzeichnis kein Schreibrecht, so wird eine entsprechende
|| Fehlermeldung ausgegeben.




|| Es folgen die Spezifikationen der Uebersetzungsfunktionen
||
|| Fuer jeden Typ von Sprachkonstrukten (Anweisungsfolgen,
|| Anweisungen, arithmetische und boole'sche Ausdruecke) gibt es
|| eine eigene Uebersetzungsfunktion. Sie alle benoetigen die
|| Symboltabelle aus der semantischen Analyse.
|| Die Uebersetzungsfunktionen, die Sprungbefehle erzeugen, benoetigen
|| noch einen weiteren Parameter: die Anfangsadresse des von ihnen zu
|| erzeugenden Codes.
||
|| Hauptfunktion der Uebersetzung ist die Funktion trans, die als
|| Eingabe ein PSA-Programm in Form einer ASCII-Datei nimmt, die
```



```
|| Analysephase des Compilers durchfuehrt und anschliessend die
|| Uebersetzungsfunktionen aufruft.

symbol_table_type == table_type token num

|| trans nimmt als Eingabe ein PSA-Programm in Form einer ASCII-Datei
|| und berechnet das zugeh"orige A-Code-Programm. Dieses
|| A-Code-Programm endet mit einem Halt-Befehl.

trans :: [char] -> [cmd_type]

stmts_trans ::
        stmt_seq_type -> symbol_table_type -> num -> ([cmd_type], num)

stmt_trans ::
        stmt_type -> symbol_table_type -> num -> ([cmd_type], num)

exp_trans ::
        exp_type -> symbol_table_type -> ([cmd_type], num)

bexp_trans ::
        bexp_type -> symbol_table_type -> ([cmd_type], num)
```

9 Aufgabe 6: Analyse einer erweiterten Sprache

In den ersten fünf Versuchen dieses Praktikums haben wir eine abstrakte Maschine implementiert und einen Compiler für die Übersetzung einer sehr einfachen Programmiersprache in den Code dieser Maschine geschrieben. Diese Programmiersprache soll nun um einen einfachen Mechanismus zur Behandlung von Ein-/Ausgabe erweitert werden. Dies dient zwei Zielen:

- Die Anpaßbarkeit und Erweiterbarkeit funktionaler Software soll demonstriert werden.
- Die Entwicklung interaktiver Programme mit Miranda soll geübt werden.

9.1 Problembeschreibung

9.1.1 Erweiterung von PSA zu PSA+

Die Programmiersprache PSA wird durch die folgenden Ergänzungen zur Sprache PSA+ erweitert:

Kontextfreie Syntax Es werden als zusätzliche Regeln eingeführt:

Statement ::= **read Variable**
write Expression
write String

Mikrosyntax

- Eine Zeichenkette (**String**) ist eine beliebige, in Hochkommata eingeschlossene Folge von Zeichen über dem Alphabet, das aus Groß- und Kleinbuchstaben, Ziffern, dem Leerzeichen sowie den Zeichen

  ! ( ) ? - . : ,

  besteht.

Semantik

- Eine **write**-Anweisung bewirkt, daß das angegebene Objekt auf die Standardausgabe ausgegeben wird; anschließend wird ein Zeilenumbruch ausgeführt. Zeichenketten werden ohne die sie umschließenden Hochkommata ausgegeben.
- Eine **read**-Anweisung liest einen neuen Wert für die als Parameter angegebene Variable von der Standardeingabe ein. Die Mikrosyntax der Zahlen ist dabei wie in PSA. Leerzeichen, Tabulatorsprünge und newline-Zeichen haben die Funktion von Trennzeichen und sind ansonsten ohne Bedeutung. Alle anderen Zeichen in der Eingabe sind ohne Bedeutung und werden überlesen, sie wirken also auch nicht als Trennzeichen. **read**-Anweisungen überlesen insbesondere auch führende Trennzeichen.

9.1.2 Verbesserung des Parsers

Die Funktion `parse_alt`, die wir bisher zur Behandlung von Alternativen auf rechten Regelseiten von Grammatiken verwendet haben, ist vergleichsweise ineffizient, da sie die gegebenen Alternativen einfach nacheinander ausprobiert ohne Rücksicht darauf, ob sich nicht schon durch einen einfachen Test wie etwa die Überprüfung des nächsten Tokens entscheiden läßt, welcher der möglichen Fälle vorliegt. Beispiel:

- Ist bei einem Aufruf von `stmt_parser` das nächste Token eine Variable, so kann nur eine Wertzuweisung vorliegen.
- Ist bei einem Aufruf von `stmt_parser` das nächste Token ein `If`, so kann nur eine If-Anweisung vorliegen.
- Ist bei einem Aufruf von `stmt_parser` das nächste Token ein `While`, so kann nur eine While-Anweisung vorliegen.

In vielen Fällen kann man durch Berücksichtigung solch einfacher Regeln die Laufzeit eines Parsers reduzieren.

9.2 Aufgabenstellung

Programmieren Sie – ausgehend von den entsprechenden Modulen für PSA – den Analyseteil eines Compilers für PSA+, also die lexikalische, die syntaktische und die semantische Analyse. Eliminieren Sie dabei die `parse_alt`-Konstruktionen aus dem Parser.

10 Aufgabe 7: Synthesephase einer Spracherweiterung

10.1 Problembeschreibung

10.1.1 Erweiterung der abstrakten Maschine

Die A-Maschine, die wir als Zielmaschine für den PSA-Compiler hatten, hat weder Ein- noch Ausgabe. Dementsprechend ist sie als Zielmaschine für PSA+ nicht geeignet. Wir modifizieren Sie zu einer neuen Maschine mit Namen A+.

A+ hat außer den Komponenten der A-Maschine (Befehlszähler, Datenkeller und Hauptspeicher) noch ein Eingabe- und ein Ausgabemedium, die sequentiell gelesen bzw. beschrieben werden können. Sowohl die Ein- als auch die Ausgabe besteht aus einer Liste von Buchstaben (Zeichenkette).

Der Befehlssatz der A+ umfaßt neben den Befehlen der A-Maschine noch drei weitere:

- `Put`
  Die Zahl auf der Spitze des Datenkellers wird mit einem nachfolgenden newline-Zeichen auf die Ausgabe geschrieben und vom Datenkeller entfernt, und der Befehlszähler wird um 1 erhöht. Liegt keine Zahl auf dem Datenkeller, so bricht die Programmausführung mit einer Fehlermeldung ab.
- `Get`
  Die nächste Zahl wird von der Eingabe gestrichen und auf den Datenkeller gelegt; der Befehlszähler wird um 1 erhöht. Ist die Eingabe leer, so bricht die Programmausführung mit einer Fehlermeldung ab. Der Einfachheit halber gelte für die Syntax der Eingabe (insbesondere auch für die Konventionen über Trennzeichen) das gleiche wie für PSA+.
- `Text` *string*
  Die Zeichenkette *string* wird mit einem nachfolgenden newline-Zeichen auf die Ausgabe geschrieben, und der Befehlszähler wird um 1 erhöht.

Die Befehle, die es auch in A-Code gibt, verändern die Ein-/Ausgabeteile von A+ nicht und wirken ansonsten analog wie auf der A-Maschine.

10.1.2 Verwendung von Pfadnamen

Wir haben eine sehr einfache Konvention über die möglichen Dateinamen für Aufrufe des PSA-Compilers. Der PSA+ -Compiler soll in mehrfacher Hinsicht komfortabler werden:

- PSA+ -Quelldateien dürfen in beliebigen Verzeichnissen liegen.
- Beim Aufruf des Compilers dürfen Dateinamen mit einer Pfadangabe beginnen. Zulässig sollen sein:
  - Dateinamen ohne vorangestellten Pfad
  - Dateinamen mit einer absoluten Pfadangabe (beginnend mit '/')
  - Dateinamen mit einer relativen Pfadangabe (beginnend mit einem alphanumerischen Verzeichnisnamen oder mit '..'
- Der Compiler kann von überallher aufgerufen werden. Er legt die von ihm erzeugten Dateien in das Verzeichnis, in dem auch die Quelldatei liegt.
- Die compilierten Dateien sollen verschiebbar sein, wobei jedoch die zwei zu einer PSA+ -Quelldatei gehörenden übersetzten Dateien jeweils im selben Verzeichnis liegen müssen.

Um dies zu realisieren, muß das PSA-System insbesondere wissen, wo es selber im Verzeichnisbaum liegt, denn in den vom Compiler erzeugten Dateien muß ja die abstrakte Maschine zur Verfügung stehen. Zu diesem Zweck kann man eine UNIX-Umgebungsvariable einrichten, die den Pfad zum PSA-System enthält. (Dies kann man z. B. unter Verwendung des Befehls `setenv` in der Datei `.cshrc` tun.)

10.2 Aufgabenstellung

Implementieren Sie – ausgehend von der A-Maschine – die A+ und entwickeln Sie aus dem PSA-Compiler einen Compiler für PSA+.

Beachten Sie dabei folgendes:

- Mangels einer expliziten Ausgabe hatten wir als Ausgabe eines A-Code-Programms einen Speicherabzug der A-Maschine am Ende des Programms vereinbart. Bei der A+ ist eine solche Konvention nicht sinnvoll; hier soll die Ausgabe aus dem bestehen, was explizit auf das Ausgabemedium gegeben wird.

- Achten Sie bei der Implementierung der A+ darauf, daß Ein- und Ausgabe auf eine sinnvolle Art zeitlich verzahnt werden. (Zur Synchronisation können Sie die Miranda-Funktion `seq` benutzen.)
- Beachten Sie die oben beschriebenen Konventionen über die Verwendbarkeit des Compilers im Dateisystem von UNIX.

11 Zusammenfassung der Ergebnisse

Der einwöchige Programmierkurs verlief trotz der Kürze der zur Verfügung stehenden Zeit erfolgreich. Hierbei dürften zwei Gesichtspunkte eine große Rolle gespielt haben:

- Aus der Vorlesung über Programmierung und der Sprache Modula-2 kannten die Praktikumsteilnehmer bereits wichtige Grundbegriffe aus der Theorie der Programmiersprachen, insbesondere auch die Begriffe
  - Lokalität von Deklarationen
  - Funktionsabstraktion
  - Abstrakte Datentypen
  - Modularisierung
  - Ex- und Import von Objekten

  Dadurch wurde die Erfassung und Einordnung der entsprechenden Sprachkonstrukte in Miranda sehr erleichtert. Der Kurs wäre gewiß nicht in so kurzer Zeit durchführbar gewesen, wenn man all diese Begriffe auch von der inhaltlichen Seite hätte einführen müssen.
- Die enge Verzahnung von theoretischem Unterricht und praktischer Arbeit am Rechner ermöglichte eine sehr schnelle Rekapitulierung und Vertiefung des erlernten Stoffes. Diese Arbeitsform war aber nur deshalb realisierbar, weil die Zahl der Praktikumsteilnehmer so klein war, daß es möglich war, jedem eine Woche lang praktisch ganztägig einen eigenen Arbeitsplatz am Rechner zur Verfügung zu stellen.

Die meisten Praktikumsteilnehmer fanden einen recht guten Zugang zu den ihnen bis dahin noch unbekannten Konzepten der funktionalen Programmierung; zu nennen sind hier insbesondere die Arbeit mit unendlichen Datenobjekten in Verbindung mit lazy evaluation sowie ZF-Ausdrücke. Schwierigkeiten gab es jedoch bei den meisten mit Funktionen höheren Typs, sobald die Beispiele nicht-trivial wurden, also z. B. bei der Besprechung des zum Miranda-System gehörenden Moduls `interaction.m`, der Funktionen höheren Typs zur Erzeugung interaktiver Programme zur Verfügung stellt.

Das Praktikum wurde nach seiner Beendigung von den Teilnehmern in Form einer anonymen Befragung beurteilt. Das Echo war einhellig positiv; insbesondere wurden die Aufgaben von allen Teilnehmern als gut lösbar und von fast allen als vom Umfang her angemessen eingestuft. Alle Teilnehmer beendeten das Praktikum erfolgreich.

Das vielleicht wichtigste Ergebnis des Praktikums ist wohl, daß Studenten auch ohne Vorkenntnisse über die mathematischen Grundlagen funktionaler Programmiersprachen eine solche Sprache in vergleichsweise kurzer Zeit verstehen und ihre Anwendung erlernen können, und zwar in ihrer ganzen Tiefe, einschließlich der anspruchsvollen Konstrukte wie etwa "unendliche Datenobjekte" und "Funktionen höheren Typs".

A Lösungen der Aufgaben

A.1 Aufgabe 1

A.1.1 Die Datei psa_lex.m

```
|| Definition der Symbolklassen und des Scanners fuer die
|| Modellsprache PSA

%export string arithop relop token lex

string == [char]

arithop ::= Plus | Minus | Times | Div

relop ::= EQ | NE | LT | LE | GT | GE

token ::=
       ArithOp arithop | AssignOp | ClosePar | DoSym | ElseSym |
       EndSym | FullStop | Ident string | IfSym | Num num |
       OpenPar | RelOp relop | Semicolon | ThenSym | VarSym | WhileSym

lex :: [char] -> [token]

lex (h : t)
     = check_Id (h : takewhile letter_or_digit t) :
          lex (dropwhile letter_or_digit t), letter h
          || Schluesselwoerter und Bezeichner werden zunaechst alle wie
          || Bezeichner eingelesen, anschliessend durch die Funktion
          || check_Id ueberprueft und in die entsprechenden Symbole
          || umgewandelt.
     = Num (numval (h : takewhile digit t)) :
```



```
          lex (dropwhile digit t), digit h
     = convert_Sym (h : t), otherwise
          where
          letter_or_digit c =
               letter c \/ digit c
lex [] = []

check_Id str
     = IfSym      , str = "if"
     = ThenSym    , str = "then"
     = ElseSym    , str = "else"
     = EndSym     , str = "end"
     = WhileSym   , str = "while"
     = DoSym      , str = "do"
     = VarSym     , str = "var"
     = Ident str  , otherwise

convert_Sym (h : t)
     = Semicolon : rest         , h = ';'
     = ArithOp Plus : rest      , h = '+'
     = ArithOp Minus : rest     , h = '-'
     = ArithOp Times : rest     , h = '*'
     = ArithOp Div : rest       , h = '/'& (t = [] \/ hd t ~= '=')
     = RelOp EQ : rest          , h = '='
     = RelOp NE : lex (tl t)    , h ='/' & t ~= [] & hd t = '='
     = RelOp LT : rest          , h ='<' & (t = [] \/ hd t ~= '=')
     = RelOp LE : lex (tl t)    , h ='<' & t ~= [] & hd t = '='
     = RelOp GT : rest          , h ='>' & (t = [] \/ hd t ~= '=')
     = RelOp GE : lex (tl t)    , h ='>' & t ~= [] & hd t = '='
     = OpenPar : rest           , h = '('
     = ClosePar : rest          , h = ')'
     = FullStop : rest          , h = '.'
     = AssignOp : lex (tl t)    , h = ':' & t ~= [] & hd t = '='
     = rest                     , member "\n\t " h
     = error ("Illegal character: "++ [h])
                                , otherwise
          where
          rest = lex t
```

A.2 Aufgabe 2

A.2.1 Die Datei parser_tool.m

Die vorgegebene Datei ist um eine Funktion zu ergänzen:

```
parse_star :: parser_part * -> parser_part [*]
parse_star p [] =
     Parsed ([], [])
parse_star p tokenlist =
     continue p tokenlist
     where
     continue Fail =
          Parsed ([], tokenlist)
          || Laesst sich mit p kein weiteres Objekt parsen, so wird
          || eine leere Liste zurueckgegeben.
     continue (Parsed (t, rest)) =
          ((parse_star p) $modify (t :)) rest
          || Laesst sich noch ein weiteres Objekt parsen, so wird
          || parse_star p noch einmal aufgesetzt; das bereits erkannte
          || Objekt wird anschliessend der erzeugten Liste
          || hinzugefuegt.
```

A.2.2 Die Datei psa_parser.m

Die vorgegebene Datei ist um die folgenden Funktionen zu ergänzen:

```
decl_parser =
     ((parse_lit VarSym) $parse_seq (id_parser) $parse_seq
          (parse_star (id_parser)) $parse_seq
          (parse_lit Semicolon)) $modify mk_decl
     where
     mk_decl (v, (id, (ids, sem))) = id : ids

assignment_parser =
     ((id_parser) $parse_seq (parse_lit AssignOp) $parse_seq
          exp_parser) $modify mk_assign_stmt
     where
     mk_assign_stmt (v, (as, e)) = Assignment v e

while_parser =
     ((parse_lit WhileSym) $parse_seq bexp_parser $parse_seq
          (parse_lit DoSym) $parse_seq stmt_seq_parser $parse_seq
```



```
          (parse_lit EndSym)) $modify mk_while_stmt
     where
     mk_while_stmt (wh, (e, (do, (s, en)))) = While e s

bexp_parser =
     (exp_parser $parse_seq (relop_parser) $parse_seq exp_parser)
          $modify mk_bexp
     where
     mk_bexp (e1, (rel, e2)) = (e1, rel, e2)

relop_parser (RelOp op : anything) =
     Parsed (RelOp op, anything)
relop_parser anything_else =
     Fail
```

A.3 Aufgabe 3

A.3.1 Die Datei table_type.m

Die vorgegebene Datei wird wie folgt ergänzt:

```
implementation * ** == * -> table_entry **
|| Die Tabellen werden als Funktionen implementiert.

empty_table = const NoEntry
|| Eine leere Tabelle liefert stets die Antwort "Kein Eintrag".

insert_table name obj tab =
     tab1
     where
     tab1 m
          = Entry obj, name = m
          = tab m    , otherwise

read_table n tab =
     tab n
```

A.3.2 Die Datei psa_analyser.m

```
|| Dieser Modul stellt eine Funktion analyse zur Verfuegung, die zu
|| dem Syntaxbaum eines PSA-Programms die Symboltabelle aufbaut und
|| an Hand dieser Symboltabelle die kontextsensitiven Nebenbedingungen
|| der PSA_Syntax ueberprueft.
```

```
%export analyse table_type table_entry read_table

%include "psa_parser"

%include "table_type"

|| analyse liefert als explizites Ergebnis nur die Symboltabelle; die
|| syntaktische Korrektheit wird implizit ueberprueft in dem Sinne,
|| dass analyse bei einem nicht korrekt gebildeten Programm mit einer
|| Fehlermeldung aussteigt.
|| Als Speicherplaetze fuer die einzutragenden Bezeichner werden
|| fortlaufende Zahlen beginnend mit der 1 verwendet.

analyse :: program_type -> table_type token num

analyse (Program decls stmts)
     = tab                             , stmts_check tab stmts
          || Falls der Anweisungsteil bzgl der Symboltabelle korrekt
          || ist, wird diese als Ergebnis zurueckgegeben.
          || Die Korrektheit des Deklarationsteils selber wird beim
          || Aufbau der Symboltabelle ueberprueft.
     = error "variable not declared" , otherwise
          where
          tab = make_table decls 1

make_table [] n =
     empty_table
make_table ([]:v) n =
     make_table v n
make_table ((t:r):v) n
     = error "variable declared twice", declared tab1 t
     = insert_table t n tab1, otherwise
          where
          tab1 = make_table (r:v) (n+1)

stmts_check tab [] =
     True
stmts_check tab (s:t) =
     stmt_check tab s & stmts_check tab t

stmt_check tab (Assignment t e) =
     declared tab t & exp_check tab e
```



```
stmt_check tab (If b s1 s2) =
     bexp_check tab b & stmts_check tab s1 & stmts_check tab s2
stmt_check tab (While b s) =
     bexp_check tab b & stmts_check tab s

exp_check tab (Variable t) =
     declared tab t
exp_check tab (Number t) =
     True
exp_check tab (Compound e1 t e2) =
     exp_check tab e1 & exp_check tab e2

bexp_check tab (e1,t,e2) =
     exp_check tab e1 & exp_check tab e2

declared tab t =
     read_table t tab ~= NoEntry
```

A.4 Aufgabe 4

A.4.1 Die Datei `machine.m`

```
|| Dieser Modul realisiert eine abstrakte Stackmaschine mit
|| Hauptspeicher. Er stellt eine Funktion run zur Verfuegung, die ein
|| Maschinenprogramm aus einer Datei liest und ausfuehrt, sowie den
|| Typ der Maschinenbefehle, damit man auch ausserhalb dieses Moduls
|| Programme schreiben kann.

%export run cmd_type

|| Eine Maschine besteht aus einem Befehlszaehler, einem Datenkeller
|| und einem Hauptspeicher. Der Typ des Hauptspeichers wird importiert.

%include "store_type"

machine_type == (num, [stack_entry], store_type)

|| Der Datenkeller kann als Eintraege Zahlen und boole'sche Werte
|| enthalten

stack_entry ::= Bool bool | Num num

|| Es gibt die folgenden Befehle:
```

```
|| Lesen und Beschreiben des Hauptspeichers, bedingten und unbedingten
|| Sprung, arithmetische und relationale Operationen, Anhalten.

cmd_type ::=
          Lod num | Sto num | Jmp num | Jmc num |
          Lit num | Ad | Sb | Ml | Dv |
          Eq | Ne | Gt | Ge | Lt | Le | Halt

|| Ein Programmlauf fuehrt die Befehle eines Programms aus und liefert
|| als Ausgabe einen Speicherabzug der Maschine.

run :: [cmd_type] -> [char]

run cmds =
     execute cmds (0,[],empty_store)
     where
     execute cmds (ip, stack, store)
          = display_store store, next_cmd = Halt
               || Ist das naechste Kommando ein Haltebefehl, so bricht
               || die Ausfuehrung ab, und ein Speicherabzug wird
               || ausgegeben.
          = execute cmds (exec next_cmd (ip, stack, store)), otherwise
               || Liegt kein Haltebefehl vor, so wird der Befehl
               || ausgefuehrt, und die Maschine fuehrt ihr Programm
               || bzgl des neuen Zustandes weiter aus.
               where
               next_cmd = cmds!ip, ip <= #cmds - 1
                     = error "undefined code address", otherwise

exec :: cmd_type -> machine_type -> machine_type

exec (Lod a) (ip, stack, store)
     = error "reading uninitialized storage cell", rs = NoEntry
     = (ip + 1, (Num (get rs)) : stack, store), otherwise
          where
          rs = read_store a store
          get (Entry n) = n

exec (Sto a) (ip, Num n : stack, store) =
     (ip + 1, stack, insert_store a n store)
exec (Sto a) anything_else = error "attempt to store an inproper value"

exec (Jmp a) (ip, stack, store) = (a, stack, store)
```



```
exec (Jmc a) (ip, Bool b : stack, store)
     = (a, stack, store), b = False
     = (ip + 1, stack, store), otherwise
exec (Jmc a) anything_else = error "boolean value expected"

exec (Lit n) (ip, stack, store) = (ip + 1, (Num n) : stack, store)

exec (Ad) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Num (a1 + a2) : stack, store)
exec (Ad) anything_else = error "missing or wrong operands for Ad"

exec (Sb) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Num (a1 - a2) : stack, store)
exec (Sb) anything_else = error "missing or wrong operands for Sb"

exec (Ml) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Num (a1 * a2) : stack, store)
exec (Ml) anything_else = error "missing or wrong operands for Ml"

exec (Dv) (ip, Num a2 : Num a1 : stack, store)
     = (ip + 1, Num (a1 div a2) : stack, store), a2 ~= 0
     = error "division by zero", otherwise
exec (Dv) anything_else = error "missing or wrong operands for Dv"

exec (Eq) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Bool (a1 = a2) : stack, store)
exec (Eq) anything_else = error "missing or wrong operands for Eq"

exec (Ne) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Bool (a1 ~= a2) : stack, store)
exec (Ne) anything_else = error "missing or wrong operands for Ne"

exec (Gt) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Bool (a1 > a2) : stack, store)
exec (Gt) anything_else = error "missing or wrong operands for Gt"

exec (Ge) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Bool (a1 >= a2) : stack, store)
exec (Ge) anything_else = error "missing or wrong operands for Ge"

exec (Lt) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Bool (a1 < a2) : stack, store)
```

```
exec (Lt) anything_else = error "missing or wrong operands for Lt"

exec (Le) (ip, Num a2 : Num a1 : stack, store) =
     (ip + 1, Bool (a1 <= a2) : stack, store)
exec (Le) anything_else = error "missing or wrong operands for Le"
```

A.5 Aufgabe 5

A.5.1 Die Datei psa_compiler.m

```
|| Dieser Modul enthaelt den eigentlichen Uebersetzer: die Funktion
|| compile.

%include "machine"

%include "psa_parser"

%include "psa_analyser"

%export cmd_type compile

compile :: [char] -> [sys_message]

|| compile akzeptiert einen Dateinamen mit und ohne die Endung ".psa";
|| fehlt die Endung ".psa", so wird sie automatisch ergaenzt. compile
|| legt das Programm fuer die A-Maschine in einer Datei gleichen
|| Namens, aber mit der Endung ".psx" ab; ausserdem wird ein "magic
|| string", der das gespeicherte A-Code-Programm auf der A-Maschine
|| ausfuehrt, unter dem um die Endung ".psa" verkuerzten Dateinamen
|| abgelegt.
||
|| Besteht fuer die angegebene Datei kein Leserecht bzw. besteht fuer
|| das Zielverzeichnis kein Schreibrecht, so wird eine entsprechende
|| Fehlermeldung ausgegeben.

compile filename =
     gen_file file
     || genfile berechnet zu dem ggfs. um die Endung '.psa'
     || verkuerzten Dateinamen die Liste der "system messages",
     || durch deren Auswertung (bzw. Ausfuehrung) die vom Compiler
     || zu erzeugenden Dateien beschrieben werden.
     where
     file
```



```
          = filename, take 4 (reverse filename) ~= "asp."
          = reverse (drop 4 (reverse filename)), otherwise

gen_file file
     = [Stderr ("Can't write to file: " ++ file ++ ".psx\n")]
          , ~w_permission
     = [Stderr ("Can't open file: " ++ file ++ ".psa\n")]
          , ~r_permission
     = generate_files, otherwise
          where
          generate_files =
               || Die Datei 'file.psa' wird gelesen und uebersetzt;
               || der erzeugte A-Code wird in die Datei 'file.psx'
               || geschrieben.
               [Tofile (file ++ ".psx")
                    (lay (map show (trans (read (file++".psa"))))),
               || Unter dem Namen 'file' wird eine Datei angelegt,
               || deren Ausfuehrung die Auswertung eines Laufes der
               || A-Maschine mit dem Programm 'file.psx' bewirkt.
               Tofile file (call_machine file),
               || Zu diesem Zwecke wird die Datei 'file' ausfuehrbar
               || gemacht.
               System ("chmod 755 " ++ file)]
          r_permission = member (filemode (file ++ ".psa")) 'r'
          w_permission = member (filemode ".") 'w'

call_machine file =
     "#! /global/bin/mira -exp\n"++
     "run (readvals \"" ++ file ++ ".psx\")\n" ++
     "\n" ++
     "%include \"machine\""

|| Es folgen die Uebersetzungsfunktionen
||
|| Fuer jeden Typ von Sprachkonstrukten (Anweisungsfolgen, Anweisungen,
|| arithmetische und boole'sche Ausdruecke) gibt es eine eigene
|| Uebersetzungsfunktion. Sie alle benoetigen die Symboltabelle aus
|| der semantischen Analyse.
|| Die Uebersetzungsfunktionen, die Sprungbefehle erzeugen, benoetigen
|| noch einen weiteren Parameter: die Anfangsadresse des von ihnen zu
|| erzeugenden Codes.
||
|| Hauptfunktion der Uebersetzung ist die Funktion trans, die als
```

```
|| Eingabe ein PSA-Programm in Form einer ASCII-Datei nimmt, die
|| Analysephase des Compilers durchfuehrt und anschliessend die
|| Uebersetzungsfunktionen aufruft.

symbol_table_type == table_type token num

|| trans nimmt als Eingabe ein PSA-Programm in Form einer ASCII-Datei
|| und berechnet das zugeh"orige A-Code-Programm. Dieses
|| A-Code-Programm endet mit einem Halt-Befehl.

trans :: [char] -> [cmd_type]

trans input =
     stmts_code ++ [Halt]
     where
     Program decls stmts = psa_parser input
     st = analyse (Program decls stmts)
     (stmts_code, stmts_len) = stmts_trans stmts st 0

stmts_trans ::
     stmt_seq_type -> symbol_table_type -> num -> ([cmd_type], num)

stmts_trans [] st n =
     ([], 0)
stmts_trans (stmt:stmts) st n =
     (stmt_code ++ stmts_code, stmt_len + stmts_len)
     where
     (stmt_code, stmt_len) = stmt_trans stmt st n
     (stmts_code, stmts_len) = stmts_trans stmts st (n + stmt_len)

stmt_trans ::
     stmt_type -> symbol_table_type -> num -> ([cmd_type], num)

stmt_trans (Assignment id exp) st n =
     (exp_code ++ [Sto num], exp_len + 1)
     where
     (exp_code, exp_len) = exp_trans exp st
     Entry num = read_table id st

stmt_trans (While bexp stmts) st n =
     (bexp_code ++ [Jmc (n + bl + sl + 2)]
          ++ stmts_code ++ [Jmp n], bl + sl + 2)
     where
```



```
     (bexp_code, bl) = bexp_trans bexp st
     (stmts_code, sl) = stmts_trans stmts st (n + bl + 1)

stmt_trans (If bexp stmts1 stmts2) st n =
     (bexp_code ++ [Jmc (n + bl + s1 + 2)]
          ++ stmts1_code ++ [Jmp (n + bl + s1 + s2 + 2)]
          ++ stmts2_code, bl + s1 + s2 + 2)
     where
     (bexp_code, bl) = bexp_trans bexp st
     (stmts1_code, s1) = stmts_trans stmts1 st (n + bl + 1)
     (stmts2_code, s2) = stmts_trans stmts2 st (n + bl + s1 + 2)

exp_trans ::
     exp_type -> symbol_table_type -> ([cmd_type], num)

exp_trans (Variable ident) st =
     ([Lod num], 1)
     where
     Entry num = read_table ident st
exp_trans (Number (Num n)) st =
     ([Lit n], 1)
exp_trans (Compound exp1 op exp2) st =
     (exp1_code ++ exp2_code ++ [cmd], e1 + e2 + 1)
     where
     (exp1_code, e1) = exp_trans exp1 st
     (exp2_code, e2) = exp_trans exp2 st
     cmd = Ad, op = ArithOp Plus
         = Sb, op = ArithOp Minus
         = Ml, op = ArithOp Times
         = Dv, op = ArithOp Div

bexp_trans ::
     bexp_type -> symbol_table_type -> ([cmd_type], num)

bexp_trans (exp1, relop, exp2) st =
     (exp1_code ++ exp2_code ++ [cmd], e1 + e2 + 1)
     where
     (exp1_code, e1) = exp_trans exp1 st
     (exp2_code, e2) = exp_trans exp2 st
     cmd = Eq, relop = RelOp EQ
         = Ne, relop = RelOp NE
         = Gt, relop = RelOp GT
         = Ge, relop = RelOp GE
```

```
        = Lt, relop = RelOp LT
        = Le, relop = RelOp LE
```

A.5.2 Die Datei psa

```
#! /global/bin/mira -exp
output

%include "psa_compiler"
output =
     concat (map compile (tl $*))
     || Fuer jeden Parameter des Aufrufs von 'psa' wird mit compile
     || die Liste der entsprechenden "system messages" berechnet; diese
     || werden konkateniert, so dass man wiederum eine Liste von
     || "system messages" erhaelt. Die Auswertung dieser Liste
     || bewirkt, dass die gewuenschten Dateien angelegt werden.
```

A.6 Aufgabe 6

A.6.1 Die Datei psa+_lex.m

```
|| Definition der Symbolklassen und des Scanners fuer die
|| Modellsprache PSA+

%export string arithop relop token lex

string == [char]

arithop ::= Plus | Minus | Times | Div

relop ::= EQ | NE | LT | LE | GT | GE

|| Erweiterung um die beiden Schluesselworter read und write:

token ::=
       ArithOp arithop | AssignOp | ClosePar | DoSym | ElseSym |
       EndSym | FullStop | Ident string | IfSym | Num num |
       OpenPar | RelOp relop | Semicolon | ThenSym | VarSym |
       WhileSym | ReadSym | WriteSym | String string

lex :: [char] -> [token]

lex [] = []
```



```
lex (h : t)
     = read_String t, h = '"'
     = check_Id (h : takewhile letter_or_digit t) :
          lex (dropwhile letter_or_digit t), letter h
     = Num (numval (h : takewhile digit t)) :
          lex (dropwhile digit t), digit h
     = convert_Sym (h : t), otherwise
       where
       letter_or_digit c = letter c \/ digit c

read_String t
     = error "missing end of string", rest = []
     = String (takewhile stringchar t) : lex (tl rest), hd rest = '"'
     = error "illegal character for string constant"   , otherwise
       where
       stringchar c =
          letter c \/ digit c \/ member " !()?-.:," c
       rest = dropwhile stringchar t

check_Id str
        = IfSym     , str="if"
        = ThenSym   , str="then"
        = ElseSym   , str="else"
        = EndSym    , str="end"
        = WhileSym  , str="while"
        = DoSym     , str="do"
        = VarSym    , str="var"
        = ReadSym   , str="read"
        = WriteSym  , str="write"
        = Ident str , otherwise

convert_Sym (h : t)
     = Semicolon : rest             , h = ';'
     = ArithOp Plus : rest          , h = '+'
     = ArithOp Minus : rest         , h = '-'
     = ArithOp Times : rest         , h = '*'
     = ArithOp Div : rest           , h = '/'& (t = [] \/ hd t ~= '=')
     = RelOp EQ : rest              , h = '='
     = RelOp NE : lex (tl t)        , h ='/' & t ~= [] & hd t = '='
     = RelOp LT : rest              , h ='<' & (t = [] \/ hd t ~= '=')
     = RelOp LE : lex (tl t)        , h ='<' & t ~= [] & hd t = '='
     = RelOp GT : rest              , h ='>' & (t = [] \/ hd t ~= '=')
     = RelOp GE : lex (tl t)        , h ='>' & t ~= [] & hd t = '='
```

```
     = OpenPar : rest              , h = '('
     = ClosePar : rest             , h = ')'
     = FullStop : rest             , h = '.'
     = AssignOp : lex (tl t)       , h = ':' & t ~= [] & hd t = '='
     = rest                        , member "\n\t " h
     = error ("Illegal character: "++ [h])
                                   , otherwise
          where
          rest = lex t
```

A.6.2 Die Datei psa+_parser.m

```
|| Dieser Modul enthaelt den Parser fuer PSA+. Er exportiert
|| die Token und den Typ der Syntaxbaeume
|| von PSA+, ausserdem den Scanner und den Parser fuer PSA+.

|| Er stuetzt sich auf die Symbolklassenbeschreibung und
|| den Scanner aus dem Modul psa_lex sowie den
|| Modul parser_tool

%include "psa+_lex"

%include "parser_tool" {token==token; scanner=lex;}

%export
     token arithop relop string psa_parser
     program_type declaration_type stmt_seq_type
     stmt_type exp_type bexp_type

|| Der Parser heisst psa_parser, er nimmt als Eingabe
|| eine Zeichenfolge
psa_parser :: parser_type program_type
psa_parser = parser program_parser

program_type ::= Program [declaration_type] stmt_seq_type
program_parser =
     ((parse_star decl_parser) $parse_seq stmt_seq_parser
          $parse_seq (parse_lit FullStop)) $modify mk_prog
     where
     mk_prog (decls, (stmts, fs)) = Program decls stmts

declaration_type == [token]
|| Dies muss natuerlich eine nicht-leere Liste von
```



```
|| Bezeichnern sein

decl_parser =
     ((parse_lit VarSym) $parse_seq (id_parser) $parse_seq
          (parse_star (id_parser)) $parse_seq
          (parse_lit Semicolon)) $modify mk_decl
     where
     mk_decl (var, (id, (id_list, semi))) = id:id_list

id_parser (Ident i : anything) =
     Parsed (Ident i, anything)
id_parser anything_else =
     Fail

stmt_seq_type == [stmt_type]
stmt_seq_parser = parse_star stmt_parser


stmt_type ::=
              Assignment token exp_type |
              If bexp_type stmt_seq_type stmt_seq_type |
              While bexp_type stmt_seq_type |
              Read token |
              WriteExp exp_type |
              WriteStr token

stmt_parser (Ident i : rest) =
     assignment_parser (Ident i : rest)
stmt_parser (IfSym : rest) =
     if_parser (IfSym : rest)
stmt_parser (WhileSym : rest) =
     while_parser (WhileSym : rest)
stmt_parser (ReadSym : rest)  =
     read_parser (ReadSym : rest)
stmt_parser (WriteSym : rest) =
     write_parser (WriteSym : rest)
stmt_parser anything_else =
     Fail

read_parser =
     ((parse_lit ReadSym) $parse_seq id_parser)
          $modify (Read.snd)
```

```
write_parser (WriteSym : String str : rest) =
     Parsed (WriteStr (String str), rest)
write_parser =
     ((parse_lit WriteSym) $parse_seq exp_parser)
          $modify (WriteExp.snd)

assignment_parser =
     ((id_parser) $parse_seq (parse_lit AssignOp)
          $parse_seq exp_parser) $modify mk_assign_stmt
     where
     mk_assign_stmt (id, (ass, exp)) = Assignment id exp

if_parser =
     ((parse_lit IfSym) $parse_seq bexp_parser
          $parse_seq (parse_lit ThenSym)
          $parse_seq stmt_seq_parser $parse_seq
          (parse_lit ElseSym) $parse_seq stmt_seq_parser $parse_seq
          (parse_lit EndSym)) $modify mk_if_stmt
     where
     mk_if_stmt (i, (e, (t, (s1, (l, (s2, d)))))) = If e s1 s2

while_parser =
     ((parse_lit WhileSym) $parse_seq bexp_parser
          $parse_seq (parse_lit DoSym) $parse_seq
          stmt_seq_parser $parse_seq
          (parse_lit EndSym)) $modify mk_while_stmt
     where
     mk_while_stmt (w, (bexp, (do, (stmts, end)))) =
          While bexp stmts

exp_type ::=
     Variable token | Number token | Compound exp_type token exp_type

exp_parser (Ident i : rest) = Parsed (Variable (Ident i), rest)
exp_parser (Num n : rest)   = Parsed (Number (Num n), rest)
exp_parser anything_else    = compound_parser anything_else

num_parser (Num n : anything) = Parsed (Num n, anything)
num_parser anything_else = Fail

compound_parser =
     ((parse_lit OpenPar) $parse_seq exp_parser
          $parse_seq (arithop_parser)
```



```
          $parse_seq exp_parser $parse_seq
          (parse_lit ClosePar)) $modify mk_compound_exp
     where
     mk_compound_exp (k1, (e1, (ar, (e2, k2)))) =
          Compound e1 ar e2

arithop_parser (ArithOp op : anything) = Parsed (ArithOp op, anything)
arithop_parser anything_else = Fail

bexp_type == (exp_type, token, exp_type)

bexp_parser =
        (exp_parser $parse_seq (relop_parser) $parse_seq exp_parser)
        $modify mk_bexp
        where
        mk_bexp (e1,(rel,e2)) = (e1,rel,e2)

relop_parser (RelOp op :anything) = Parsed(RelOp op,anything)
relop_parser anything_else = Fail
```

A.6.3 Die Datei psa+_analyser.m

Die Datei `psa_analyser.m` wird um die folgenden Gleichungen erweitert:

```
stmt_check tab (Read i) =
     declared tab i
stmt_check tab (WriteStr str) =
     True
stmt_check tab (WriteExp exp) =
     exp_check tab exp
```

A.7 Aufgabe 7

A.7.1 Die Datei machine+.m

```
|| Dieser Modul realisiert eine abstrakte Stackmaschine mit
|| Hauptspeicher. Er stellt eine Funktion run zur Verfuegung, die ein
|| Maschinenprogramm aus einer Datei liest und ausfuehrt, sowie den
|| Typ der Maschinenbefehle, damit man auch ausserhalb dieses Moduls
|| Programme schreiben kann.

%export run cmd_type
```

```
|| Eine Maschine besteht aus einem Befehlszaehler, einem Datenkeller
|| und einem Hauptspeicher sowie einem Eingabe- und einem Ausgabeband.
|| Das Eingabeband wird als Liste von Zahlen angenommen. Ein
|| vorgeschalteter Scanner uebernimmt es, die Standardeingabe
|| entsprechend den in der Definition der A+ gegebenen Regeln in eine
|| Liste von Zahlen umzuwandeln.
|| Der Typ des Hauptspeichers wird importiert.

%include "store_type"

machine_type == (num, [stack_entry], store_type, [num], [char])

|| Der Datenkeller kann als Eintraege Zahlen und boole'sche Werte
|| enthalten

stack_entry ::= Bool bool | Num num

|| Es gibt die folgenden Befehle:
|| Lesen und Beschreiben des Hauptspeichers, bedingten und unbedingten
|| Sprung, arithmetische und relationale Operationen, Lesen und
|| Schreiben, Anhalten.

cmd_type ::=
          Lod num | Sto num | Jmp num | Jmc num |
          Lit num | Ad | Sb | Ml | Dv |
          Eq | Ne | Gt | Ge | Lt | Le | Halt |
          Put | Get | Text [char]

run :: [cmd_type] -> [char]

run cmds =
     execute cmds (0, [], empty_store, numbers $-, [])
     where
     execute cmds (ip, stack, store, input, output)
          = output, next_cmd = Halt
          = output ++ execute cmds (exec next_cmd
               (ip, stack, store, input, [])), otherwise
               where
               next_cmd
                    = cmds!ip, 0 <= ip <= #cmds - 1
                    = error "undefined code address", otherwise

exec :: cmd_type -> machine_type -> machine_type
```



```
exec (Lod a) (ip, stack, store, input, output)
     = error "reading uninitialized storage cell", rs = NoEntry
     = (ip + 1, (Num (get rs)) : stack, store, input, output)
              , otherwise
          where
          rs = read_store a store
          get (Entry n) = n

exec (Sto a) (ip, Num n : stack, store, input, output) =
     (ip + 1, stack, insert_store a n store, input, output)
exec (Sto a) anything_else =
     error "attempt to store an inproper value"

exec (Jmp a) (ip, stack, store, input, output) =
     (a, stack, store, input, output)

exec (Jmc a) (ip, Bool b : stack, store, input, output)
     = (a, stack, store, input, output) , b = False
     = (ip + 1, stack, store, input, output) , otherwise
exec (Jmc a) anything_else = error "boolean value expected"

exec (Lit n) (ip, stack, store, input, output) =
     (ip + 1, (Num n) : stack, store, input, output)

exec (Ad) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Num (a1 + a2) : stack, store, input, output)
exec (Ad) anything_else = error "missing or wrong operands for Ad"

exec (Sb) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Num (a1 - a2) : stack, store, input, output)
exec (Sb) anything_else = error "missing or wrong operands for Sb"

exec (Ml) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Num (a1 * a2) : stack, store, input, output)
exec (Ml) anything_else = error "missing or wrong operands for Ml"

exec (Dv) (ip, Num a2 : Num a1 : stack, store, input, output)
     = (ip + 1, Num (a1 div a2) : stack, store, input, output)
          , a2 ~= 0
     = error "division by zero", otherwise
exec (Dv) anything_else = error "missing or wrong operands for Dv"
```

```
exec (Eq) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Bool (a1 = a2) : stack, store, input, output)
exec (Eq) anything_else = error "missing or wrong operands for Eq"

exec (Ne) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Bool (a1 ~= a2) : stack, store, input, output)

exec (Ne) anything_else = error "missing or wrong operands for Ne"

exec (Gt) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Bool (a1 > a2) : stack, store, input, output)
exec (Gt) anything_else = error "missing or wrong operands for Gt"

exec (Ge) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Bool (a1 >= a2) : stack, store, input, output)
exec (Ge) anything_else = error "missing or wrong operands for Ge"

exec (Lt) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Bool (a1 < a2) : stack, store, input, output)
exec (Lt) anything_else = error "missing or wrong operands for Lt"

exec (Le) (ip, Num a2 : Num a1 : stack, store, input, output) =
     (ip + 1, Bool (a1 <= a2) : stack, store, input, output)
exec (Le) anything_else = error "missing or wrong operands for Le"

exec (Put) (ip, (Num n) : stack, store, input, output) =
      (ip+1, stack, store, input, output++show n ++"\n")
exec (Put) anything_else = error "missing or wrong values for Put"

exec (Get) (ip, stack, store, input, output)
      = (pc, Num (hd input) : stack, store, tl input, output),
                              input ~= []
      = error "missing or wrong values for Get", otherwise
          where
          pc = seq (hd input) (ip+1)
          || An dieser Stelle findet die Synchronisation von Ein-
          || und Ausgabe statt: Die Verwendung des seq-Operators
          || erzwingt, dass - unabhaengig davon, wann wieder Werte
          || auf dem Datenkeller gebraucht werden - eine Eingabe
          || gelesen werden muss, bevor der naechste Wert des
          || Programmzaehlers und damit der naechste auszufuehrende
          || Befehl bestimmt werden kann.
```



```
exec (Text out) (ip, stack, store, input, output) =
         (ip+1, stack, store, input, output++out++"\n")

numbers :: [char] -> [num]
|| Diesen Funktion wandelt die Eingabe in eine Liste von Zahlen um

numbers =
     (map (numval)).(filter (~= [])).lines.
          (map convert).(filter relevant)
     where
     convert c
          = c    , digit c
          = '\n' , otherwise
     relevant c
          = digit c \/ member " \n\t" c
```

A.7.2 Die Datei `psa+_compiler.m`

```
|| Dieser Modul enthaelt den eigentlichen Uebersetzer: die Funktion
|| compile.

%include "machine+"

%include "psa+_parser"

%include "psa+_analyser"

%export cmd_type compile

compile :: [char] -> [sys_message]

|| compile akzeptiert einen Dateinamen mit und ohne die Endung ".psa";
|| fehlt die Endung ".psa", so wird sie automatisch ergaenzt. compile
|| legt das Programm fuer die A-Maschine in einer Datei gleichen
|| Namens, aber mit der Endung ".psx" ab; ausserdem wird ein "magic
|| string", der das gespeicherte A-Code-Programm auf der A-Maschine
|| ausfuehrt, unter dem um die Endung ".psa" verkuerzten Dateinamen
|| abgelegt.
||
|| Besteht fuer die angegebene Datei kein Leserecht bzw. besteht fuer
|| das Zielverzeichnis kein Schreibrecht, so wird eine entsprechende
|| Fehlermeldung ausgegeben.
```

```
compile filename =
     gen_file file
     where
     file = filename, take 4 (reverse filename) ~= "asp."
          = reverse (drop 4 (reverse filename)), otherwise


gen_file file
     = [Stderr ("Can't open file: " ++ file ++ ".psa\n")]
          , ~r_permission
     = [Stderr ("Can't write to file: " ++ file ++ ".psx\n")]
          , ~w_permission
     = generate_files, otherwise
          where
          generate_files =
               [Tofile (file ++ ".psx")
                    (lay (map show (trans (read (file++".psa"))))),
               Tofile file (call_machine file),
               System ("chmod 755 " ++ file)]
          r_permission = member (filemode (file ++ ".psa")) 'r'
          w_permission = member (filemode directory) 'w'
          || Um die Schreibberechtigung pruefen zu koennen, muss
          || der Verzeichnisname aus dem Dateinamen extrahiert
          || werden. Es werden drei Faelle unterschieden:
          || - Die Datei liegt im Wurzelverzeichnis;
          || - Die Datei liegt im aktuellen Verzeichnis;
          || - Die Datei liegt in irgend einem anderen Verzeichnis.
          dir = reverse (drop 1 ((dropwhile (~='/').reverse) file))
          directory
               = "/", (dir = "") & (hd file = '/')
               = ".", (dir = "") & (hd file ~= '/')
               = dir, otherwise


call_machine file =
     "#! /global/bin/mira -exp\n"++
     "run (readvals \"" ++ filename ++ ".psx\")\n" ++
     "\n" ++
     "%include \""++ getenv "PSA" ++ "/machine+\"\n"
     || Der Zugriffspfad zur abtrakten Maschine wird in der
     || Umgebungsvariablen "PSA" nachgeschlagen.
     where
     filename = reverse ((takewhile (~='/').reverse) file)
```



```
|| Es folgen die Uebersetzungsfunktionen
||
|| Fuer jeden Typ von Sprachkonstrukten (Anweisungsfolgen,
|| Anweisungen, arithmetische und boole'sche Ausdruecke) gibt es eine
|| eigene Uebersetzungsfunktion. Sie alle benoetigen die
|| Symboltabelle aus der semantischen Analyse.
|| Die Uebersetzungsfunktionen, die Sprungbefehle erzeugen, benoetigen
|| noch einen weiteren Parameter: die Anfangsadresse des von ihnen zu
|| erzeugenden Codes.
||
|| Hauptfunktion der Uebersetzung ist die Funktion trans, die als
|| Eingabe ein PSA-Programm in Form einer ASCII-Datei nimmt, die
|| Analysephase des Compilers durchfuehrt und anschliessend die
|| Uebersetzungsfunktionen aufruft.

symbol_table_type == table_type token num

|| trans nimmt als Eingabe ein PSA-Programm in Form einer
|| ASCII-Datei und berechnet das zugeh"orige A-Code-Programm.
|| Dieses A-Code-Programm endet mit einem Halt-Befehl.

trans :: [char] -> [cmd_type]

trans input =
     stmts_code ++ [Halt]
     where
     Program decls stmts = psa_parser input
     st = analyse (Program decls stmts)
     (stmts_code, stmts_len) = stmts_trans stmts st 0

stmts_trans ::
     stmt_seq_type -> symbol_table_type -> num -> ([cmd_type], num)

stmts_trans [] st n = ([], 0)
stmts_trans (stmt:stmts) st n =
     (stmt_code ++ stmts_code, stmt_len + stmts_len)
     where
     (stmt_code, stmt_len) = stmt_trans stmt st n
     (stmts_code, stmts_len) = stmts_trans stmts st (n + stmt_len)

stmt_trans ::
     stmt_type -> symbol_table_type -> num -> ([cmd_type], num)
```

```
stmt_trans (Assignment id exp) st n =
     (exp_code ++ [Sto num], exp_len + 1)
     where
     (exp_code, exp_len) = exp_trans exp st
     Entry num = read_table id st
stmt_trans (While bexp stmts) st n =
     (bexp_code ++ [Jmc (n + bl + sl + 2)]
          ++ stmts_code ++ [Jmp n], bl + sl + 2)
     where
     (bexp_code, bl) = bexp_trans bexp st
     (stmts_code, sl) = stmts_trans stmts st (n + bl + 1)

stmt_trans (If bexp stmts1 stmts2) st n =
     (bexp_code ++ [Jmc (n + bl + s1 + 2)]
          ++ stmts1_code ++ [Jmp (n + bl + s1 + s2 + 2)]
          ++ stmts2_code, bl + s1 + s2 + 2)
     where
     (bexp_code, bl) = bexp_trans bexp st
     (stmts1_code, s1) = stmts_trans stmts1 st (n + bl + 1)
     (stmts2_code, s2) = stmts_trans stmts2 st (n + bl + s1 + 2)

stmt_trans (Read id ) st n =
        ([Get ,Sto num ],2)
        where
        Entry num = read_table id st

stmt_trans (WriteExp exp ) st n =
        ( exp_code ++ [Put ] ,exp_len + 1)
        where
        (exp_code,exp_len) = exp_trans exp st

stmt_trans (WriteStr (String str)) st n =
        ([Text str],1)

exp_trans ::
     exp_type -> symbol_table_type -> ([cmd_type], num)

exp_trans (Variable ident) st =
     ([Lod num], 1)
     where
     Entry num = read_table ident st
exp_trans (Number (Num n)) st =
```



```
     ([Lit n], 1)
exp_trans (Compound exp1 op exp2) st =
     (exp1_code ++ exp2_code ++ [cmd], e1 + e2 + 1)
     where
     (exp1_code, e1) = exp_trans exp1 st
     (exp2_code, e2) = exp_trans exp2 st
     cmd = Ad, op = ArithOp Plus
         = Sb, op = ArithOp Minus
         = Ml, op = ArithOp Times
         = Dv, op = ArithOp Div

bexp_trans ::
     bexp_type -> symbol_table_type -> ([cmd_type], num)
bexp_trans (exp1, relop, exp2) st =
     (exp1_code ++ exp2_code ++ [cmd], e1 + e2 + 1)
     where
     (exp1_code, e1) = exp_trans exp1 st
     (exp2_code, e2) = exp_trans exp2 st
     cmd = Eq, relop = RelOp EQ
         = Ne, relop = RelOp NE
         = Gt, relop = RelOp GT
         = Ge, relop = RelOp GE
         = Lt, relop = RelOp LT
         = Le, relop = RelOp LE
```

A.7.3 Die Datei psa+

```
#! /global/bin/mira -exp
output

%include "/home/alpha/maassen/swp/psa+_compiler"
|| Leider erlaubt es die Miranda-Syntax nicht, den Pfad zum Compiler
|| an dieser Stelle aus der Umgebungsvariablen "PSA" einzulesen.
output =
     concat (map compile (tl $*))
```

B Übungsaufgaben des Steilkurses Miranda

- Schreiben Sie eine Funktion, die aus einer Liste die Teilliste der Einträge mit ungeradem Index herausfiltert.
- Schreiben Sie Programme für die Bibliotheksfunktionen

abs and concat const drop dropwhile filter fst hd id index init last lines map map2 max max2 member merge mkset postfix product rep repeat reverse spaces until

- Schreiben Sie eine Funktion, die zu einer ganzen Zahl testet, ob diese eine Quadratzahl ist.
- Schreiben Sie eine Funktion, die zu zwei Punkten in einem zweidimensionalen Koordinatensystem deren Abstand berechnet.
- Schreiben Sie eine Funktion, die die benachbarten Einträge einer Liste vertauscht, d.h. den 1. mit dem 2., den 3. mit dem 4. usw.
- Schreiben Sie eine Funktion, die zwei sortierte Listen verschmelzt.
- Schreiben Sie eine Funktion, die zwei sortierte Listen verschmelzt, dabei jedoch alle Duplikate aus dem Ergebnis entfernt. (Benutzen Sie nicht die in diesem Fall sehr ineffiziente Bibliotheksfunktion **mkset**.)
- Schreiben Sie eine Funktion, die zu einer Funktion f und einem Startwert i die Liste [i, fi, ffi, fffi, ...] liefert.
- Schreiben Sie eine Funktion, die zu einer Liste $[a_0, a_1, a_2 \ldots]$ und zu einer streng monoton wachsenden Folge $[i_0, i_1, i_2 \ldots]$, die ebenfalls als Liste gegeben ist, die Liste $[a_{i_0}, a_{i_1}, a_{i_2} \ldots]$ ausgibt.
- Schreiben Sie eine Funktion, die zu einer Liste von Listen, deren Einträge Zahlen sind, die Summe der jeweils ersten Listeneinträge berechnet.
- Punkte in einem n-dimensionalen Koordinatensystem kann man durch eine Liste der n Koordinaten beschreiben. Definieren Sie eine Funktion, die zu zwei Punkten in solcher Darstellung den Abstand berechnet.
- Definieren Sie (mit und ohne fold) eine Funktion, die zu einer Liste von Zahlen die mit dem kleinsten Betrag ermittelt.
- Ein Text besteht aus Worten, die durch Leerzeichen, **\t** und/oder **\n** getrennt sind. Schreiben Sie eine Funktion, die einen Text einliest und in Zeilen der Länge 80 umbricht, wobei innerhalb einer Zeile je zwei Worte durch genau ein Leerzeichen getrennt sind.
- Schreiben Sie eine Funktion, die die Zeichenkettendarstellung einer Zahl in die Zahl selber umwandelt.
- Schreiben Sie eine Funktion, die zu zwei ganzen Zahlen deren Quotienten in Dezimalbruchdarstellung mit einer beliebigen, vom Benutzer anzugebenden Zahl von Nachkommastellen ausgibt.
- Definieren Sie die Liste der Fakultäten.

- Schreiben Sie eine Funktion **choose :: num -> [*] -> [[*]]**, die zu einer Zahl k und einer Liste l die Liste aller Segmente von l, die die Länge k haben, liefert.
- Definieren Sie Funktionen **all** und **none** (jeweils mit dem Typ **(* -> bool) -> [*] -> bool**), die testen, ob alle bzw. keiner der Einträge einer Liste ein Prädikat erfüllen. Inwieweit können diese Funktionen auch für unendliche Listen vernünftige Ergebnisse liefern?
- Definieren Sie eine Funktion, die zu zwei totalen Funktionen des Typs **num -> num** den Wert **False** liefert, falls diese sich an mindestens einer Stelle unterscheiden, und undefiniert ist, wenn die Funktionen gleich sind.
- Programmieren Sie Insertion Sort, wobei das Sortierkriterium als Parameter angegeben werden soll.
- Erzeugen Sie nach der Methode des Sieb des Erathostenes die Liste der Primzahlen.
- Definieren Sie einen Datentyp **complex** für komplexe Zahlen, auf dem die Operationen Addition, Subtraktion, Multiplikation, Division, Test auf Gleichheit zur Verfügung stehen.
- Gegeben sei der folgende algebraische Typ von arithmetischen Ausdrücken einer Variablen X :

```
expr ::=
        Add expr expr | Sub expr expr | Mult expr expr |
        Div expr expr Log expr | Power expr expr |
        Const num | X
```

  Dabei soll **Log** den natürlichen Logarithmus darstellen. Schreiben Sie ein Miranda-Programm, das arithmetische Ausdrücke der obigen Form symbolisch nach X differenziert.

  Lösen Sie diese Aufgabe auch für den nachfolgend angegebenen Typ:

```
expr ::=
        Const num | X | Neg expr | Add [expr] |
        Mult [expr] | Div expr expr
```

- Definieren Sie einen geeigneten Datentyp zur Darstellung eines DEA und implementieren Sie eine Funktion, die zu einem gegebenen DEA und einem Wort testet, ob das Wort in der von dem DEA erkannten Sprache liegt.

- Schreiben Sie ein Miranda-Programm, welches zu einer rechtslinearen Grammatik G, die nur Regeln der Form `A -> a B` und `A -> a` enthält, und einem Wort w feststellt, ob w in L(G) liegt. Definieren Sie hierzu die algebraischen Typen

```
nonterminal ::=
        A|B|C|D|E|F|G|H|I|J|K|L|M|
        N|O|P|Q|R|S|T|U|V|W|X|Y|Z
production  ::=
        Rule nonterminal char nonterminal |
        Termrule nonterminal char
```

  und schreiben Sie eine Funktion

```
is_in_l :: [production] -> [char] -> bool
```

  die testet, ob ein Wort in einer durch eine Liste von Produktionen gegebenen Grammatik liegt. (Das Startsymbol sei immer S.)

- Schreiben Sie eine Funktion, die zu einer Liste von Objekten einen möglichst gut höhenbalanzierten Binärbaum erzeugt, in dem die Einträge der Liste in natürlicher Reihenfolge an den Blättern stehen. Verallgemeinern Sie dies auf Bäume beliebiger Stelligkeit.
- Definieren Sie `foldtree` und `maptree` für Bäume beliebiger Stelligkeit.
- Definieren Sie mit Hilfe von `fold` eine Funktion, die zu einer Liste von Funktionen bzw. zu einer Liste von Funktionsnamen und einer Interpretation dieser Namen die Verkettung dieser Funktionen berechnet. Definieren Sie diese Funktion auch ohne Verwendung von `fold`.
- Implementieren Sie Prioritäts-Warteschlangen mit ***leftist heaps***, d. h. mit heaps, bei denen der kürzeste Weg zu einem Blatt von jedem Knoten aus nach rechts führt.
- Definieren Sie einen geeigneten Datentyp zur Darstellung von ungerichteten Graphen und schreiben Sie eine Funktion, die zu einem Graphen dessen Zusammenhangskomponenten ermittelt.
- Implementieren Sie Pushdown-Automaten.
- Schreiben Sie eine Funktion, die testet, ob ein Binärbaum ausgeglichen ist.
- Als Kosten eines bewerteten Baumes definieren wir die Summe aus den Produkten aus Gewichten und Weglängen an allen Knoten. Schreiben Sie eine Funktion, die die Kosten eines Baumes ermittelt.

- Das verallgemeinerte Hamming-Problem erhält man, indem man von der festen Liste mit den Faktoren [2, 3, 5] zu einer beliebigen Liste mit Faktoren übergeht. Schreiben Sie eine Funktion für das verallgemeinerte Hamming-Problem.
- Definieren Sie das Pascal'sche Dreieck.
- Ändern Sie das hangman-Programm dahingehend, daß die zu ratenden Wörter nicht mehr von Hand eingegeben werden, sondern aus einer Datei, die beim Start des Spieles angegeben wird, eingelesen werden. Das Spiel soll nacheinander alle in der Datei stehenden Wörter verwenden. Dabei sei ein Wort eine ununterbrochene Folge von Kleinbuchstaben, alle anderen Zeichen sollen überlesen werden.
- Programmieren Sie das hangman-Programm unter Verwendung der Funktionen aus dem Bibliotheksmodul `interaction.m` neu.

Literatur

[Backus] John Backus: ***Can Programming Be Liberated from the van Neumann Style? A Functional Style and Its Algebra of Programs***, Comm. ACM, Vol. 21, No. 8 (1978), 613-641

[Bird] Richard Bird and Philip Wadler: ***Introduction to Functional Programming***, Prentice Hall, New York (1988)

[Gulbins] Jürgen Gulbins: *UNIX*, Springer-Verlag Berlin, Heidelberg (1984)

[Hughes] John Hughes: ***Why Functional Programming Matters***, Programming Methodology Group Report 16, University of Göteborg and Chalmers University of Technology, Göteborg (1984)

[Reade] Chris Reade: ***Elements of Functional Programming***, Addison-Wesley, Reading, Mass. (1989)

[Turner1] David Turner: ***An Overview of Miranda***, in: SIGPLAN Notices, Dezember 1986, (1986)

[Turner2] David Turner: ***Miranda: A non-strict functional language with polymorphic types***, in: Proc. IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy 1985, LNCS 201, Springer Verlag Berlin - Heidelberg - New York - Tokyo, (1986)

[Wikstrom] Åke Wikström: ***Functional Programming Using Standard ML***, Prentice Hall, New York (1987)