Praktikum Compilerbau

Michael Jäger, Hellwig Geisse, Uwe Meyer

7. April 2021

(Version 1.4.2 vom 2.4.2021)

Inhaltsverzeichnis

Vorbemerkung

Dieses Skript enthält die notwendigen Grundlagen für das Praktikum Compilerbau und sollte zusammen mit den Informationen zur Vorlesung (siehe Moodle) im Modul 'CS 1019 Compilerbau' verwendet werden.

Dieses Skript wurde initial von Prof. Dr. Michael Jäger erstellt und enthält die SPL-Sprachdefinition und die ECO32-Zielplattform, die vom Kollegen Prof. Dr. Hellwig Geisse und seinem Team an der THM entwickelt wurden. Die in diesen Praktikumsunterlagen enthaltenen Sprach- und Zielplattformbeschreibungen sind aus der SPL- und ECO32-Dokumentation entnommen.

Für das Wintersemester 2019/2020 wurden sowohl Änderungen an den Quellcode-Dateien, die den Studierenden zur Verfügung gestellt werden, vorgenommen als auch am Referenz-Compiler. Im vorliegenden Skript wurde dies von Prof. Dr. Uwe Meyer entsprechend berücksichtigt.

Falls Sie einen Fehler finden oder einen Verbesserungsvorschlag haben, senden Sie bitte eine Mail an uwe.meyer@mni.thm.de.

1 Einführung

Das Compilerbau-Praktikum ist ein Software-Entwicklungsprojekt mit dem Ziel, für die einfache imperative Programmiersprache SPL (Simple Programming Language) einen Compiler für die RISC-Maschine ECO32 zu entwickeln.

Der Compiler wird in der Programmiersprache Java oder C implementiert, wobei der Scanner mit dem Scannergenerator *jflex* und der Parser mit dem Parsergenerator *cup* erzeugt werden (Java), beziehungsweise mit *flex* und *bison*.

2 SPL Sprachdefinition

2.1 Allgemeines

SPL ("Simple Programming Language") ist eine einfache prozedurale Programmiersprache. Sie enthält einen vordefinierten primitiven Typ für ganze Zahlen sowie einen Typkonstruktor für Felder. SPL benutzt außerdem Wahrheitswerte; es lassen sich aber weder Variablen von diesem Typ anlegen noch gibt es Literale dieses Typs.

SPL kennt Prozeduren (aber keine Funktionen), sowohl mit Wert- als auch mit Referenzparametern. Felder müssen als Referenzparameter übergeben werden. Prozeduren können lokale Variablen vereinbaren. Globale Variablen gibt es nicht, ebensowenig wie ineinander geschachtelte Prozedurvereinbarungen.

Als Anweisungen stehen die bedingte Anweisung (ein- und zweiarmig), die abweisende Schleife, die Zuweisung, der Aufruf einer Prozedur sowie die zusammengesetzte Anweisung für Anweisungsfolgen zur Verfügung.

Ausdrücke werden über ganzen Zahlen konstruiert. Es stehen sechs Vergeichsoperatoren, die vier Grundrechenarten und die Negation zur Verfügung. Klammern erlauben die beliebige Zusammenfassung von Teilausdrücken. Für den vordefinierten Typ "ganze Zahl" können Literale in verschiedenen Darstellungen notiert werden.

Felder werden durch ganzzahlige Ausdrücke indiziert, sowohl auf der linken wie auf der rechten Seite von Zuweisungen bzw. in Argumentausdrücken. Zulässige Indexausdrücke liefern einen Wert im Bereich 0..(n-1), wenn das Feld n Elemente hat. Die Indizierung eines Feldes außerhalb dieses Bereichs ruft einen Laufzeitfehler hervor.

Die Laufzeitbibliothek bietet Prozeduren zur Eingabe und Ausgabe ganzer Zahlen sowie einzelner Zeichen auf dem Textbildschirm an. Sie stellt auch eine Prozedur zum sofortigen Beenden eines Programms zur Verfügung. Eine weitere Prozedur gibt die seit dem Start des Programms vergangene Zeit zurück. Auf dem Graphikbildschirm können einzelne Pixel, gerade Linien oder Kreise in beliebigen Farben gezeichnet werden. Es ist auch möglich, den ganzen Grafikbildschirm mit einer beliebigen Farbe zu füllen.

Am Ende dieser Sprachdefinition befindet sich ein Beispielprogramm, in dem einige der Möglichkeiten von SPL demonstriert werden.

2.2 Lexikalische Konventionen

Groß- und Kleinschreibung wird unterschieden.

Leerzeichen und horizontale Tabulatoren trennen Token, haben aber sonst keine Bedeutung für das Programm. Zeilenumbrüche zählen ebenfalls als Leerzeichen.

Ein Kommentar wird durch einen doppelten Schrägstrich ("//") eingeleitet und reicht bis zum Ende der Zeile, in der er steht. Auch er trennt Token und hat für das Programm keine Bedeutung.

Beispiel:

```
// Das ist ein Kommentar.
```

Bezeichner bestehen aus beliebig vielen Buchstaben, Ziffern und Unterstrichen. Sie müssen mit einem Buchstaben beginnen, wobei der Unterstrich als Buchstabe betrachtet wird.

Beispiel:

```
das_ist_ein_Bezeichner
```

Zahlen werden durch Aneinanderreihung von Dezimalziffern gebildet. Alternativ können Zahlen im Hexadezimalsystem angegeben werden. Sie beginnen mit dem Präfix “0x” und enthalten Hexadezimalziffern (0-9,a-f). Die Hexadezimalziffern a-f können auch groß geschrieben werden. Eine dritte Alternative für die Darstellung von Zahlen ergibt sich durch den Einschluss genau eines Zeichens in Apostrophe. Die dargestellte Zahl ist dann der ASCII-Code des Zeichens zwischen den Apostrophen. Die Zeichenfolge \n gilt dabei als ein Zeichen mit der Bedeutung “Zeilenumbruch” (0x0a). Zahlen müssen im Bereich 0..2^31-1 liegen; zur Darstellung von negativen Zahlen steht der unäre Operator “-” zur Verfügung (siehe unten). Die Interpretation von Zahlen, die aus dem angegebenen Bereich herausfallen, ist undefiniert.

Beispiele:

```
1234
0x1a2f3F4e
'a'
'\n'
```

SPL enthält die folgenden reservierten Wörter, die nicht als Bezeichner benutzt werden können:

```
array else if of proc ref type var while
```

Die folgenden Zeichen und Zeichenkombinationen tragen Bedeutung:

```
( ) [ ] { } = # < <= > >= := : , ; + - * /
```

Alle nichtzulässigen Zeichen oder Zeichenfolgen werden erkannt und zurückgewiesen.

2.3 Das Hauptprogramm

Ein SPL-Programm besteht aus einer Sammlung von Typen- und Prozedur- vereinbarungen ohne bestimmte Reihenfolge. Bezeichner für Typen müssen vor ihrer Benutzung vereinbart werden. Bei Prozedurbezeichnern besteht diese Einschränkung nicht, damit wechselseitig rekursive Prozeduren formuliert werden können.

Mindestens eine Prozedurvereinbarung ist erforderlich, und zwar die der Prozedur mit dem festgelegten Namen “main”. Diese Prozedur hat keine Parameter und wird beim Programmstart automatisch aktiviert.

2.4 Typen und Typvereinbarungen

Es gibt einen vordefinierten primitiven Typ für ganze Zahlen (“int”). Der Bezeichner “int” ist kein reserviertes Wort. Er gilt als implizit vor allen Benutzer-Deklarationen vereinbart.

Der Datentyp-Konstruktor “array” konstruiert ein Feld über einem Basistyp. Die Feldgröße wird statisch zum Übersetzungszeitpunkt festgelegt und ist Bestandteil des Typs. Der Basistyp kann irgendein beliebiger Typ sein.

Beispiele:

```
array [3] of int
array [3] of array [5] of int
```

Eine Typvereinbarung verbindet einen Bezeichner mit einem Typ. Sie hat die folgende Struktur:

```
type <name> = <typ> ;
```

Danach kann <name> als Abkürzung für <typ> benutzt werden.

Beispiele:

```
type myInt = int;
type vector = array [5] of int;
type matrix = array [3] of vector;
type mat3 = array [10] of array [20] of vector;
```

Jeder Typausdruck konstruiert einen neuen Typ. Zwei Typen sind gleich, wenn sie durch denselben Typausdruck konstruiert wurden.

Beispiel für gleiche Typen:

```
type typ1 = array [5] of int;
type typ2 = typ1;
```

Beispiel für verschiedene Typen:

```
type typ1 = array [5] of int;
type typ2 = array [5] of int;
```

2.5 Prozedurvereinbarungen

Eine Prozedurvereinbarung verbindet einen Bezeichner mit einer Prozedur. Sie hat folgende Form:

```
proc <name> ( <parameterliste> ) { <deklarationen> <anweisungsliste> }
```

Die optionale <parameterliste> nennt die formalen Parameter der Prodedur. Jeder Parameter wird in einer von zwei Formen angegeben:

```
<name> : <typ>
```

oder

```
ref <name> : <typ>
```

Die erste Form bezeichnet einen Wertparameter, die zweite Form einen Referenzparameter. Die einzelnen Parameter werden in der Liste durch Kommata getrennt. Die Namen der Parameter haben Gültigkeit bis zum Ende der Prozedur.

Die optionalen <deklarationen> vereinbaren lokale Variable. Jede Deklaration hat die folgende Form:

```
var <name> : <typ> ;
```

Diese Namen haben ebenfalls bis zum Ende der Prozedur Gültigkeit. Sie dürfen nicht mit einem Parameternamen kollidieren.

Es ist möglich, Parameter oder lokale Variable mit einem Namen zu vereinbaren, der weiter außerhalb schon eine andere Bedeutung besitzt, d.h. ein Typ- oder Prozedurname ist. Diese äußere Bedeutung wird durch die lokale Bedeutung verdeckt.

Die optionale <anweisungsliste> besteht aus beliebig vielen Anweisungen.

Beispiele:

```
proc nothing() {}
proc copy(i: int, ref j: int) { j := i; }
proc swap(ref i: int, ref j: int) {
  var k: int;
  k := i; i := j; j := k;
}
```

2.6 Anweisungen

Anweisungen dienen zum Erreichen eines Effektes (Ändern des Zustands, Seiteneffekt). Es gibt sechs verschiedene Anweisungen.

Die leere Anweisung besteht nur aus einem Semikolon und bewirkt nichts.

Die Zuweisung hat die Form:

```
<lhs> := <rhs> ;
```

Dabei muss <lhs> eine Variable vom Typ **int** sein (auch indizierte Feldvariablen sind erlaubt) und <rhs> ein Ausdruck von gleichem Typ. Zur Laufzeit wird die Adresse der Variable auf der linken Seite der Zuweisung berechnet bevor die rechte Seite ausgewertet wird. Anschließend wird der Wert der rechten Seite dem Speicherplatz mit dieser Adresse zugewiesen.

Beispiel:

```
x[3] := x[2] * 2;
```

Die bedingte Anweisung kann in einer von zwei Formen vorliegen:

```
if ( <ausdruck> ) <anweisung1>
if ( <ausdruck> ) <anweisung1> else <anweisung2>
```

Dabei ist <ausdruck> ein Ausdruck, der einen Wahrheitswert liefert und <anweisung1> bzw. <anweisung2> jeweils eine Anweisung. Die hinter einem "else" stehende Anweisung gehört zum innersten "if", dem noch kein "else" zugeordnet ist. Zur Laufzeit wird <ausdruck> ausgewertet. Liefert er "wahr", dann wird die <anweisung1> ausgeführt. Liefert er "falsch", dann wird in der ersten Form nichts weiter getan; in der zweiten Form wird in diesem Fall <anweisung2> ausgeführt. In jedem Fall wird die Ausführung mit der auf die "if"-Anweisung folgenden Anweisung fortgeführt.

Beispiele:

```
if (x < 0) x := 42;
if (x < 0) x := 42; else x := 43;
```

Die abweisende Schleife wird wie folgt formuliert:

```
while ( <ausdruck> ) <anweisung>
```

Dabei ist <ausdruck> ein Ausdruck, der einen Wahrheitswert liefert und <anweisung> eine Anweisung. Zur Laufzeit wird <ausdruck> ausgewertet. Liefert er "wahr", dann wird die ausgeführt und die Schleife aufs neue durchlaufen. Liefert er "falsch", wird mit der auf die "while"- Anweisung folgenden Anweisung fortgefahren.

Beispiel:

```
while (x < 10) x := x + 1;
```

Um eine Anweisungfolge syntaktisch als eine einzige Anweisung erscheinen zu lassen, gibt es die zusammengesetzte Anweisung. Sie besteht aus beliebig vielen Anweisungen (möglicherweise auch keiner einzigen), die in geschweifte Klammern gesetzt sind.

Beispiel:

```
{ k := i; i := j; j := k; }
```

Ein Prozeduraufruf wird durch folgende Form erreicht:

```
<name> ( <argumentliste> ) ;
```

Die optionale <argumentliste> ist eine durch Kommata getrennte Liste von Ausdrücken, deren Anzahl und Typen mit der Anzahl der Parameter und deren Typen in der Vereinbarung der Prozedur übereinstimmen muss. Zur Laufzeit werden die Argumentausdrücke von links nach rechts ausgewertet und dann die Prozedur aktiviert. Für Referenzparameter können nur Variablen (einfache Variable oder Feldvariable) in der Argumentliste stehen, während für Wertparameter beliebige Ausdrücke zulässig sind. Felder müssen als Referenzparameter übergeben werden. Nach Abarbeitung der Prozedur kehrt die Ausführung hinter den Prozeduraufruf zurück.

Beispiel:

```
swap(n, m);
sum(3 * x + 5, 9 - y / 2, z);
```

2.7 Ausdrücke

Ausdrücke dienen zur Berechnung von Werten. Die Werte sind entweder ganzzahlig oder Wahrheitswerte.

Zur Berechnung von ganzzahligen Werten stehen die vier Grundrechenarten mit den üblichen Präzedenzen und Assoziativitäten zur Verfügung. Ebenfalls verfügbar ist das unäre Minus; es bindet stärker als die Multiplikation.

Die sechs Vergleichsoperatoren sind `<`, `<=`, `>`, `>=`, `=` und `#` (ungleich). Sie vergleichen die Werte von zwei ganzzahligen Ausdrücken und liefern einen Wahrheitswert. Sie binden schwächer als die Addition. Es stehen keine Operatoren zur Kombination von Wahrheitswerten zur Verfügung.

An jeder Stelle eines Ausdrucks können runde Klammern benutzt werden, um die eingebauten Präzedenzen und Assoziativitäten außer Kraft zu setzen.

Die Operanden eines Ausdrucks sind andere Ausdrücke, und letztendlich Literale oder Variablen. Letztere sind entweder einfache Variable oder indizierte Feldvariable. Die Indizierung eines Feldes geschieht durch einen Ausdruck mit ganzzahligem Wert in eckigen Klammern hinter der Feldvariablen. Operationen mit einem Feld als ganzem sind nicht erlaubt (außer der Übergabe des ganzen Feldes als Referenz).

Zur Laufzeit werden bei allen Ausdrücken mit zwei Operanden zuerst die beiden Operanden in nicht definierter Reihenfolge ausgewertet; danach wird der Operator angewandt.

Beispiele:

```
1
x
3 + x * y
(3 + x) * y
5 * -a[n-2]
i < n
b - 2 # a + 3
```

2.8 Bibliotheksprozeduren

`printi(i: int)`	Gibt den Wert von i auf dem Textbildschirm aus.
`printc(i: int)`	Gibt das Zeichen mit dem ASCII-Code i auf dem Textbildschirm aus.
`readi(ref i: int)`	Liest eine ganze Zahl von der Tastatur ein und speichert sie in i. Die Eingabe erfolgt zeilenweise gepuffert mit Echo.
`readc(ref i: int)`	Liest ein Zeichen von der Tastatur ein und speichert seinen ASCII-Code in i. Die Eingabe erfolgt ungepuffert und ohne Echo.
`exit()`	Beendet das laufende Programm und kehrt nicht zum Aufrufer zurück.
`time(ref i: int)`	Gibt in i die seit dem Start des Programms vergangene Zeit in Sekunden zurück.
`clearAll(color: int)`	Löscht den Graphikbildschirm mit der Farbe color. Farben werden durch Angabe der R-, G- und B-Komponenten nach dem Muster 0x00RRGGBB gebildet. Es stehen also für jede Komponente die Werte 0..255 zur Verfügung.
`setPixel(x: int, y: int, color: int)`	Setzt den Pixel mit den Koordinaten x und y auf die Farbe color. Grenzen: 0 <= x < 640, 0 <= y < 480.
`drawLine(x1: int, y1: int, x2: int, y2: int, color: int)`	Zeichnet eine gerade Linie von (x1|y1) nach (x2|y2) mit der Farbe color. Grenzen wie bei setPixel.
`drawCircle(x0: int, y0: int, radius: int, color: int)`	Zeichnet einen Kreis um den Mittelpunkt (x0|y0) mit dem Radius radius und der Farbe color.

2.9 Programmbeispiel

Das folgende kleine SPL-Programm berechnet Lösungen zum wohlbekannten “Acht-Damen-Problem”.

```
// queens.spl -- the 8-queens problem

type A8 = array [8] of int;
type A15 = array [15] of int;

proc main() {
  var row: A8;
  var col: A8;
  var diag1: A15;
  var diag2: A15;
  var i: int;

  i := 0;
  while (i < 8) {
    row[i] := 0;
    col[i] := 0;
    i := i + 1;
  }
  i := 0;
  while (i < 15) {
    diag1[i] := 0;
    diag2[i] := 0;
    i := i + 1;
  }
  try(0, row, col, diag1, diag2);
}

proc try(c: int, ref row: A8, ref col: A8, ref diag1: A15, ref diag2: A15) {
  var r: int;

  if (c = 8) {
    printboard(col);
```

```
  } else {
    r := 0;
    while (r < 8) {
      if (row[r] = 0) {
        if (diag1[r + c] = 0) {
          if (diag2[r + 7 - c] = 0) {
            // update
            row[r] := 1;
            diag1[r + c] := 1;
            diag2[r + 7 - c] := 1;
            col[c] := r;
            // try
            try(c + 1, row, col, diag1, diag2);
            // downdate
            row[r] := 0;
            diag1[r + c] := 0;
            diag2[r + 7 - c] := 0;
          }
        }
      }
      r := r + 1;
    }
  }
}

proc printboard(ref col: A8) {
  var i: int;
  var j: int;

  i := 0;
  while (i < 8) {
    j := 0;
    while (j < 8) {
      printc(' ');
      if (col[i] = j) {
        printc('0');
      } else {
        printc('.');
      }
      j := j + 1;
    }
    printc('\n');
    i := i + 1;
  }
  printc('\n');
}
```

3 Referenzimplementierung und Grundgerüst

Auf Moodle stehen Ihnen eine Referenzimplementierung sowie verschiedene Tools für den Eco32 für Linux und macOS zur Verfügung. Dieses können Sie auf Ihren Rechner herunterladen und entweder

- wenn Sie einen Linux/macOS-Rechner haben, dort mit

  ```
  tar xvfz eco32tools.tgz
  ```

  entpacken, oder
- per sftp auf saturn.mni.thm.de in Ihr dortiges Home-Verzeichnis kopieren. Per ssh oder putty können Sie sich auf saturn einloggen und das Archiv wie oben erklärt entpacken. Zu testende SPL-Programme können Sie ebenfalls per sftp auf saturn hochladen.

Windowsnutzer können auch das Windows Subsystem for Linux nutzen. Mit den beiden enthaltenen Skripten `compile.sh` und `run.sh` können Sie SPL-Programme compilieren und das Ergebnis auf dem Simulator testen. Enthalten ist eine `ReadMe.txt`, die weitere Informationen zur Benutzung der Tools enthält.

Beispiel für ssh-Login mit Benutzer-ID user1234 von der Kommandozeile:

```
$ ssh -l user1234 saturn.mni.thm.de
user1234@saturn.mni.thm.de's password: StrengGeheim
saturn:~/spl$ tar xvfz eco32tools.tgz
saturn:~$ ls -l
insgesamt 14
-rwxr-xr-x+ 1 user1234 4552 3190 27. Mär 16:18 bigtest.spl
drwxr-xr-x+ 2 user1234 4552    7 27. Mär 16:14 bin
-rwxr-xr-x+ 1 user1234 4552  206 27. Mär 16:18 compile.sh
drwxr-xr-x+ 2 user1234 4552    6 27. Mär 16:14 lib
-rw--wx-wx+ 1 user1234 4552  432 27. Mär 16:29 README
-rwxr-xr-x+ 1 user1234 4552   50 27. Mär 16:22 run.sh
```

Der Simulator benötigt zwingend das Package `xterm`, welches Sie mit dem Package-Manager Ihrer Distribution installieren können. `xterm` benötigt wiederum eine Grafikausgabe. Falls Ihnen keine zur Verfügung steht (zum Beispiel weil Sie WSL nutzen), können Sie mit dem `xvfb-run` den Simulator dennoch nutzen. Weitere Informationen dazu finden Sie in der angesprochenen `ReadMe.txt`.

3.1 Grundgerüst

Für die Implementierung Ihres Compilers steht Ihnen (ebenfalls auf Moodle) ein Grundgerüst zur Verfügung. Es enthält notwendigen Bibliothekscode für die C- und Java-Implementierungen. Essentielle Informationen zum Grundgerüst und der Entwicklung Ihres Compilers sind in der enthaltenen `ReadMe.txt` zu finden.

4 Die lexikalische Analyse für SPL

Die lexikalische Analyse zerlegt den SPL-Quelltext in eine Sequenz von Tokens. Sie wird als eigenständige Compiler-Komponente („Scanner“) implementiert. Der Kern des Scanners ist ein Unterprogramm (Java: Methode *next_token*, C: Funktion *yylex*), das bei jedem Aufruf das nächste Token aus dem Quelltext bestimmt und als Resultat an den Aufrufer zurückliefert. Als Aufrufer agiert zunächst ein Testtreiber (*show_token*), der das gefundene Token zur Überprüfung der Korrektheit auf die Konsole ausgibt. Später übernimmt der Parser diese Rolle, der den Scanner immer dann aufruft, wenn er das nächste Token aus dem Eingabestrom benötigt.

4.1 Tokens

Die Attribute eines Tokens sind

1. **Token-Typ**, z.B. `INTLIT` (Ganzzahl-Literal)

   Die Token-Typen sind die Terminalsymbole in der dem Parser zugrunde liegenden kontextfreien Grammatik.
2. **Lexem**, z.B. `"0xFF"`

   Die Syntax der Lexeme muss auf Basis der informalen Definition in Abschnitt 2.2 durch reguläre Ausdrücke spezifiziert werden, z.B. für dezimale Ganzzahl-Literale: `[0-9]+`.
3. **Position**, z.B. Zeile 12, Spalte 3

   Die Position dient zur Anzeige der Fehlerstelle bei Syntax- oder Semantikfehlern.
4. **Token-Wert**, z.B. 255

   Ein Token-Wert wird nur bei Zahlen (`INTLIT`) und Bezeichnern (`IDENT`) benötigt. Das Lexem eines Bezeichners wird von der semantischen Analyse beim Symboltabelleneintrag benötigt, der Wert einer Zahl vom Codegenerator.

   Die meisten Tokens haben keinen Wert, weil der Token-Typ schon alle relevanten Informationen zum Token beinhaltet. Beispiel: Zum Token-Typ `WHILE` gibt es nur ein einziges Lexem, nämlich das Schlüsselwort „while“. Dieses wird natürlich regelmäßig mehrfach im Quelltext auftauchen. Die zugehörigen Tokens unterscheiden sich dann aber nur in der Position. Abgesehen von Fehlermeldungen, bei denen Anzeige der Tokenposition sinnvoll ist, reicht ansonsten für die Weiterverarbeitung der Token-Typ völlig aus.

Ein SPL-Token wird in Java implementiert als Objekt der Klasse *Symbol*:

```
package java_cup.runtime;

public class Symbol {
  public int sym;       /* Token-Typ     */
  public int left;      /* Zeilennummer  */
  public int right;     /* Spaltennummer */
  public Object value;  /* Wert          */

  /* Konstruktor für Tokens ohne Wert */
  public Symbol(int s, int l, int r) { ... }

  /* Konstruktor für Tokens mit Wert */
  public Symbol(int s, int l, int r, Object v) { ... }
}
```

Das Tokenattribut *sym* spezifiziert den Token-Typ, für dessen mehr als 30 Ausprägungen jeweils symbolische Konstanten definiert sind. Dazu gehören

1. SPL-Schlüsselwörter: ELSE, WHILE, REF, IF, OF, TYPE, PROC, ARRAY, VAR
2. SPL-Operatoren: LT (“less than“), NE (“Not equal“), ASGN (“assignment“), PLUS, SLASH, STAR, GT, LE, MINUS, GE, EQ
3. verschiedene Sorten Klammern: LPAREN, RPAREN, LBRACK, RBRACK, LCURL, RCURL
   L=“left“ (öffnende Klammer), R=“right“ (schließende Klammer)
   PAREN=“paranthesis“ (runde Klammern: `()`)
   BRACK=“(square) brackets“ (eckige Klammern: `[]`)
   CURL=“curly brackets“ (geschweifte Klammern: `{}`)

4. Bezeichner: IDENT
5. Sonstige: COLON (":"), SEMIC (";")
6. Das Eingabeende-Token EOF
7. Das Fehler-Token error

4.2 Scanner-Generator

Der Scanner wird mit Hilfe eines Scannergenerators (Java: jflex, C: flex) aus einer formalen Spezifikation erzeugt. Die Spezifikation besteht im Kern aus einer Liste von Paaren, die jeweils aus einem regulären Ausdruck (*REGEXP*) und einem Programmfragment (*AKTION*) bestehen.

Der Generator erzeugt aus seiner Eingabedatei den (Java/C)-Quelltext für den Scanner. Der generierte Scanner versucht bei jedem Aufruf, eine Anfangssequenz des restlichen Eingabestroms einem regulären Ausdruck zuzuordnen. Wenn dies gelingt, wird die dem Ausdruck zugeordnete Aktion ausgeführt.

4.2.1 Eingabedateiformat für *jflex*

Die Eingabedatei des Generators besteht aus drei Segmenten, die durch %%-Zeilen getrennt sind:

```
    Segment 1
%%
    Segment 2
%%
    Segment 3:
        REGEXP          AKTION
          .               .
          .               .
          .               .
        REGEXP          AKTION
```

1. Segment 1 ist Java-Code, der vom Generator ohne Änderung in den generierten Scanner kopiert wird. Hier gehören globale Definitionen hinein, z.B.

   ```
   package parse;
   import java_cup.runtime.*;
   ```

2. Segment 2 kann unterschiedliche Bestandteile enthalten

   (a) Anweisungen zur Konfiguration des Generators, z.B. Name und Sichtbarkeit der Scanner-Klasse, Unterstützung von Zeilen- und Spaltennummern, Schnittstellen-Kompatibilität mit cup-generiertem Parser, z.B.

   ```
   %class Scanner
   %public
   %line
   %column
   %cup
   %cupdebug
   ```

   (b) reguläre Ausdrücke zur Wiederverwendung in Segment 3, z.B.

   ```
   L           = [A-Za-z_]
   D           = [0-9]
   H           = [0-9A-Fa-f]
   ID          = {L}({L}|{D})*
   DECNUM      = {D}+
   HEXNUM      = 0x{H}+
   ```

   Benannte reguläre Ausdrücke zu definieren ist sinnvoll, wenn komplexe Ausdrücke mehrfach wiederverwendet werden können. **Für SPL ist dies nicht der Fall!**

   (c) Java-Code zur Ergänzung der Scanner-Klasse
   Dieser Code wird in Klammern eingeschlossen:

   ```
   % {
   ....
   % }
   ```

und vom Generator in die Definition der Scanner-Klasse eingefügt. Daduch kann man die Scanner-Klasse mit eigenen Methoden erweitern, die für die Aktionen im 3. Segment nützlich sind.

Beispiel: „Wrapper"-Methoden für die Symbol-Konstruktoren zur Einsparung der Angabe der Tokenposition in den Aktionen

```
% {
  private Symbol symbol(int type) {
    return new Symbol(type, yyline + 1, yycolumn + 1);
  }

  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline + 1, yycolumn + 1, value);
  }
% }
```

Ein Anwendungsbeispiel folgt weiter unten in 4.3.

3. Segment 3 ist der Kern der Token-Spezifikation und besteht aus beliebig vielen (*REGEXP*, *AKTION*)-Paaren.

4.3 Reguläre Ausdrücke und Aktionen

Für jeden Token-Typ gibt es (mindestens) ein (*REGEXP*, *AKTION*)-Paar. *REGEXP* ist ein regulärer Ausdruck, der die Token-Syntax spezifiziert. *AKTION* ist ein beliebiges Code-Fragment. Es implementiert die Reaktion des generierten Scanner für den Fall, dass im Quelltext ein zu *REGEXP* passendes Token gefunden wird. Die Aktion ist im Fall des SPL-Compilers durch die geplante Kooperation des Scanners mit dem SPL-Parser bestimmt: Der Parser ruft den Scanner auf und erwartet das nächste Token im Eingabestrom als Return-Wert. Daher muss die Aktion eine *return*-Anweisung enthalten, die das richtige Token zurückliefert, z.B. für das Schlüsselwort ELSE:

```
else   { return new Symbol(Sym.ELSE, yyline + 1, yycolumn + 1); }
```

oder bei Verwendung der oben im Beispiel für Segment 2 definierten *symbol*-Methoden:

```
else   { return symbol(Sym.ELSE); }
```

Für Kommentare und Whitespace-Zeichen muss ebenfalls die Syntax und die zugehörige Aktion angegeben werden. Ist die Aktion leer, ignoriert der Scanner diese Quelltextbestandteile, z.B.

```
[ \t\n]  { /* für SPACE, TAB und NEWLINE ist nichts zu tun */ }
```

Für den Fall fehlerhafter Tokens im SPL-Quelltext muss eine „Catchall"-Aktion angegeben werden. Dazu ist der reguläre Ausdruck „ . " nützlich, der zu jedem beliebigen Zeichen außer „`\n`" passt.

4.3.1 Konflikte

Ein Konflikt entsteht, wenn es verschiedene Zuordnungsmöglichkeiten gibt. Die Konfliktlösung erfolgt nach folgenden Regeln:

1. Wenn verschieden lange Zeichenfolgen passen, wird die längstmögliche Zeichenkette zugeordnet.
   Beispiel: In der Eingabe kommt als nächstes „`200+max;`". Die Zeichenketten „2", „20" und „200" passen alle zum regulären Ausdruck für die Zahlen `[0-9]+` , längere Zeichensequenzen, z.B. „200+" oder „200+m" passen zu keinem einzigen regulären Ausdruck. Also ist hier das längstmögliche passende Lexem die Zahl „200".
2. Passt das längstmögliche Lexem zu mehreren regulären Ausdrücken, wird die erste Zuordnung gemäß der Anordnung in der Liste genommen.
   Beispiel: In der Eingabe kommt als nächstes „`while (x < max)`". Das längstmögliche zu einem regulären Ausdruck passende Lexem ist „while". Allerdings passt es sowohl zu dem regulären Ausdruck „while" für den Token-Typ WHILE, als auch zum regulären Ausdruck „ `[a-zA-Z_][a-zA-Z_0-9]*` " für den Token-Typ `IDENT`. Hier müssen die Schlüsselwörter also in der Liste zuerst kommen, damit sie korrekt zugeordnet werden.

4.3.2 Zahlenvarianten

Bei Ganzzahl-Literalen (`INTLIT`) gibt es mehrere syntaktische Varianten, z.B. die dezimale oder hexadezimale Schreibweise. Da die Bestimmung des Zahlenwerts aus dem Lexem variantenspezifisch ist, sollte für jede Variante ein eigenes Paar aus *REGEXP* und *AKTION* spezifiziert werden.

4.3.3 Schnittstelle für Token-Attribute

In den Aktionen stehen mehrere Variablen und Methoden zur Verfügung, die Informationen zum Token enthalten bzw. liefern:

- *yytext()* liefert das Lexem als *String*-Objekt, *yylength()* dessen Länge.
- Die Variablen *yyline* und *yycolumn* enthalten die Position, wobei die Zählung mit 0 beginnt.

4.3.4 Generator-Aufruf

Der jflex-Generator ist selbst ein Java-Programm und wird durch Maven bereitgestellt. Aufrufbeispiel:

```
mvn jflex:generate
```

Bei Erfolg findet man den generierten Scanner danach im Verzeichnis für generierte Quelldateien. Dort liegt er unter `jflex` als Datei `de.thm.mni.compilerbau.phases._01_scanner.Scanner.java`

4.4 Hinweise für die Scanner-Implementierung in C

4.4.1 Eingabedatei-Format

Eine typische *flex*-Eingabedatei hat folgenden Aufbau:

```
%{

    hier stehen im Scanner benutzte C-Deklarationen
    (z.B. include-Anweisungen, Typdefinitionen, Funktions-Header)

    Der Code wird vom Generator ohne Änderung in den Scanner-Quelltext
    kopiert.

%}

    hier können reguläre (Hilfs-)Ausdrücke definiert werden,
    entweder, weil sie mehrfach benötigt werden, oder zur
    Strukturierung besonders komplizierter Ausdrücke

    Format:       Name  Ausdruck
    Verwendung:   {Name}

%%

    hier steht eine beliebige Liste von regulären Ausdrücken
    und zugehörigen Aktionen. Eine Aktion ist eine beliebige
    C-Anweisung.

%%
    Dieses Segment wird vom Generator ohne Änderung in den Scanner-Quelltext
    kopiert. Hier sollten die Definitionen der in den Aktionen benutzten Funktionen
    stehen.
```

4.4.2 Parser-Schnittstelle

Die Schnittstelle zum einem von *yacc/bison* generierten Parser beteht im wesentlichen aus folgenden Konventionen:

- Der Scanner ist eine Funktion

  ```
  int yylex(void)
  ```

  Der Rückgabewert ist der Token-Typ. Dafür muss in den Aktionen durch entsprechende return-Anweisungen gesorgt werden.
- Der Wert eines Tokens muss innerhalb der entsprechenden Aktion in der globalen Variablen *yylval* gespeichert werden.
- In den Aktionen können zwei globale Variablen genutzt werden: *char *yytext* ist ein Verweis auf das Lexem, *int yyleng* die Länge des Lexems.

Das Token als logische Einheit wird also aus technischen Gründen bei der C-Implementierung aufgeteilt in den Token-Typ (Returnwert von yylex) und die anderen Attribute (globale Variable *yylval*).

4.4.2.1 Token-Typen

Die Token-Typ-Definitionen werden vom Parsergenerator aus der Spezifikation der kontextfreien Grammatik generiert und in der Datei „`parser.h`“ gespeichert:

```
enum yytokentype
{
  ARRAY = 258,
  ELSE = 259,
  IF = 260,
  OF = 261,
  PROC = 262,
  ...
}
```

Diese Datei wird im Scanner importiert.

4.4.2.2 Token-Werte

Die Token-Werte haben 3 Ausprägungen, die in `scanner.h` definiert sind: `IdentVal` für `IDENT`, `IntVal` für `INTLIT` und `NoVal` für alle anderen Token-Typen. Der C-Typ `YYSTYPE` für einen Token-Wert ist ein `union`-Typ mit den dazugehörigen Varianten `identVal`, `intVal` und `noVal`. Die Typdefinition dazu wird vom Parsergenerator in der Datei „`parser.tab.h`“ generiert. Details stehen in Abschnitt 6.4.1, S. 31 unter „Anmerkung für die C-Implementierung mit bison“.

Alle Token-Werte enthalten eine Position bestehend aus Zeilen- und Spaltennummer (`position`). Beim Erzeugen von Tokens müssen diese Werte daher ermittelt werden.

Der Generator bietet für Zeilennummern eingebaute Unterstützung (vgl. „`%option yylineno`“). Für Spaltennummern existiert eine solche Generatorunterstützung nicht. Hier bleibt also nur das manuelle Hochzählen als Lösung. Mit einigen Hilfsfunktionen kann dies vereinfacht werden.

Im ersten Segment der Eingabedatei wird ein Zähler als Variable definiert:

```
%{
    #include <stdio.h>
    #include   ...
       ...
    static Position currentPosition = {1, 1};
%}
```

Zum Inkrementieren des Zählers stehen zwei Funktionen zur Verfügung:

```
static void advanceColumnCounter(){
  currentPosition.column += yyleng;
}

static void advancePositionToNextLine(){
  currentPosition.line++;
  currentPosition.column = 1;
}
```

Diese Funktionen können nun in jeder Aktion im 2. Segment aufgerufen werden, zum Beispiel für Zeilenumbrüche:

```
\n          {advancePositionToNextLine();}
```

Analog zu den `symbol` Methoden in Java gibt es jedoch zusätzlich weitere Hilfsfunktionen, die `advanceColumnCounter` bereits aufrufen:

```
static int symbol(int type) {
    yylval.noVal.position = currentPosition;
    advanceColumnCounter();
    return type;
}

static int symbolIdentVal(int type, Identifier* value) {
```

```
    yylval.identVal.position = currentPosition;
    advanceColumnCounter();
    yylval.identVal.val = value;
    return type;
}

static int symbolIntVal(int type, int value) {
    yylval.intVal.position = currentPosition;
    advanceColumnCounter();
    yylval.intVal.val = value;
    return type;
}
```

Mit diesen Hilfsfunktionen sehen Aktionen so aus:

```
%%

array       {return symbol(ARRAY);}
else        {return symbol(ELSE);}

...
```

Man beachte, dass der Pointer auf das Lexem `yytext` in Aktionen nicht einfach kopiert werden darf. Der Speicherbereich auf den `yytext` zeigt wird vom Scanner wiederverwendet und ein erkanntes Lexem würde überschrieben werden. Werden Bezeichner erkannt (`IDENT`) muss daher eine Kopie angelegt werden um das Lexem dauerhaft verfügbar zu halten. Die Bibliotheksfunktion `newIdentifier` aus dem Grundgerüst erledigt dies bereits für Sie und kopiert den übergebenen String.

5 Syntaxanalyse für SPL

Die Syntaxanalyse konstruiert für das SPL-Quellprogramm eine Rechtsableitung nach dem LALR(1)-Bottum-Up-Verfahren. Sie wird als eigenständige Compiler-Komponente („Parser“) implementiert.

Der Parser wird mit einem Parsergenerator (Java: *cup*, C: *bison*) aus einer formalen Spezifikation generiert, deren Kern eine kontextfreie Grammatik für SPL ist.

5.1 SPL-Grammatik

Die Grammatik muss auf Basis der informalen Definition im Kapitel 2, S. 5 in der cup-Eingabedatei „`parser.cup`“ erstellt werden.

5.2 Spezifikation der Terminalsymbole

Terminalsymbole sind die Token-Typen, die der Scanner als Bestandteil der Tokens liefert. Terminalsymbole müssen mit „terminal“ definiert werden, wobei bei Terminalsymbolen mit Werten der Werte-Typ angegeben werden muss.

Beispiele:

```
terminal         ARRAY, ELSE, IF, OF, PROC, REF, TYPE, VAR, WHILE;
terminal String  IDENT;
terminal Integer INTLIT;
```

5.2.1 Operatoreigenschaften

Für den SPL-Compiler wird empfohlen, die Operatoreigenschaften *Präzedenz* und *Assoziativität* gemäß den Empfehlungen in den Folien und dem Compilerbau-Skript in den Ableitungsregeln zu verankern.

5.3 Spezifikation der Nonterminalsymbole

Nonterminalsymbole werden bei der Konstruktion der Grammatik nach Bedarf eingeführt. Die Bezeichnungen sollten möglichst selbsterklärend sein, als Schreibweise wird „CamelCase“ empfohlen.

Beispiel: *ProcedureDefinition* oder *ProcDef* steht offensichtlich für eine SPL-Prozedurdefinition, *WhileStmt* für „while statement“, *Program* für das gesamte Programm.

Schlechte Bezeichnungen:

- *X12345* – Ja, was ist das denn nun für ein Ding?
- *Pdef* – Parameter-Definition? Prozedurdefinition?
- *PROCDEF* – Großschreibung nur für Terminalsymbole
- *Prozedur* – Ist die Prozedurdefinition oder ein Prozeduraufruf gemeint?

5.3.1 Werte für Nonterminalsymbole

Jedem Grammatik-Symbol, ob Terminal oder Nonterminal, kann der Entwickler ein Attribut zuordnen. Dies ist eine dem Symbol zugeordnete Zusatzinformation, die gespeichert werden muss, weil sie zu einem späteren Zeitpunkt vom Compiler wieder benötigt wird. Die Grammatik wird dadurch zu einer attributierten Grammatik erweitert.

Für das Terminalsymbol `IDENT` enthält das Wert-Attribut den Namen des Bezeichners. Der Wert wird vom Scanner ermittelt und später von der semantischen Analyse für den Symboltabelleneintrag benutzt. Für das Terminal `INTLIT` ist das Attribut der Wert der Zahl. Auch hier wird der Wert vom Scanner ermittelt. Später wird der Codegenerator darauf zugreifen.

Für Nonterminalsymbole gibt es zunächst noch keine Attribute. Später wird die Grammatik aber mit semantischen Aktionen erweitert, die einen abstrakten Syntaxbaum als Compiler-interne Programmrepräsentation aufbauen. Dann werden die meisten Nonterminalsymbole als Wert einen Syntaxbaum zugeordnet bekommen (vgl. Kapitel 6).

Nonterminalsymbole müssen mit „non terminal“ definiert werden, wobei bei Nonterminalsymbolen mit Werten der Werte-Typ angegeben werden muss.

Beispiel für Nonterminalsymbole ohne Attribut:

```
non terminal GlobalDeclarations;
non terminal GlobalDeclaration;
```

Später wird einer Liste globaler Deklarationen ein abstrakter Syntaxbaum vom Typ *List<GlobalDeclaration>* (Deklarationsliste) und einer einzelnen globalen Deklaration ein Syntaxbaum vom Typ *GlobalDeclaration* (Deklaration) als Attribut zugeordnet:

```
non terminal List<GlobalDeclaration>  GlobalDeclarations;
non terminal GlobalDeclaration        GlobalDeclaration;
```

Ein Nonterminalsymbol wird als Startsymbol deklariert:

```
start with Program;
```

5.4 Ableitungsregeln

Eine Ableitungsregel hat die Form

```
<Nonterminal> ::= <rechte Seite> ;
```

Dabei können mehrere Regeln mit der gleichen linken Seite wie üblich notiert werden als

```
<Nonterminal> ::= <rechte Seite 1> |  <rechte Seite 2> | ...  ;
```

Beispiele:

```
Program ::= GlobalDeclarations;
WhileStatement ::= WHILE LPAREN Expression RPAREN Statement ;
IfStatement ::= IF LPAREN Expression RPAREN Statement |
                IF LPAREN Expression RPAREN Statement ELSE Statement ;
```

Hat die Regel die Form $X \to \varepsilon$, steht in der cup-Notation:

```
X  ::=  ;
```

oder besser:

```
X  ::= /* leer */ ;
```

5.5 *cup*-Eingabedateiformat am Beispiel

Die Eingabedatei „`parser.cup`“:

```
package de.thm.mni.compilerbau.phases._02_03_parser;

import java_cup.runtime.*;
import de.thm.mni.compilerbau.absyn.*;
import de.thm.mni.compilerbau.table.Identifier;

/* Definition der Terminalsymbole */
terminal         ARRAY, ELSE, IF, OF, PROC, REF, TYPE, VAR, WHILE;
terminal         LPAREN, RPAREN, LBRACK, RBRACK, LCURL, RCURL;
   ...
terminal String  IDENT;
terminal Integer INTLIT;

/* Definition der Nonterminalsymbole */
non terminal Program Program;
non terminal List<GlobalDeclaration> GlobalDeclarationList;
   ...
start with Program;

program ::= GlobalDeclarationList;
   ...
```

5.6 Generator-Aufruf und Erzeugung des Parsers

Der cup-Generator ist selbt ein Java-Programm und von Maven bereitgestellt. Aufrufbeispiel:

```
mvn cup:generate
```

Bei Erfolg findet man den generierten Parser danach im Verzeichnis für generierte Quelldateien. Dort liegt er unter `cup` als Datei `de.thm.mni.compilerbau.phases._02_03_parser.Parser.java`

Zum Compilieren des generierten Parsers benötigt man „java-cup-11b-runtime.jar“. Bei der Verwendung von Maven, wird diese Abhängigkeit automatisch heruntergeladen.

5.6.1 Debuggen des Parsers

Möchten Sie den Parser debuggen, sollten Sie nicht versuchen, den generierten Parser-Code zu debuggen, sondern die Optionen nutzen, die der Parser-Generator bereitstellt. Dazu müssen Sie `cup` mit den Optionen `-dump_states -debug` aufrufen. Wenn Sie `maven` nutzen, editieren Sie die Datei `pom.xml` und setzen Sie in dem Element

`<configuration>`

die Parameter

`<debug>true</debug>` und `<dumpStates>true</dumpStates>`.

In `Main.java` rufen Sie nun nicht `parse()` auf, sondern `debug_parse()`.

Hier ein Ausschnitt des dumps:

```
lalr_state [126]: { [$ START ::= program EOF (*) , EOF ] }
```

und der Debug-Ausgaben:

```
# Initializing parser
# Current Symbol is #23
# Shift under term #23 to state #5
# Current token is #32
# Shift under term #32 to state #19
```

5.7 Hinweise für die Parser-Implementierung in C

In C wird der Parsergenerator *bison* verwendet, der auf *yacc* beruht und mit diesem weitgehend kompatibel ist. Die Eingabedatei für den Generator heißt „`parser.y`“ und hat (typischerweise) folgende Bestandteile:

```
%{
    C-Deklarationen
%}

    bison-Deklarationen

%%
    Grammatik (ggf. mit semantischen Aktionen)
%%
    C-Definitionen
```

Beispiele für bison-Deklarationen:

```
/* YYSTYPE:  Attributtyp mit 4 Varianten */
%union {
  NoVal noVal;           /* nur Zeilennummer            */
  IntVal intVal;         /* Wert für INTLIT             */
  IdentVal identVal;     /* Name für IDENT              */

  /* Knotentypen für Nonterminale im Syntaxbaum */
  GlobalDeclaration *globalDeclaration;
  GlobalDeclarationList *globalDeclarationList;
    ...
```

```
}

/* Terminalsymbole mit Attributtyp-Variante */
%token      <noVal>         ARRAY WHILE PROC
%token      <identVal>      IDENT
%token      <intVal>        INTLIT

/* Nonterminalsymbole mit Attributtyp-Variante */
%type    <globalDeclarationList>    GlobalDeclarations
%type    <globalDeclaration>        GlobalDeclaration

/* Startsymbol */
%start Program
```

Beispiele für Grammatik-Regeln:

```
Program : GlobalDeclarations;
WhileStatement : WHILE LPAREN Expression RPAREN Statement ;
IfStatement : IF LPAREN Expression RPAREN Statement |
              IF LPAREN Expression RPAREN Statement ELSE Statement ;
```

5.7.1 Generator-Aufruf

Aufrufbeispiel:

```
bison -dtv parser.y
```

Bei Erfolg findet man danach folgende Dateien:

- `parser.output` – Beschreibung des Parsers als Grundlage für das Debuggen der Grammatik
- `parser.h` – Definitionen der Token-Typen und der Typen der semantischen Werte (`YYSTYPE`), wird auch vom Scanner benutzt!
- `parser.c` – Quelltext des Parsers

Um den Parser zu debuggen, setzen Sie in `main.c` den Wert der Variable `yydebug` auf 1:

```
yydebug=1;
```

6 Abstrakter Syntaxbaum und „Visitor“-Entwurfsmuster

6.1 Abstrakter Syntaxbaum: Rolle im Compiler und Aufbau

Der Abstrakte Syntaxbaum (AST – *Abstract Syntax Tree*) ist die Schnittstelle zwischen dem Frontend und dem Backend des SPL-Compilers. Ein AST repräsentiert ein SPL-Programm in einer für die Weiterverarbeitung durch das Backend geeigneten Form. Die Wurzel des AST verkörpert das SPL-Programm als Ganzes. Die inneren Knoten stellen komplexe Sprachkonstrukte wie Prozedurdefinitionen, While-Schleifen oder Wertzuweisungen dar. Dafür werden unterschiedliche Knotentypen definiert. Abhängig vom Knotentyp kann ein AST-Knoten einfache Attribute und komplexe Unterstrukturen haben, die dann als Nachfolgeknoten im AST erscheinen. Die vom Scanner gelieferten Tokens werden als Blattknoten in den AST integriert.

Im Gegensatz zum Ableitungsbaum („konkreter“ Syntaxbaum) sind syntaktische Elemente aus dem Quelltext, etwa Schlüsselwörter, Klammern und Semikolons, im AST nicht mehr vorhanden, wenn sie nur für die Syntaxerkennung selbst, nicht aber für die Weiterverarbeitung gebraucht werden.

Während der Syntaxanalyse eines SPL-Quelltextes muss der Parser den AST aufbauen. Die semantische Analyse benutzt den AST als Ausgangsbasis u.a. für die Konstruktion von Symboltabellen. Der Codegenerator benutzt sowohl den AST als auch die Symboltabellen für die Erzeugung von Assemblerbefehlen. Dabei wird jeweils der AST von der Wurzel ausgehend traversiert, wobei die Verarbeitung jedes Baumknotens von dessen Knotentyp abhängig ist.

6.2 Ein Beispiel

Der SPL-Compiler kann zu einem SPL-Programm eine textuelle Darstellung des AST erzeugen.

Beispiel: Der AST zum nachfolgenden SPL-Programm `test13.spl` kann mit dem Kommando

```
spl --absyn test13.spl test13.asm
```

ausgegeben werden.

`test13.spl:`

```
proc aux(ref i: int) {
  var j: int;
  j := i + 1;
}
proc main() {
  var i: int;
  aux(i);
}
```

Die textuelle Ausgabe des AST:

```
Program(
 ProcedureDeclaration(
   aux,
   Parameters(
     ParameterDeclaration(
       i,
       NamedTypeExpression(
         int),
       true)),
   Variables(
     VariableDeclaration(
       j,
       NamedTypeExpression(
         int))),
   Body(
     AssignStatement(
       NamedVariable(
         j),
       BinaryExpression(
         ADD,
```

```
        VariableExpression(
          NamedVariable(
            i)),
        IntLiteral(
          1))))),
ProcedureDeclaration(
  main,
  Parameters(),
  Variables(
    VariableDeclaration(
      i,
      NamedTypeExpression(
        int))),
  Body(
    CallStatement(
      aux,
      Arguments(
        VariableExpression(
          NamedVariable(
            i)))))))
```

Erläuterung einiger Knotentypen:

- `Program` repräsentiert das gesamte Programm.
- `ProcedureDeclaration` repräsentiert eine Prozedurdeklaration. Der Baumknoten hat ein Attribut vom Typ *Identifier*, das den Prodedurbezeichner „aux“ enthält, und drei komplexe Unterstrukturen, die durch Kindknoten repräsentiert werden:
  - Die Liste der Parameterdeklarationen: Ein `Parameters`-Knoten
  - Die Liste der Deklarationen lokaler Variablen: `Variables`-Knoten
  - Der Prozedur-Rumpf: Ein `Body`-Knoten, der die Sequenz von Anweisungen repräsentiert, aus denen der Rumpf besteht.

  Was es mit der Klasse *Identifier* auf sich hat, die nicht nur für Prodedurbezeichner, sondern für Bezeichner aller Art verwendet wird, ist weiter unten erläutert.
- `NamedVariable` und `ArrayAccess` sind Referenzen, also Speicherplatzrepräsentanten, die an bestimmten Stellen implizit dereferenziert werden. Im AST wird eine Dereferenzierung durch einen `VariableExpression`-Knoten dargestellt.

  Beispiel: `j := i;`

  Auf der linken Seite der Wertzuweisung steht der Variablenbezeichner *j* für den Speicherplatz. Auf der rechten Seite steht ebenfalls ein Variablenbezeichner (*i*), der aber nicht den Speicherplatz, sondern den dort abgespeicherten Wert repräsentiert. Die Dereferenzierung, die zum Speicherplatz den Wert liefert, erscheint im Baum als *VariableExpression*-Operation.

  ```
  AST:        AssignStatement(
               NamedVariable(j),
               VariableExpression(
                 NamedVariable(i)))
  ```

  Man beachte dabei, dass auf der Maschinenebene eine Referenz durch eine Speicheradresse repräsentiert wird. Die Dereferenzierung, also die Ermittlung des Werts erfolgt durch eine Maschinenoperation, nämlich einen lesenden Hauptspeicherzugriff, z.B. *ldw* („Load Word“-Instruktion)

  Ein `ArrayAccess`-Knoten steht für eine Feldkomponenten-Selektion, z.B. `meinfeld[i+1][10]` , ebenfalls eine Referenz, die z.B. auf der rechten Seite von Wertzuweisungen implizit dereferenziert wird.

Spezifikation des AST

Der AST ist im Praktikum vorgegeben, weil das AST-Design den Erfordernissen des Compiler-Backends Rechnung tragen muss und daher ohne die Erfahrung einer Backend-Implementierung nicht vernünftig definiert werden kann.

Das vorgegebene Compiler-Skelett enthält dazu das Paket *absyn*.

Die Bestimmung geeigneter Knotentypen für der AST hängt von den Anforderungen des Backends ab. Betrachtet man beispielsweise binäre Operationen (`+, -, *, /, =, <, >, ...`), so könnte man

a) für jede Operation einen eigenen Knotentyp definieren. Man bekommt dann viele Knotentypen, alle mit den gleichen Attributen: linker Operand und rechter Operand, beide vom Typ AST, oder

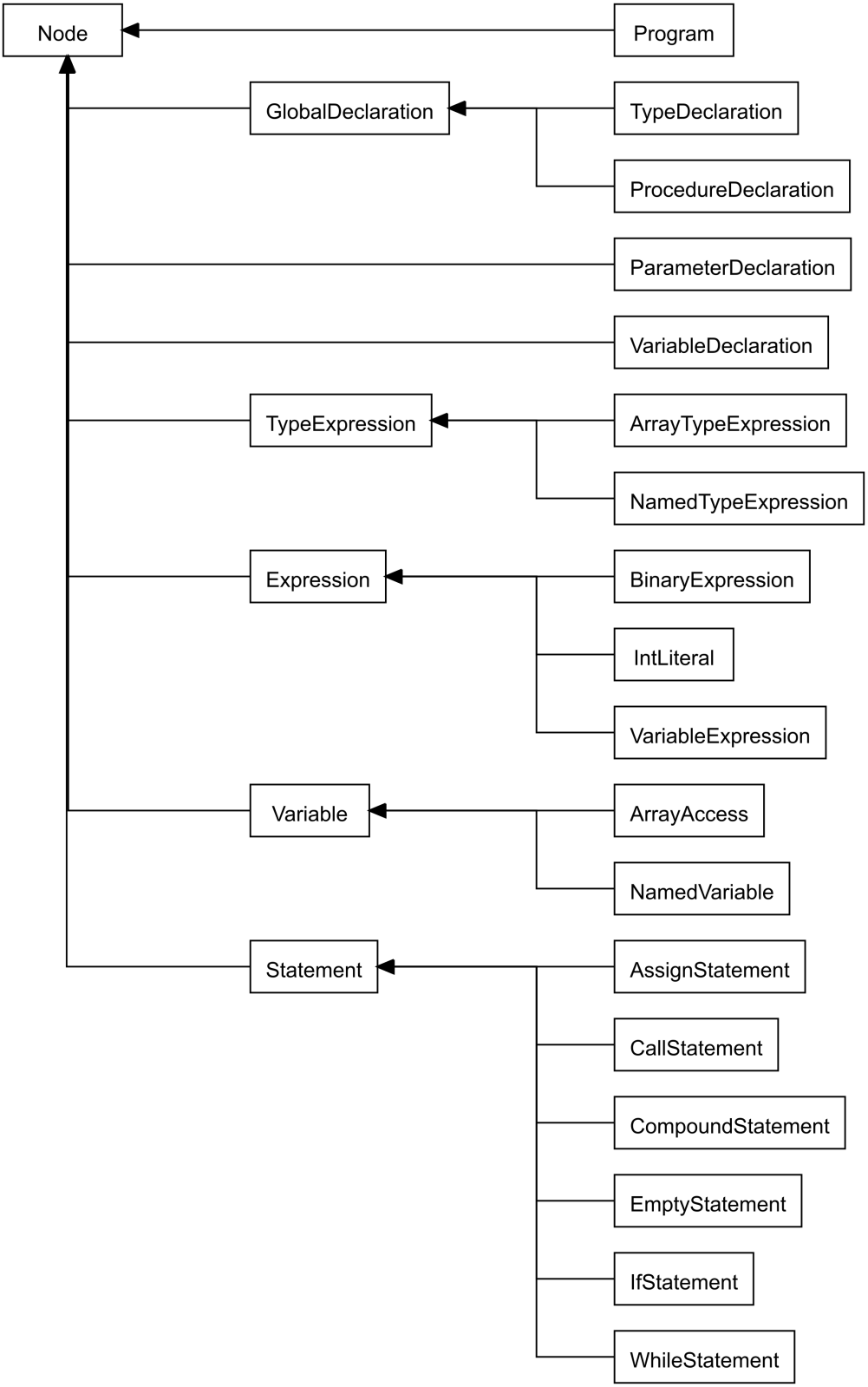
b) für alle Operationen denselben Knotentyp *BinärOperation* verwenden. Zu den beiden Operanden kommt dann ein weiteres einfaches Attribut *Operator* hinzu.

Für die Entwurfsentscheidung ist ausschlaggebend, ob es bei der Verarbeitung der Operationen mehr Gemeinsamkeiten oder mehr Unterschiede gibt. Sie werden am Ende des Praktikums feststellen, dass die Gemeinsamkeiten bei weitem überwiegen und dass daher die Entscheidung, mit nur einem Knotentyp *BinaryExpression* zu arbeiten, viel Redundanz vermeidet:

```
public class BinaryExpression extends Expression {
    public enum Operator {
        ADD, // +
        ...
        }

    public final Operator operator;
    public final Expression leftOperand;
    public final Expression rightOperand;
    ...
}
```

6.3 Übersicht über die AST-Klassen

Im Paket *absyn* sind eine ganze Reihe von Knotentypen für den AST in Form einer Klassenhierarchie definiert. Die Konstruktoren sind für den SPL-Parser wichtig, der den AST aufbaut.


Node
Program
GlobalDeclaration
TypeDeclaration
ProcedureDeclaration
ParameterDeclaration
VariableDeclaration
TypeExpression
ArrayTypeExpression
NamedTypeExpression
Expression
BinaryExpression
IntLiteral
VariableExpression
Variable
ArrayAccess
NamedVariable
Statement
AssignStatement
CallStatement
CompoundStatement
EmptyStatement
IfStatement
WhileStatement

Bezeichner, die Klasse *Identifier* und das „String-Interning“

Bei Betrachtung der Java-Klassen zum AST fällt auf, dass Bezeichner aus dem Quelltext im Baum nicht als *String*-Objekte auftauchen, sondern immer logisch eingebettet werden in Container-Objekte der Klasse *Identifier* aus dem Paket *table*. Beispielsweise ist der Typbezeichner, der in einer Typdeklaration neu definiert wird, im Baum als Attribut „name“ der Klasse „Identifier“ definiert:

```
public abstract class GlobalDeclaration extends Node {
    public final Identifier name;
    ...
}
```

Alle Deklarationsarten, die global auftreten könnne, erweitern die Klasse GlobalDeclaration.

Um aus einem *Identifier*-Objekt das eingebettete *String*-Objekt wieder zu extrahieren, wird die *Identifier*-Methode `toString()` verwendet.

Dahinter steckt eine optimierte Speicherung von Zeichenketten, die man „String-Interning“ nennt. Durch den Aufruf der *intern*-Methode wird eine kanonische Repräsentation des Strings erzeugt und in einem JVM-internen *String Constant Pool* abgespeichert (natürlich nur, wenn der String bzw. seine Repräsentation dort noch nicht vorhanden ist). Dadurch wird verhindert, dass gleiche Strings mehrfach gespeichert werden müssen. Zwei Strings sind dann gleich, wenn ihre Repräsentation gleich ist, d.h. sie an der gleichen Adresse im Pool abgespeichert sind. Diese Vergleich ist viel effizienter als ein zeichenweiser Vergleich.

```
package de.thm.mni.compilerbau.table;

/**
 * Represents an identifier in SPL.
 * Implements string interning internally to speed up lookups in symbol tables.
 */
public class Identifier {
    private final String identifier;

    public Identifier(String identifier) {
        //Intern the identifier.
        //This way string equality in table lookups can be determined by only comparing the references.
        this.identifier = identifier.intern();
    }

    public int hashCode() {
        return identifier.hashCode();
    }

    public boolean equals(Object other) {
        return other instanceof Identifier && ((Identifier) other).identifier.equals(identifier);
    }

    @Override
    public String toString() {
        return identifier;
    }
}
```

Anmerkung für die C-Implementierung:

In C werden Knotentypen durch einen `struct`-Typ repräsentiert, der die zugehörigen Attribute als Komponenten enthält. Eine binäre Operation (Java: Klasse *BinaryExpression*) sieht wie folgt aus (siehe `absyn.h`):

```
struct {
     binary_operator operator;
     struct expression *leftOperand, *rightOperand;
 } binaryExpression;
```

Die AST-Knotentypen

```
struct expression;
struct variable;
struct statement;
struct type_expression;
```

können aus mehreren verschiedenen Knotentypen bestehen. In Java ist dies mit abstrakten Klassen und Vererbung gelöst. In C wird stattdessen eine Struktur mit einer enthaltenen `union` verwendet wie z.B. :

```
typedef struct variable {
    int line;
    variable_kind kind;
    Type *dataType;
    union {
        struct {
            Identifier *name;
        } namedVariable;
        struct {
            struct variable *array;
            struct expression *index;
        } arrayAccess;
    } u;
} Variable;
```

Um unterscheiden zu können, um welche Art von Knoten es sich handelt, sind Aufzählungstypen deklariert, wie z.B.:

```
typedef enum {
    VARIABLE_NAMEDVARIABLE,
    VARIABLE_ARRAYACCESS
} variable_kind;
```

Das `kind` Attribut der jeweiligen Strukturen enthält jeweils einen Wert der zugehörigen Aufzählung. Darüber wird angegeben, welche Variante der `union` die aktuell gültige ist.

In seiner Struktur ist der abstrakte Syntaxbaum in C äquivalent zu der Implementierung mit Klassen in Java. Das Klassendiagram aus Abschnitt 6.3 hat also im übertragenen Sinne auch für C Gültigkeit.

Für das String-Interning sind der Konstruktor *Identifier *newIdentifier(char *string);* verfügbar. Den enthaltenen „rohen“ String (z.B. für eine Ausgabe) erhält man mit `name->string` (vgl. `identifier.h`).

6.4 Compiler Frontend: Konstruktion des AST

Beispiel für die Verwendung der Konstruktoren zum Baumaufbau

Der Parser soll nach Analyse eines korrekten SPL-Programms den zugehörigen AST als Resultat liefern. Für den Aufbau nutzt er die Konstruktoren der einzelnen Knotentypen. Ein Beispiel zum Baumaufbau „von Hand“:

```
type vector = array [10] of int;
```

Den AST dazu könnte man wie folgt aufbauen:

```
TypeDeclaration td = new TypeDeclaration(
                        new Position(row1, col1)
                        new Identifier("vector"),
                        new ArrayTypeExpression(
                            new Position(row2, col2),
                            new NamedTypeExpression(
                                new Position(row3, col3),
                                new Identifier("int")),
                            10));
```

- Die Verschachtelung der Konstruktoren spiegelt die hierarchische Struktur der Typdeklaration wieder. Das ganze ist ein Typeklaration (*TypeDeclaration*-Objekt). Die rechte Seite ist ein Typausdruck der Variante Array-Typausdruck (Objekt der Klasse *ArrayTypeExpression*). Zu einem Array-Typausdruck gehört immer als Untertyp der Komponententyp, im Beispiel ein Typname (Objekt der Klasse *NamedTypeExpression*).
- Zu jedem Knoten gehören eine Position bestehend aus Zeilen- und Spaltennummer (Attribute: row, col). Diese Attribute sind in der Klasse *Node* definiert und werden an alle Subklassen vererbt. Sie dienen im Fall eines Syntaxfehlers dazu, die Stelle des Fehlers in der Fehlermeldung ausgeben zu können, z.B. „Fehler in Typdeklaration in Zeile 10, Spalte 25.“

  Bei der Erzeugung neuer Knoten muss man diese Nummern angeben (oben z.B. `row1` und `col1`). Sie lassen sich aus den Tokens bestimmen, die der Scanner liefert. Wenn der Scanner das Token zum Schlüsselwort „type“ erzeugt, enthält dieses eine Zeilen und Spaltenangabe, die der Parser für die Typdeklaration übernehmen kann.

Anmerkung für die C-Implementierung:

Für den Aufbau der Bäume sind in `absyn.c` bzw. `absyn.h` eine Reihe von Konstruktoren definiert, die in der gleichen Weise verwendet werden, wie bei der Java-Implementierung, z.B.

```
GlobalDeclaration *newTypeDeclaration(int line, Identifier *name, TypeExpression *ty);
TypeExpression *newNamedTypeExpression(int line, Identifier *name);
```

6.4.1 Konstruktion des AST mit dem Parsergenerator

Ein Parsergenerator wie *cup* oder *bison* basiert auf dem Prinzip der *Syntaxorientierten Übersetzung*, das über die reine Syntaxprüfung hinaus gehende Berechnungen an die Ableitungsschritte gemäß der Grammatik koppelt. Dazu werden kontextfreie Grammatiken in zwei Aspekten erweitert:

1. Attributierte Grammatiken

Eine „Attributierte Grammatik“ ist eine Erweiterung, die es erlaubt, jedem Grammatiksymbol ein Attribut zuzuordnen. Dabei muss man den Typ des Attributs angeben. Das Attribut kann eine beliebige komplexe Datenstruktur sein. Intern arbeitet der Generator mit dem Attributtyp *Object*.

Beispiel 1: Für die Tokenkategorie INTLIT wird der Zahlenwert benötigt. Es wird ein Attribut vom Typ Integer definiert. In der cup-Eingabedatei wird dies wie folgt spezifiziert:

```
terminal Integer INTLIT;
```

Beispiel 2: Für die Tokenkategorie IDENT wird der Name des Bezeichners benötigt. Es wird ein Attribut vom Typ Identifier definiert. In der cup-Eingabedatei wird dies wie folgt spezifiziert:

```
terminal Identifier IDENT;
```

Beispiel 3: Für While-Schleifen wird ein abstrakter Syntaxbaum benötigt. Es wird ein Attribut vom Typ *Statement* definiert. In der cup-Eingabedatei wird dies wie folgt spezifiziert:

```
non terminal Statement while_statement;
```

Falls für ein Symbol kein Attribut benötigt wird, lässt man den Typ einfach weg:

```
terminal ARRAY, ELSE, IF, OF, PROC;
```

Anmerkung für die C-Implementierung mit *bison*:

Für den Compilergenerator bison gilt: Der Attributtyp ist `YYSTYPE`. Die obigen Beispiele sehen für *bison* wie folgt aus:

```
scanner.h:
    typedef struct { int line; }            NoVal;     // sonstige Tokens
    typedef struct { int line; int val; }   IntVal;    // fuer INTLIT
    typedef struct { int line; char *val; } IdentVal;  // fuer IDENT

parser.y:
    %union {
      NoVal noVal;
      IntVal intVal;
      IdentVal identVal;

      Expression *expression;
      Variable *variable;
      Statement *statement;
      TypeExpression *typeExpression;
      GlobalDeclaration *globalDeclaration;
      VariableDeclaration *variableDeclaration;
      ParameterDeclaration *parameterDeclaration;

      StatementList *statementList;
      ExpressionList *expressionList;
      VariableDeclarationList *variableList;
      ParameterList *parameterList;
      GlobalDeclarationList *globalDeclarationList;
    }
    /* ACHTUNG: Daraus erzeugt bison YYSTYPE:  typedef union { ... } YYSTYPE; */

    %token      <noVal>         ARRAY ELSE IF OF PROC
    %token      <identVal>      IDENT
    %token      <intVal>        INTLIT

    %type       <statement>     while_statement
```

Die Bezeichner, die für `%token` und `%type` angegeben werden sind daher keine Typausdrücke, sondern der Name einer Variante der Union. Dass der Name der Unionvariante lediglich der klein geschriebene Name des Typen ist, ist lediglich eine Konvention und wird von bison nicht geprüft.

2. Semantische Aktionen

Die Ableitungsregeln der Grammatik können durch Code-Fragmente erweitert werden, die wir als semantische Aktionen bezeichnen. Wir benötigen nur Aktionen, die am Ende der rechten Regelseite (vor dem Semikolon) eingefügt werden:

X ::= $X_1 \ldots X_n$ {: $Aktion$:} ;

Die Aktionen bestehen aus Java-Code mit speziellen Erweiterungen zum Zugriff auf die Attribute. Als Beispiel nehmen wir die Ableitungsregel für eine Multiplikation:

```
term ::= term STAR factor;
```

Die Multiplikation ist linksassoziativ und wird daher (siehe Skript) mit einer linksrekursiven Regel definiert. Ein AST für die Multiplikation ist ein Objekt der Klasse *BinaryExpression*. Der Konstruktor benötigt eine Position, die Codierung des Operators (hier `BinaryExpression.Operator.MUL`), und die Bäume für die Operanden.

Ein Baum für einen Ausdruck ist ein Objekt der Klasse *Expression*. Wir definieren also:

```
non terminal Expression term;
non terminal Expression factor;
```

Der Parser baut den Baum Bottom-Up auf, also zuerst die Teilbäume und danach den Wurzelknoten. Die semantische Aktion beschreibt also, wie man den Baum für die gesamte Multiplikation aufbaut, wobei man die Attribute der Unterbestandteile nutzen kann. In der Aktion steht das Schlüsselwort `RESULT` für das Attribut des Nonterminals auf der linken Regelseite. In unserem Beispiel ist `RESULT` der AST für die Multiplikation. Auf die Attribute der Symbole auf der rechten Regelseite kann man zugreifen, indem man innerhalb der Regel eindeutige Namen (Tags) einführt und die Schreibweise *Symbol*:*Tag* benutzt.

Beispiel:

Statt

```
term ::= term STAR factor;
```

schreibt man

```
term ::= term:lop STAR:opr factor:rop;
```

Die Tags sind: lop (linker Operand), opr (Operator) und rop (rechter Operand). In der Aktion repräsentieren die Tags jetzt die Attribute der entsprechenden Symbole. Also ist `lop` der AST zum linken Operanden, `opr` das Attribut des Multiplikationsoperators und `rop` der AST zum rechten Operanden.

Zu jedem Tag *xxx* ist außerdem das Tag *xxx*`left` und das Tag *xxx*`right` in der Aktion verwendbar. Diese stehen jeweils für die Zeilen- und Spaltennummer des jeweiligen Knotens.

Der Baumaufbau erfolgt in der semantischen Aktion:

```
term ::= term:lop STAR:op factor:rop {:
    RESULT = new BinaryExpression(new Position(opleft, opright),
                    BinaryExpression.Operator.MUL, lop, rop);
             :};
```

Ein anderes Beispiel: Statementlisten

```
non terminal List<Statement> statement_list;
non terminal Statement statement;

statement_list ::= statement:head statement_list:tail {:
                     RESULT = cons(head, tail); :}
                   | /*empty*/ {:
                     RESULT = nil(); :};
```

Die in diesem Beispiel verwendeten Methoden `cons` und `nil` sind im Parser definierte Hilfsmethoden zum Konstruieren von Listen.

6.4.2 Verarbeitung von Attributen und semantischen Aktion durch den Parsergenerator

Der vom Generator erzeugte Parser wird die Attribute in der Regel unabhängig vom Attributtyp handhaben. Dazu wird Parser-intern für alle Attribute derselbe Attributtyp *Object* verwendet. Ein SHIFT/REDUCE-Parser berechnet eine Ableitung rückwärts und bringt dabei zuerst die rechte Seite einer Ableitungsregel auf seinen Parser-Stack, um sie dann in einer REDUCE-Aktion durch die linke Regelseite zu ersetzen.

Als Beispiel betrachten wir noch einmal die Multiplikation:

```
term ::= term STAR factor;
```

Die REDUCE-Aktion findet statt, wenn alle Bestandteile der rechten Regelseite ganz oben auf dem Parser-Stack stehen, d.h. wenn der Parser den linken Operanden, den Operator und den rechten Operanden komplett gelesen hat.

```
Stack:      vor REDUCE                          nach REDUCE
                .                                   .
                .                                   .
                .                                   .
              term                  TOP ---->     term
              STAR
TOP ---->     factor
```

Um Attribute zu unterstützen, wird der Parser auf seinem Stack zu jedem Grammatik-Symbol X den dazu gehörenden Attributwert $attr_X$ mit auf dem Stack abspeichern. Auf dem Stack stehen also Paare der Form $(X, attr_X)$, im Beispiel:

```
Stack:      vor REDUCE                          nach REDUCE
                .                                   .
                .                                   .
                .                                   .
            (term,attr1)            TOP ---->   (term, attr4)
            (STAR,attr2)
TOP ---->   (factor,attr3)
```

Die semantischen Aktionen werden immer bei der REDUCE-Aktion ausgeführt. Da vor der Reduktion die rechte Regelseite mitsamt den Attributen oben auf dem Stack steht, können in der Aktion diese Attribute (attr1,attr2,attr3) genutzt werden, um das neue Attribut für die linke Regelseite (attr4) zu berechnen. Dieses wird zwischengespeichert, bevor die Symbole der rechten Regelseite vom Stack entfernt werden. Danach wird das Symbol auf der linke Regelseite mit seinem zwischengespeicherten Attribut auf dem Stack abgelegt.

Betrachten wir nun die semantischen Aktionen an unserem Beispiel:

```
term ::= term:lop STAR:op factor:rop {:
    RESULT = new BinaryExpression(new Position(opleft, opright),
                    BinaryExpression.Operator.MUL, lop, rop);
             :};
```

Das Attribut des Multiplikations-Terms `RESULT` entspricht in der obigen Stack-Darstellung dem Attribut `attr4`. Zur Berechnung werden `attr1 (lop)`, `attr2 (op)` und `attr3 (rop)` verwendet. Der Generator kann aus der Länge der rechten Regelseite den Speicherort der Attribute zu den rechts auftretenden Symbolen relativ zum TOP-Zeiger leicht bestimmen und im generierten Code entsprechende Stack-Zugriffe generieren. Da die Attributwerte vom Typ *Object* sind, muss im generierten Java-Code noch ein Cast auf den korrekten Attributtyp stehen. Dieser kann anhand der Typangabe für das Symbol erzeugt werden.

Beispiel: Der Attributwert für den rechten Operanden (Symbol: factor, Wert: rop) steht ganz oben auf dem Stack, weil *factor* das letzte Symbol der rechten Regelseite ist. Aus der Deklaration

```
non terminal Expression factor;
```

kann der Generator für den Attributwert einen Cast von *Object* nach *Expression* erzeugen. Durch die einmalige Typangabe für das Attribut erspart man sich also an allen Verwendungsstellen des Symbols das Casten des Attributwerts.

Die Zeilen- und Spaltennummern sind nichts anderes als zusätzliche, automatisch verwaltete Komponenten der Attribute.

Fazit: Der Parsergenerator kopiert im Prinzip die semantischen Aktionen aus seiner Input-Datei in den generierten Parser-Quelltext. Dabei muss er aber alle Zugriffe auf die Attribute im Hinblick auf Speicherort-Bestimmung und Downcast auf die richtige Absyn-Subklasse verarbeiten, um korrekten Java-Code zu erzeugen. Die Ausführung einer semantischen Aktion zu einer Regel R erfolgt immer bei der Reduktion der Regel.

Anmerkung für die C-Implementierung mit *bison*:

Für den Compilergenerator bison gilt im Prinzip das Gleiche, nur die Notation ist etwas anders. Unser Multiplikations-Beispiel für bison:

```
term STAR factor {$$ = newBinaryExpression($2.line, ABSYN_OP_MUL, $1, $3);}
```

Statt `RESULT` verwendet man `$$` und das Attribut für das i-te Symbol auf der rechten Regelseite ist `$`i.

6.5 Für das Backend: Verarbeitung des AST gemäß Visitor-Entwurfsmuster

Im Compiler-Backend werden mehrere Algorithmen benötigt, die alle den AST von der Wurzel ausgehend rekursiv bearbeiten, z.B. für die Ausgabe des Baums zu Testzwecken, die semantische Analyse oder die Assemblercode-Erzeugung. Gemeinsam ist diesen Algorithmen, dass die Verarbeitung eines Knotens von dessen Typ abhängt. Da der Knotentyp in einer OO-Sprache wie Java durch eine AST-Subklasse implementiert wird, liegt es nahe, jede beim Durchlaufen des Baums benötigte Knotentyp-spezifischen Aktion als Methode der entsprechenden AST-Subklasse zu definieren. Wenn beispielsweise die Erzeugung von Assemblercode für einen AST-Knoten durch eine Methode *codegen* umgesetzt werden soll, steht die Code-Erzeugung für While-Schleifen als Methode *codegen* in der Klasse *WhileStatement* und die Code-Erzeugung für Prozeduraufrufe als Methode *codegen* in der Klasse *CallStatement*.

Für den Entwickler des Algorithmus ist es allerdings äußert unpraktisch und unübersichtlich, wenn der Algorithmus auf 30 oder mehr Klassendefinitionen verteilt ist. Ein Beispiel mit 3 Algorithmen: *show* (Ausgabe des Baums), *codegen* (Code-Generator) und *typeCheck* (Typprüfung/semantische Analyse):

```
public abstract class Node {
  ...
  public abstract void show();
  public abstract Type typeCheck();
  public abstract void codegen();
  ...
}
```

Die Knotentyp-spezifischen Anteile der Algorithmen sind Methoden der AST-Subklassen:

```
class BinaryExpression extends Expression {
  public void show ()        { ... }
  public Type typeCheck ()   { ... }
  public void codegen ()     { ... }
  ...
}

class VariableExpression extends Expression {
  public void show ()        { ... }
  public Type typeCheck ()   { ... }
  public void codegen ()     { ... }
  ...
}

class IntLiteral extends Expression {
  public void prettyPrint () { ... }
  public Type typeCheck ()   { ... }
  public void codeGen ()     { ... }
  ...
}
  ...
```

Um das Dilemma der auf viele Klassen verteilten Algorithmen zu vermeiden, wird man nach einer Möglichkeit suchen, jeden Algorithmus zusammenhängend, d.h. innerhalb einer Klassendefinition, zu implementieren. Man könnte jeden der Baumverarbeitungs-Algorithmen beispielsweise in eine einzige Methode packen, in der eine explizit programmierte Fallunterscheidung nach Knotentyp erfolgt, z.B.:

```
public class AbsynPrettyPrinter {
  public void showNode (){

    // fuer jeden Knotentyp spezifische Ausgabe:

     if (node instanceof TypeDeclaration)  { ... }
    else if (node instanceof ProcedureDeclaration)  { ... }
    else if (node instanceof ParameterDeclaration)  { ... }
    else if (node instanceof VariableDeclaration)  { ... }
    else if (node instanceof NamedTypeExpression)  { ... }
    else if ...
      .
      .
      .
  }
}
```

Solche „Verteiler“ blähen nicht nur den Code auf, sie sind auch in der Ausführung sehr ineffizient. Daher wird stattdessen das Entwurfsmuster „Visitor“ verwendet. Dazu dient die abstrakte Klasse *Visitor*, die einen Baumverarbeitungs-Algorithmus repräsentiert. Ein konkreter Algorithmus, den den Baum durchläuft, wird als Subklasse von Visitor programmiert, z.B.

```
class AbsynPrettyPrinterVisitor extends Visitor { ... }
class CodeGeneratorVisitor extends Visitor { ... }
```

In dieser Visitor-Klasse stehen dann für alle Knotentypen die entsprechenden Methoden für die Code-Erzeugung. Konkret wird für jeden Knotentyp eine Methode „visit“ definiert, in der die spezifische Verarbeitung programmiert wird.

Baumausgabe:

```
class AbsynPrettyPrinterVisitor extends Visitor {
  void visit(TypeDeclaration t) { ... }
  void visit(ProcedureDeclaration p) { ... }
  void visit(ParameterDeclaration p) { ... }
     ...
```

Codererzeugung:

```
class CodeGeneratorVisitor extends Visitor {
  void visit(TypeDeclaration t) { ... }
  void visit(ProcedureDeclaration p) { ... }
  void visit(ParameterDeclaration p) { ... }
     ...
```

6.5.1 Überladungen, Polymorphismus und trickreiche Rekursion

Zur Erinnerung: Ein Methodenaufruf `obj.m(...)` in Java ist **überladen**, wenn es für *obj* mehrere Methoden mit dem gleichen Namen gibt, die sich anhand der Parameter-Signatur (Anzahl, Typen) unterscheiden. Die Unterscheidung erfolgt **zur Übersetzungszeit** durch den Java-Compiler!

Davon zu unterscheiden ist Polymorphismus. Am Beispiel:

```
class C { abstract void m(int i) {...} ... }
class C1 extends C { void m(int i) {...} ... } // C1 implementiert m
class C2 extends C { void m(int i) {...} ... } // C2 implementiert m auch

...
  C obj = null;
  if (...)
    obj = new C1(...);
  else
    obj = new C2(...);

  obj.m(0);   // Welches m? C1 oder C2? Polymorphismus
```

Java verlangt, dass zum Zeitpunkt des Methodenaufrufs *obj* inspiziert wird. Ist der aktuelle Wert ein Objekt der Subklasse *C1* muss auch die in *C1* definierte Methode *m* aktiviert werden. Dies gilt entsprechend auch für *C2*. Die Methodenzuordnung ist also eine Laufzeitaktion.

Eine „visit“-Methode für einen AST-Knoten ruft bei einer Rekursion diejenige „visit“-Methode für einen Kindknoten auf, die für dessen Knotentyp zuständig ist. Aus technischen Gründen erfolgt der rekursive Aufruf nicht direkt, sondern über den Umweg des Aufrufs einer „accept“-Methode, die zum Knotentyp gehört.

Folgendes Beispiel (Baumausgabe für ein ganzen Programm) zeigt, dass ein direkt rekursiver Aufruf von *visit* nicht ohne weiteres möglich ist:

```
class AbsynPrettyPrinterVisitor extends Visitor {
  void visit(Program program) {
     // Ausgabe der Liste -> jedes Element der Liste ausgeben
     for (GlobalDeclaration dec: program.declarations)
        visit(dec);  // FEHLER: visit-Methode hängt vom konkreten Typ von dec ab!
  }
  ...
```

Das Problem ist, dass für jeden Algorithmus und jeden Knotentyp eine *visit*-Methode existiert. Bei n Algorithmen und m Knotentypen gibt es also $n * m$ verschiedene *visit*-Methoden. Um den Aufruf `visit(dec)` also eindeutig zuordnen zu können, ist die Visitor-Subklasse (Klasse von *this*) und die *Absyn*-Subklasse von *dec* nötig. Während im Beispiel-Aufruf die Visitor-Subklasse bekannt ist (*AbsynPrettyPrinterVisitor*), kann der Typ von *dec* vom Java-Compiler nicht näher bestimmt werden. Erst zur Laufzeit kann man feststellen, ob es sich um *TypeDeclaration* oder eine *ProcedureDeclaration* handelt, welche alle ihre eigenen Ausgabemethode haben. Java verlangt aber, dass die Auflösung von Überladungen, also die eindeutige Zuordnung der aufzurufenden Methode anhand der Parametertypen zur Übersetzungszeit möglich sein muss.

Daher behilft man sich mit einem indirekten rekursiven Aufruf:

```
class AbsynPrettyPrinterVisitor extends Visitor {
  void visit(Program program) {
     // Ausgabe der Liste -> jedes Element der Liste ausgeben
     for (GlobalDeclaration dec: program.declarations)
        dec.accept(this);
  }
  ...
```

Da *dec* jetzt nicht mehr Parameter ist, sondern das Objekt, dessen *accept*-Methode bestimmt werden muss, handelt es sich nicht um eine Überladung, sondern um Polymorphismus, dessen Auflösung zur Laufzeit von Java unterstützt wird. Das Objekt *dec* enthält seinen Knotentyp, z.B. *TypeDeclaration*. Zum Zeitpunkt des Methodenaufrufs wird das Objekt von der Java VM inspiziert und anhand des Typs die *accept*-Methode der Klasse *TypeDeclaration* aufgerufen.

```
public class TypeDeclaration extends GlobalDeclaration {
     ...
    public void accept(Visitor v){
        v.visit(this);
    }
}
```

Dort erfolgt wiederum ein *visit*-Aufruf: `v.visit(this)` Wie wird hier die zuständige *visit*-Methode ermittelt? Der Typ des Arguments *this* ist zur Übersetzungszeit bekannt (*TypeDeclaration*), der Typ von *v* kann zur Laufzeit ermittelt werden, da es sich hier um Polymorphismus bzgl. der Visitor-Klasse handelt.

Der Ablauf komplett:

1. Beim Aufruf `dec.accept(this)` ist der Typ des Arguments zur Übersetzungszeit bekannt: *AbsynPrettyPrinterVisitor*. Der Typ von *dec* wird durch Inspektion des Objekts zur Laufzeit ermittelt und die zuständige accept-Methode von *TypeDeclaration* aufgerufen.
2. Beim Aufruf `v.visit(this)` ist der Typ des Arguments zur Übersetzungszeit bekannt: *TypeDeclaration*. Der Typ von *v* wird durch Inspektion des Objekts zur Laufzeit ermittelt und die für *TypeDeclaration* zuständige visit-Methode von *AbsynPrettyPrinterVisitor* aufgerufen.

Anmerkung für die C-Implementierung:

Die ganze Diskussion zum Visitor-Entwurfsmuster ist für nicht OO-Sprachen irrelevant. Die Auswahl der zur Knotenklasse passenden spezifischen Verarbeitung, die bei Java automatisch im Rahmen der Polymorphismus-Auflösung erfolgt, muss in C von Hand codiert werden. Dem Programmierer bleibt nichts anderes übrig, als bei **jeder** Knotenverarbeitung eine Fallunterscheidung über alle möglichen Knotentypen explizit zu programmieren. Beispiel für Statements:

```
static void showStatement(int indentation, Statement *statement) {
    if (statement == NULL) {
        error("Statement is NULL!");
        return;
    }

    switch (statement->kind) {
        case STATEMENT_ASSIGNSTATEMENT:
            showAssignStatement(indentation, statement);
            break;
        case STATEMENT_CALLSTATEMENT:
            showCallStatement(indentation, statement);
            break;
             ...
```

```
        default:
            errorUndefinedKind(statement->kind, "Statement");
    }
}
```

7 Typen, Symboltabellen und Semantische Analyse für SPL

7.1 Typen in SPL

SPL hat zwei primitive Typen: *Boolean* und *Integer*. Es gibt nur einen Typkonstruktor, mit dem komplexe Typen gebaut werden können: *Array*. Der zu einem Array-Typ gehörende Komponententyp kann selbst wieder ein Array-Typ sein, so dass beliebig verschachtelte Typhierarchien möglich sind. Als interne Repräsentation dieser Hierarchien werden Typ-Bäume verwendet.

In Java ist ein Typ-Baum ein Objekt der abstrakten Klasse *Type*. Diese Klasse hat zwei Subklassen: *PrimitiveType* und *ArrayType*. Ein primitiver Typ wird intern als Objekt der Klasse *PrimitiveType* dargestellt und ein Array-Typ als Objekt der Klasse *ArrayType*:

```
public class ArrayType extends Type {
    public final Type baseType;
    public final int arraySize;

    public ArrayType(Type baseType, int arraySize) {
        super(arraySize * baseType.byteSize);
        this.baseType = baseType;
        this.arraySize = arraySize;
    }

    public String toString() {
        return String.format("array [%d] of %s", arraySize, baseType);
    }
}
```

In der Klasse *PrimitiveType* gibt es nur zwei Objekte, je eines für *Boolean* und *Integer*.

```
public static final PrimitiveType intType = new PrimitiveType(4, "int");
public static final PrimitiveType boolType = new PrimitiveType(4, "boolean");
```

7.1.1 Gleichheit von Typen

Durch eine Typdefinition

```
type <IDENT> = <TYPAUSDRUCK> ;
```

wird kein neuer Typ erzeugt. Der neue Typbezeichner hat denselben Typ, wie der Typausdruck auf der rechten Seite. Beispiel:

```
type zahl = int ;
```

Der Typbezeichner „zahl“ kann nun genau wie „int“ für den vordefinierten Integer-Typ verwendet werden. Eine Variable vom Typ „zahl“ hat denselben Typ, wie eine Variable vom Typ „int“. Ein Typbezeichner kann insofern als ein Verweis auf einen vorhandenen Typ angesehen werden. Für die Prüfung der Typgleichheit müssen solche Verweise bzw. Verweisketten aufgelöst werden.

Jeder Array-Typausdruck definiert einen neuen Array-Typ. Auch wenn zwei Array-Typausdrücke genauso aussehen, sind die Typen, die sie repräsentieren, verschieden. Beispiel:

```
type vector  = array [10] of int;
type vector2 = vector;

var feld1 : array [10] of int;
var feld2 : array [10] of int;
var feld3 : vector;
var feld4 : vector;
var feld5 : vector2;
```

Die beiden Typausdrücke für *feld1* und *feld2* gleichen sich zwar, die Typen werden aber als verschieden betrachtet. Die Variablen „feld3“, „feld4“ und „feld5“ haben dagegen alle denselben Typ, da die Typdefinitionen von „vector“ und „vector2“ keine neuen Typen einführen und somit alle drei Variablen auf denselben Array-Typ zurückführbar sind.

In der interne Typrepräsentation wird daher für jeden Array-Typausdruck ein neuer Typ-Baum aufgebaut. Im Beispiel oben gibt es zu den drei im Quelltext vorkommenden Array-Typausdrücken also drei verschiedene Typ-Bäume, die allerdings alle das gleiche Aussehen haben:

```
new ArrayType(10, intType);
```

Für die semantische Analyse gilt also im Hinblick auf die Typen:

- Zwei Typen sind genau dann gleich, wenn sie durch dasselbe *Type*-Objekt repräsentiert werden.
- Zu jedem *ArrayTypeExpression*-Objekt im AST muss ein neues *ArrayType*-Objekt als Typrepräsentation gebaut werden. Da dazu die Typrepräsentation des Basistyps (Typ der Array-Komponenten) benötigt wird, ist der Algorithmus rekursiv.
- Zu jedem *NamedType*-Objekt im AST muss das zugehörige *Type*-Objekt in der Symboltabelle stehen. Es kann sich dabei um ein *PrimitiveType*-Objekt handeln (bei „int“, „zahl“) oder um ein *ArrayType*-Objekt (bei „vector“ und „vector2“).

7.1.2 Prozedur-Signaturen und Parametertypen

Zu jedem Parameter einer Prozedur gibt es zwei für die semantische Analyse relevante Attribute, der Typ und das Parameterübergabeverfahren:

- Ein Referenzparameter (Schlüsselwort „ref“ im Quelltext) repräsentiert einen Speicherplatz der aufrufenden Prozedur, dessen Adresse an die aufgerufene Prozedur übergeben wird.
- Ein Wertparameter (kein „ref“ im Quelltext) ist eine lokale Variable der aufgerufenen Prozedur, die beim Aufruf mit einem vom Aufrufer gelieferten Wert initialisiert wird.

Im Paket *table* wird ein Parametertyp daher als ein Objekt der Klasse *ParameterType* definiert, wobei das Attribut *isReference* angibt, ob es sich um einen Referenzparameter handelt:

```
public class ParameterType {
    public final Type type;
    public final boolean isReference;
      ...
    public ParameterType(Type type, boolean isReference) {
        this.type = type;
        this.isReference = isReference;
    }
}
```

Die semantische Analyse muss bei Prozeduraufrufen prüfen, ob die an der Aufrufstelle (im AST *CallStatement*-Objekt) übergebenen Argumente passen:

- Ist die Anzahl der Argumente korrekt?
- Für jedes Argument:
  - Stimmt der Typ?
  - Falls der Typ ein Arraytyp ist: Ist der Parameter ein Referenzparameter?
  - Falls der Parameter ein Referenzparameter ist: Ist das Argument eine Variable?

Für diese Prüfungen wird die Schnittstelleninformation oder „Signatur“ der aufgerufenen Prozedur benötigt, die die *ParamType*-Informationen aller Parameter enthält:

```
public class ProcedureEntry extends Entry {
    ...
    public final List<ParameterType> parameterTypes;
    ...
}
```

7.2 Symboltabellen in SPL

In SPL gibt es für die Bezeichner eine flache Hierarchie von Gültigkeitsbereichen:

- Global definierte Bezeichner: Typen und Prozeduren

- Lokal innerhalb einer Prozedur definierte Bezeichner: Parameter und lokale Variablen

Eine Symboltabelle repräsentiert einen Gültigkeitsbereich und enthält Einträge für alle Bezeichner dieses Bereichs. Der SPL-Compiler benötigt also

- Eine globale Symboltabelle und
- für jede Prozedur eine lokale Symboltabelle.

In der globalen Tabelle stehen neben den im SPL-Programm definierten Typ- und Prozedurbezeichnern auch die vordefinierten Bezeicher („int“, „readc“, „printi“ usw.).

Im Paket *tables* werden Symboltabellen so definiert, dass eine mehrstufige Hierarchie aus ineinander verschachtelten Gültigkeitsbereichen unterstützt wird. Eine Tabelle ist ein *Table*-Objekt und kann mit einer übergeordneten Tabelle durch einen *upperLevel*-Verweis verknüpft werden. Für die Prozedur-spezifischen lokalen Tabellen enthält also *upperLevel* einen Verweis auf die globale Tabelle.

```
package de.thm.mni.compilerbau.table;
 ...
public class SymbolTable {
    private final Map<Identifier, Entry> entries = new HashMap<>();
    private final SymbolTable upperLevel;

    public SymbolTable() {
        this.upperLevel = null;
    }

    public SymbolTable(SymbolTable upperLevel) {
        this.upperLevel = upperLevel;
    }

    public Entry enter(...) { ... }
    public Entry lookup(Identifier name) { ... }
```

Die neben den Konstruktoren wichtigen Methoden sind *enter* zum Eintragen einer neuen Definition (*Entry*-Objekt) und *lookup* zum Suchen eines Eintrags zu einem Symbol.

7.2.1 Symboltabelleneinträge

Ein Symboltabelleneintrag assoziiert einen Bezeicher mit einer Menge von Attributen. Dazu gehören Informationen, die unmittelbar aus der Definitionsstelle im Quelltext entnommen werden können, wie z.B. die Art des Bezeichners (Typ, Prozedur, ...) oder das Parameter-Attribut *isReference*. Andere Attribute werden im Rahmen der semantischen Analyse bestimmt, z.B. zu einem Variablenbezeichner der Typ. Weitere Compiler-Komponenten nutzen die Symboltabelle ebenfalls, um dort Bezeichner-spezifische Informationen abzuspeichern, z.B. Adressen von Variablen.

Natürlich ist auch die Art des Eintrags selbst ein Attribut des Bezeichners. Dies wird in Java und C allerdings unterschiedlich gehandhabt:

- In C ist die Eintragsart ein normales Attribut *kind*. Hier wird das gleiche Muster verwendet wie bereits bei den Strukturen des AST mit mehreren Varianten, die über das *kind*-Attribut unterschieden werden.
- In Java ist ein Eintrag ein Objekt der abstrakten Klasse *Entry*. Die Art des Eintrags ergibt sich aus der *Entry*-Subklasse *VariableEntry*, *TypeEntry* oder *ProcedureEntry*.

Die Attribute eines Bezeichners hängen von der Art des Bezeichers ab:

1. Zu einem Typbezeichner gehört ein Typ.
2. Zu einem Variablenbezeichner gehört ein Typ. (Später wird noch eine Speicheradresse *offset* dazu kommen.)
3. Zu einem Parameterbezeichner gehört ein Typ und ein *isReference*-Attribut. (Später wird noch eine Speicheradresse *offset* dazu kommen.)
4. Zu einem Prozedurbezeichner gehören eine Parametertypliste und eine lokale Symboltabelle. Für die Code-Erzeugung werden dann noch weitere Attribute benötigt, z.B. die Anzahl der Bytes, die für die Speicherung der lokalen Variablen nötig sind (int *localVarAreaSize*).

Warum gibt es für die Parameter keine Klasse *ParameterEntry*? Da Symboltabelleneinträge für Variablen und Parameter weitgehend gleich genutzt werden, gibt es für beide Bezeichnerarten nur eine Art von Symboltabelleneinträgen, repräsentiert durch ein *VariableEntry*-Objekt. Eine lokale Variable wird dort genau so eingetragen, wie ein Wertparameter. Da es in SPL keine Referenzvariablen gibt ist für Variablen immer `isReference = false` .

Variableneinträge:

```
public class VariableEntry extends Entry {
  public final Type type;
  public final boolean isReference;
  public int offset; // This value has to be set in phase 5

  public VariableEntry(Type type, boolean isReference) {
      this.type = type;
      this.isReference = isReference;
  }
  ...
}
```

Typeinträge:

```
public class TypeEntry extends Entry {
  public final Type type;

  public TypeEntry(Type type) {
      this.type = type;
  }
  ...
}
```

Prozedureinträge:

```
public class ProcedureEntry extends Entry {
  public final SymbolTable localTable;
  public final List<ParameterType> parameterTypes;
  public int argumentAreaSize, outgoingAreaSize, localVarAreaSize;
     // These values have to be set in phase 5

  public ProcedureEntry(SymbolTable localTable, List<ParameterType> parameterTypes){
      this.localTable = localTable;
      this.parameterTypes = parameterTypes;
  }
    ...
}
```

Einträge in C:

```
typedef struct {
  entry_kind kind;
  union {
      struct {
          Type *type;
      } typeEntry;
      struct {
          Type *type;
          bool isRef;
          int offset;        /* filled in by variable allocator */
      } varEntry;
      struct {
          ParameterTypeList *parameterTypes;
          struct table *localTable;
          int argumentArea;        /* filled in by variable allocator */
          int localvarArea;        /* filled in by variable allocator */
          int outgoingArea;        /* filled in by variable allocator */
      } procEntry;
  } u;
} Entry;
```

7.3 Semantische Analyse

Die Semantische Analyse muss alle Korrektheitsbedingungen prüfen, die nicht in der SPL-Grammatik stecken. Einige Beispiele:

- Sind beide Operanden einer Addition vom Typ *int*?
- Ist jeder Bezeichner auch passend zu seiner Verwendung definiert?
- Gibt es eine Definition der Prozedur „main“?
- Ist der Ausdruck auf der rechten Seite einer Wertzuweisung vom Typ *int*?

Ein strukturiertes Vorgehen orientiert sich an der Syntax: Für jeden Knotentyp des AST werden die notwendigen Prüfungen separat analysiert und implementiert.

7.3.1 Bezeichner und das „Declare before Use“-Prinzip

Wenn im AST Bezeichner auftreten, handelt es sich entweder

- um die Definitionsstelle des Bezeichners, z.B.

  ```
  var i: int;
  ```

  (im AST ein *VariableDeclaration*-Objekt) oder
- um eine Verwendungsstelle, z.B. die linke Seite folgender Wertzuweisung

  ```
  i := 1;
  ```

  (im AST ein *NamedVariable*-Objekt)

Semantische Analyse einer Definitionsstelle heißt

- Entry-Objekt bauen und
- in die richtige Symboltabelle mit der *enter*-Methode eintragen.

Semantische Analyse einer Verwendungsstelle erfordert:

- in der Symboltabelle nach dem Bezeichner suchen
- bei Fehlschlag eine passende Fehlermeldung liefern
- bei Erfolg prüfen, ob die Art des Bezeichners zur Verwendung passt
- mit den Attributen aus dem Eintrag sonstige Korrektheitsbedingungen prüfen

Das „Declare before Use“-Prinzip fordert, dass jeder Bezeichner zuerst (weiter oben im Quelltext) deklariert werden muss, bevor er benutzt werden darf. Diese Spracheigenschaft ermöglicht eine semantische Analyse in einem Durchgang, weil bei der Prüfung jeder Verwendungsstelle der betreffende Bezeichner in der Symboltabelle stehen muss.

In SPL gilt diese Prinzip allerdings **nicht** für Prozedurbezeichner: Der Aufruf einer Prozedur ist schon vor der Prozedurdefinition möglich. Dies erlaubt gegenseitig rekursive Aufrufe.

Deswegen wird die semantische Analyse in zwei Durchgänge aufgeteilt:

1. Aufbau der Symboltabellen ohne Prüfung der Anweisungen in den Prozedur-Rümpfen
   - Erzeugen der globalen Symboltabelle
   - Eintragen der vordefinierten Bezeichner
   - Durchlauf durch den AST und bearbeiten aller Definitionen
2. Prüfung der Anweisungen in den Prozedur-Rümpfen

7.3.2 Nutzung des Visitor-Pattern

Die semantische Analyse wird in zwei Durchgängen implementiert. Jeder der beiden Durchgänge der Analyse wird in Form einer Visitor-Klasse implementiert, der erste Durchgang als *class TableBuilder*, der zweite als *class ProcedureBodyChecker*. Das Grundgerüst ist in den beiden Paketen *_04a_tablebuild* und *_04b_semant* schon vorgegeben. Für die Initialisierung der globalen Tabelle mit den vordefinierten Bezeichnern steht die Klasse *TableInitializer* zur Verfügung.

Beide Durchgänge sollen das Entwurfsmuster „Visitor“ verwenden. Somit muss für jeden AST-Knotentyp eine passende visit-Methode bereitgestellt werden. Dabei stellen sich zwei Fragen:

1. Wie kann man bei rekursiven Aufrufen Parameter übergeben und Resultate erhalten, wenn die Signaturen von *visit* und *accept* dies nicht erlauben?

   Vorschlag: Definieren Sie für die Schnittstelle zwischen zwei Rekursionsebenen passende Attribute im Visitor-Objekt. Statt einer Parameterübergabe erfolgt eine Wertzuweisung an die entsprechenden Attribute vor dem rekursiven *visit*-Aufruf. Nach der Rückkehr aus dem rekursiven Aufruf findet die aufrufende Instanz eventuell vorhandene Resultatswerte ebenfalls in Instanz-Variablen des Visitor-Objekts.

   Solche Werte können alternativ auch direkt im AST gespeichert werden. Dies ist nur sinnvoll, wenn die Informationen von dauerhafter Relevanz sind. Im Beispiel unten wird für den von *visit* für *ArrayType* bestimmten Typ dieses Prinzip genutzt: Der errechnete Typ ist in *variableDeclaration.typeExpression.dataType* zu finden.

2. Welche Visitor-Objekte werden benötigt?

   Vorschlag: Entweder versuchen Sie, mit einem einzigen Visitor-Objekt auszukommen, oder Sie erzeugen für jede Rekursionsebene ein neues Visitor-Objekt. Im Beispiel unten wird nur ein Visitor-Objekt für alle AST-Knoten verwendet.

Beispiele:

`TableBuilder.java`:

```
        .
        .
        .
    public void visit(Program program) {
           program.declarations.forEach(dec -> dec.accept(this));
    }

    public void visit(VariableDeclaration variableDeclaration) {
           variableDeclaration.typeExpression.accept(this);

           table.enter(new VariableEntry(variableDeclaration.name,
             variableDeclaration.typeExpression.dataType, false), SplError.RedeclarationAsVariable(
               variableDeclaration.position, variableDeclaration.name));
    }

    public void visit(ArrayTypeExpression arrayTypeExpression) {
           arrayTypeExpression.baseType.accept(this);
           arrayTypeExpression.dataType = new ArrayType(
              arrayTypeExpression.baseType.dataType, arrayTypeExpression.arraySize);
     }
        .
        .
        .
```

DoNothingVisitor

Manche Algorithmen, die den AST durchlaufen, bearbeiten nur einen Teil der Knotentypen. So ignoriert beispielsweise der *TableBuilder* die Anweisungen und Ausdrücke aller Art. Man kann sich in diesem Fall die Programmierung leerer *visit*-Methoden ersparen, indem man eine Visitor-Klasse mit leeren Methoden für alle Knotentypen implementiert und diese dann bei Bedarf überschreibt. Die Klasse *DoNothingVisitor* ist dafür vorgegeben.

Einen Nachteil hat diese Vorgehensweise: Wenn man für einen Knotentyp die Implementierung vergisst, wird man vom Java-Compiler keinen Hinweis bekommen.

7.4 Hinweise für die Implementierung der semantischen Analyse in C

Die in diesem Kapitel beschriebenen Prinzipien gelten natürlich genauso auch für C. Die erwähnten Datenstrukturen (`SymbolTable`, `Entry`, `Type`, `ParameterType`) existieren in äquivalenter Form im vorgegebenen C-Code. Sie finden Sie in den Headern `table.h` sowie `types.h`. Die Implementierung der Datenstrukturen mit mehreren Varianten folgt dem gleichen Muster, das auch bereits für die Strukturen des AST zum Einsatz kam.

Lediglich die Implementierung der Analysephasen weicht aufgrund fehlender Vererbung von der Java-Implementierung mit dem Visitor-Muster ab. Statt eines Visitors werden Sie mehrere Funktionen programmieren, die jeweils Teile der jeweiligen Phase übernehmen. Beispielsweise werden Sie für den Tabellenaufbau eine Funktion benötigen, die den Typbaum für einen gegebenen Typausdruck ermittelt. Mit einem `switch`-Statement können Sie zwischen den verschiedenen Varianten unterscheiden.

8 Laufzeitstack-Organisation für SPL

8.1 Unterprogrammaufrufe in SPL

SPL hat ein einfaches Unterprogrammkonzept, das die Definition beliebig vieler Prozeduren auf der globalen Ebene erlaubt. Ineinander verschachtelte Prozedurdefinitionen sind nicht möglich. Da es in der Sprache auch keine globalen Variablen gibt, kann eine Prozedur ausschließlich auf ihre Parameter und ihre lokalen Variablen zugreifen.

Es gibt zwei Arten der Parameterübergabe: Wertübergabe („call by value“) und Referenzübergabe („call by reference“).

8.1.1 Begriffe

Im Zusammenhang mit Unterprogrammen (Prozeduren, Funktionen, Methoden) muss zwischen den Parametern unterschieden werden, die als Teil der Unterprogrammdeklaration definiert werden, und den Argumenten, die an der Aufrufstelle angegeben werden. Erstere werden manchmal als *formale Parameter* und letztere als *aktuelle Parameter* bezeichnet. Wir benutzen im folgenden stattdessen die Begriffe „Parameter“ (formale Parameter) und „Argumente“ (aktuelle Parameter).

Bei Unterprogrammaufrufen gibt es eine klare Rollenverteilung. Das aufrufende Unterprogramm bezeichnen wir im folgenden als *Caller*, das aufgerufene Unterprogramm als *Callee*.

8.1.2 Parameterübergabeverfahren

Die Deklaration der Parameter legt den *Typ* und das *Parameterübergabeverfahren* fest.

Beispiel:

```
proc modulo ( op1: int, op2: int, ref result: int) {
   result := op1 - op1 / op2 * op2;
}
```

Die Parameter von *modulo* sind *op1*, *op2* und *result*. Die Parametertypen sind alle `int`. Die ersten beiden Parameter sind *Wertparameter*: Ein Wertparameter ist eine lokale Variable des Callee, die vom Caller mit einem Initialwert versorgt wird. Als Argument kann an der Aufrufstelle ein beliebiger Ausdruck angegeben werden. Zum Aufrufzeitpunkt wird dessen Wert berechnet und im Parameter gespeichert. Eine Wertänderung von *op1* durch *modulo* würde nichts am Wert von *i* ändern.

Der Parameter *result* ist ein *Referenzparameter*: Ein Referenzparameter ist, vereinfacht gesagt, ein vom Callee benutzter Verweis auf eine Variable des Caller. Im Beispiel hat *main* eine lokale Variable *i* und übergibt beim Aufruf von *modulo* einen Verweis auf *i* als drittes Argument. Genauer betrachtet ist ein Referenzparameter eine lokale Variable der aufgerufenenen Prozedur, die ähnlich wie eine Zeigervariable (Pointer) genutzt wird. Sie wird beim Aufruf mit der Adresse der Argument-Variablen initialisiert.

Aufrufbeispiel:

```
proc main () {
   var i: int;
   i := 21;
   modulo(i, i-16, i);
   printi(i);
}
```

Die Argumente sind *i*, *i-16*, und *i*. Der erste Parameter von *modulo* (*op1*) wird mit 21 (Wert von *i*) initialisiert, der zweite (*op2*) mit 5. Da der dritte Parameter (*result*) ein Referenzparameter ist, wird nicht der Wert von *i*, sondern die Adresse von *i* an *modulo* übergeben. Dadurch kann *modulo* die lokale Variable *i* von *main* verändern: Im Beispielaufruf wird nach der Rückkehr *i* den Wert 1 (21 modulo 5) haben.

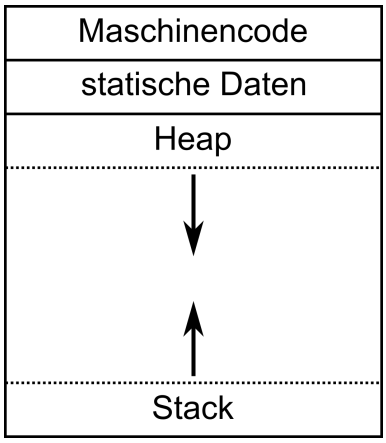
Den Bezug von Referenzparametern zu Pointervariablen verdeutlicht die entsprechende C-Implementierung. Da C keine Referenzparameter kennt, müssen stattdessen Pointer verwendet werden.

```
#include <stdio.h>

void modulo ( int op1, int op2, int *result) {
       *result = op1 - op1 / op2 * op2;
}

int main () {
       int i = 21;
       modulo(i, i-16, &i);
       printf("%d\n", i);
}
```

Im Gegensatz zu der SPL-Version muss in der C-Implementierung beim dritten Argument im Aufruf von *modulo* der explizite Adressoperator verwendet werden, um die Adresse von *i* zu übergeben. Auch innerhalb von *modulo* ist der Unterschied sichtbar: Um auf das *i* von *main* zugreifen zu können, muss in C der Dereferenzierungsoperator verwendet werden (`*result`), der beim Referenzparameter in der SPL-Variante weggelassen wird. Auf der Maschinenebene sind beide Lösungen vollkommen äquivalent, die Unterschiede liegen nur auf der Quelltextebene.

8.2 Hauptspeicher-Layout und Laufzeitstack

Der Hauptspeicher eines Prozesses ist (vereinfacht) wie folgt in logische Teilbereiche (Segmente) gegliedert:



Für SPL wird der Heap nicht benutzt.

Der Stack enthält für jeden aktiven Unterprogramm-Aufruf einen Eintrag, den Aktivierungsrahmen (auch „activation record“ oder „frame“) des Unterprogramm-Aufrufs. Dieser Speicherbereich bietet Platz für die Speicherung der Parameter, der lokalen Variablen und aller sonstiger Werte die mit einem Unterprogrammaufruf assoziiert sind. Detail dazu folgen weiter unten.

Bei jedem Unterprogramm-Aufruf wird ein neuer Rahmen auf dem Stack reserviert und beim Rücksprung wieder freigegeben. Die Anordnung der Aktivierungsrahmen gibt die Aufruf-Verschachtelung zu einem bestimmten Zeitpunkt wieder.

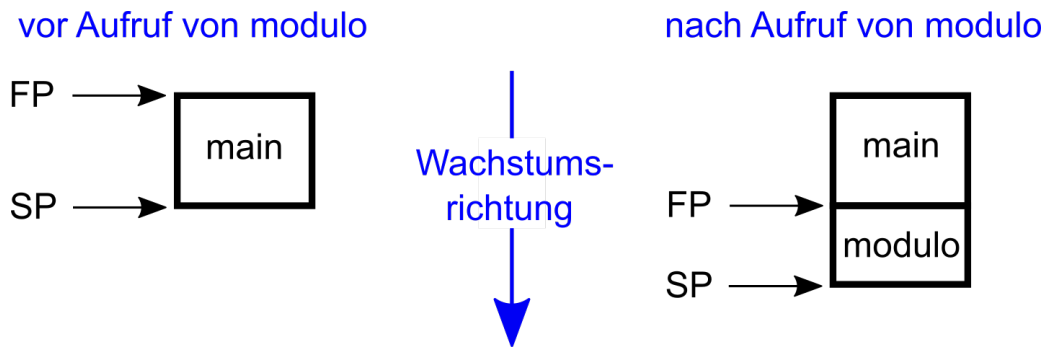
8.2.1 Stackpointer und Framepointer

Damit eine Prozedur ihren Aktivierungsrahmen findet, werden zwei Zeiger genutzt:

1. Stackpointer (SP): Der SP verweist auf das „obere“ Ende des Stacks und damit gleichzeitig auf das Ende des aktuellen Rahmens.
2. Framepointer (FP): Der FP verweist auf den Beginn des aktuellen Rahmens.

Für SP und FP werden zwei Register des Prozessors verwendet (ECO32 SP=$29, FP=$25). Die Adressen der in einem Rahmen gespeicherten Daten können jeweils in der Form *FP+Offset* oder *SP+Offset* angegeben werden. *Offset* kann vom Compiler bestimmt werden. SP und FP müssen bei jedem Aufruf und jedem Rücksprung aktualisert werden. Dazu muss der Compiler Maschinenbefehle generieren.

Stack zum Beispiel von oben:



Aus dem Gesamtlayout des Adressraums wird deutlich, dass der Stack in Richtung kleinerer Adressen wächst: Der Wachstumsrichtungspfeil in der letzten Abbildung zeigt nach unten, in der Abbildung von Seite 2 allerdings nach oben. SP ist also immer kleiner als FP, genauer gesagt gilt im Beispiel nach dem Aufruf von *modulo* :

$$SP = FP - Rahmengroesse(modulo)$$

Die Rahmengröße einer Prozedur wird vom Compiler berechnet. Beim Aufruf von *modulo* werden die beiden Verweise also offensichtlich wie folgt aktualisiert:

```
FP := SP ;
SP := FP - Rahmengröße(modulo) ;
```

Es stellt sich die Frage, ob Caller oder Callee für die Aktualisierung sorgen. In Sprachen mit *getrennter Übersetzung* (z.B. C) kann der Compiler einzelne Komponenten eines Programms auch dann übersetzen, wenn andere Komponenten noch nicht vorliegen. Auf unser Beispiel übertragen heißt das: Die Übersetzung von *main* muss ohne genaue Kenntnis von *modulo* möglich sein. Insbesondere kennt der Compiler beim Übersetzen des Callers i.A. die Rahmengröße des Callee nicht und umgekehrt. Daher wird wie folgt verfahren:

Der Callee ist für die Reservierung und die Freigabe seines Rahmens zuständig. Wenn der Compiler allerdings bei der Übersetzung von *modulo* den Maschinencode für die Freigabe des Rahmens vor Rücksprung nach *main* erzeugt, sind folgende Aktionen nötig:

```
SP := FP ;
FP := SP + Rahmengröße (main) ;
```

Das Problem liegt hier in der Bestimmung des alten FP, denn die Rahmengröße von *main* ist bei der Übersetzung von *modulo* i.A. nicht bekannt. Daher wird wie folgt verfahren:

Bei Reservierung eines neuen Rahmens wird der alte FP gerettet, d.h. in den neuen Rahmen kopiert (FP-alt). Bei Freigabe des Rahmens wird dann diese Kopie verwendet, um den FP auf den alten Wert zu restaurieren.

Aufruf von modulo:

```
FP-alt := FP;
FP := SP ;
SP := FP - Rahmengröße(modulo) ;
```

Rücksprung zu main:

```
SP := FP ;
FP := FP-alt ;
```

Man beachte, dass FP-alt ein Speicherplatz im Callee-Rahmen ist, sodass bei verschachtelten Aufrufen zu jeder Aufrufebene ein eigener FP-alt-Wert existiert.

8.2.2 Rücksprungadressen

Sowohl der Unterprogrammaufruf als auch die Rückkehr werden auf der Maschinenebene durch Sprungbefehle realisiert. Während der Compiler bei der Übersetzung des Callers die Zieladresse kennt, ist bei der Übersetzung des Callees die Aufrufstelle und damit die Zieladresse für den Rücksprung nicht bekannt.

Vielmehr wird die Rücksprungadresse zum Aufrufzeitpunkt vom Aufrufer ähnlich wie ein Argument an den Callee übergeben. Bei SPL wird dazu das Register $31 (RETURN) verwendet. Im Caller muss die Ausführung nach dem Rücksprung mit dem Maschinenbefehl fortgesetzt werden, der unmittelbar hinter dem Sprung in den Caller-Code steht. Spezielle Sprungbefehle setzen RETURN auf den korrekten Wert und schreiben dann das Sprungziel in den Programmzähler (PC) des Prozessors.

Wenn der Callee nun seinerseits eine Prozedur aufruft, wird das RETURN-Register überschrieben. Damit der Callee trotzdem korrekt zurückspringen kann, muss er den alten Wert vorher retten (in seinen Rahmen kopieren). Mit dem geretteten Wert kann er vor dem Rücksprung das RETURN-Register restaurieren. Das Retten und Restaurieren entfällt, wenn der Callee selbst keine Unterprogrammaufrufe durchführt. Bei solchen Prozeduren handelt es sich um sogenannte *Leaf Procedures*

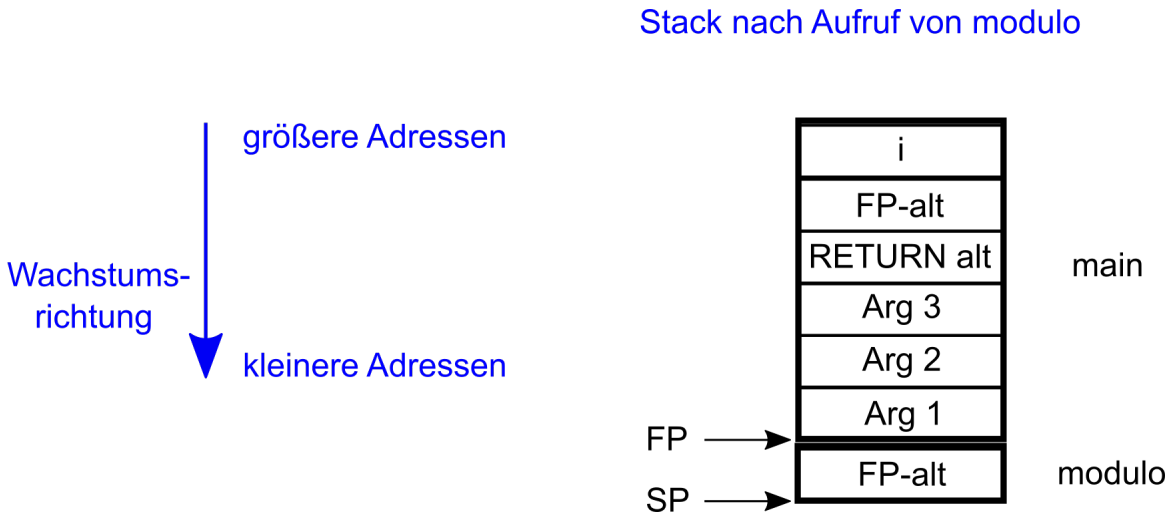
8.3 Rahmen-Layout

Informationen, die im Rahmen eines Unterprogrammaufrufs zwischen Caller und Callee ausgetauscht werden müssen, können prinzipiell sowohl auf dem Laufzeitstack als auch in den Registern des Prozessors gespeichert werden. Die gleiche Wahl besteht auch für lokale Variablen sowie für Zwischenergebnisse, die bei der Berechnung komplexer Ausdrücke gespeichert werden müssen.

Für den SPL-Compiler werden folgende Konventionen festgelegt:

1. **Argumente**
   Alle Argumente stehen im Hauptspeicher. Die Argumente für den Callee stehen am Ende des Caller-Rahmens in umgekehrter Reihenfolge: Das erste Argument steht ganz am Ende des Rahmens. Falls der Caller unterschiedliche Callees aufruft, wird der maximal benötigte Speicher für die Argumente ermittelt und am Rahmenende reserviert. Die Rahmengröße ist fix.
2. **lokale Variablen**
   Alle lokalen Variablen stehen im Hauptspeicher am Anfang des Aktivierungsrahmens.
3. **gerettete Registerinhalte**
   - hinter den lokalen Variablen steht FP-alt
   - hinter FP-alt steht RETURN-alt, falls RETURN gerettet werden muss
4. **Zwischenergebnisse**
   Alle Zwischenergebnisse stehen in Mehrzweckregistern. Wenn diese nicht ausreichen, bricht die Programmausführung mit einer Fehlermeldung ab.

Im Beispiel hat der Callee *modulo* keine lokalen Variablen und enthält keine Unterprogrammaufrufe. Daher besteht der Rahmen nur aus dem Speicherplatz für FP-alt.



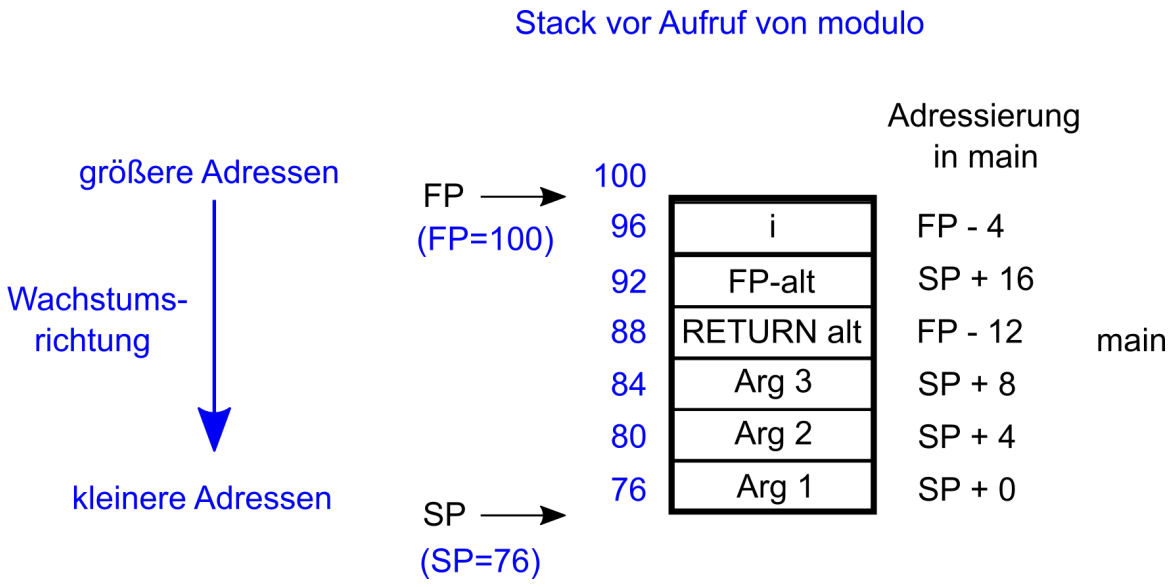
8.3.1 Adressierung der Rahmeninhalte

Für die Adressierung der Argumente verwendet der Caller SP als Bezugspunkt und der Callee FP. Da Caller-SP und Callee-FP gleich sind, gelten die gleichen Offset-Werte. Das erste Argument hat den Offset 0, weitere Argumente haben positive Offsets. Die lokalen Variablen werden relativ zu FP adressiert. Alle lokalen Variablen haben negative Offsets. FP-alt wird relativ zu SP adressiert (Offset positiv), RETURN-alt relativ zu FP (Offset negativ).

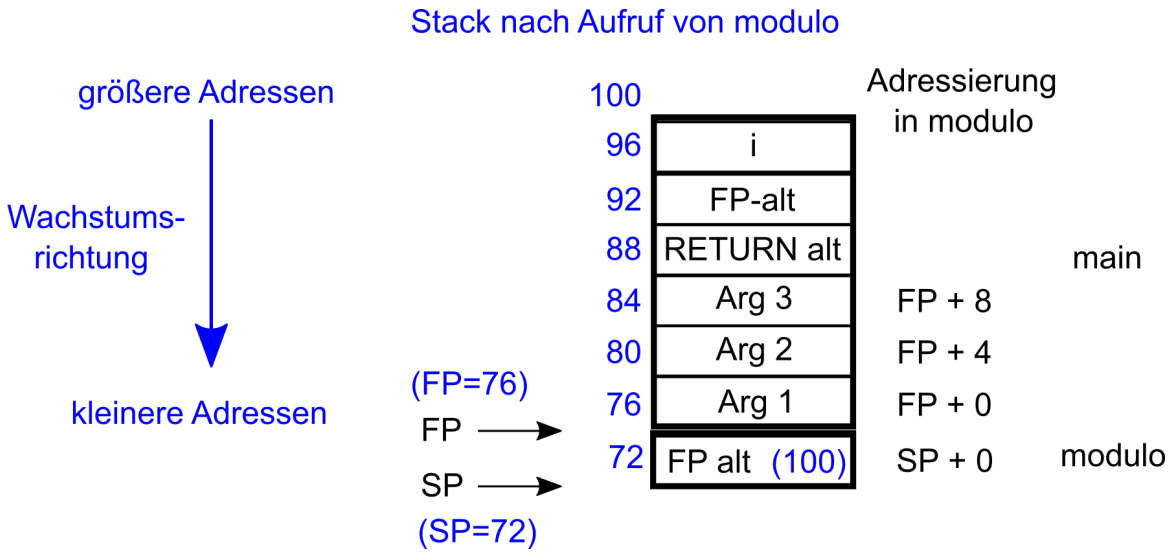
Ein Adressbeispiel:

Die Speicherplätze für Integer-Werte, Referenzen und gerettete Register sind jeweils 4 Byte groß. Die Größe für Array-Variablen ist das Produkt aus Komponentenanzahl und Größe der Komponenten.

Annahme: Der Frame von *main* „beginnt“ bei Adresse 100, d.h. vor dem Aufruf vom modulo ist FP=100. Da er 6 Speicherplätze à 4 Byte enthält, ist er 24 Byte groß. Wegen des Stack-Wachstums in Richtung kleiner werdender Adressen ist der Wert des FP tatsächlich die Adresse des letzten Byte vor dem main-Rahmen. Der main-Rahmen selbst erstreckt sich von Byte 76 bis Byte 99. Die lokale Variable *i* mit der Adresse 96 steht auf dem ersten Speicherplatz im Rahmen und belegt die Bytes 96-99. SP verweist auf das Ende des Rahmens. Dort ist Platz für das erste Argument für *modulo* in den Bytes 76-79.



Nach dem Aufruf von *modulo* sieht das Bild wie folgt aus:



8.4 Variablen-Allokation

Vor der Codegenerierung muss der Compiler für alle Parameter und Variablen die Offsets bestimmen. Ebenso muss die Framegröße für alle Prozeduren berechnet werden und das Frame-Layout:

a) **Speicherbedarf für Typen bestimmen**
Der Speicherbedarf für eine Variable oder einen Parameter hängt vom Typ ab und wird pro Typ nur einmal berechnet. Das Ergebnis wird im Typ gespeichert (*byteSize*). Diese bei komplexen Typen rekursive Berechnung kann innerhalb der Typkonstruktoren erfolgen.

b) **Parameter-Offsets bestimmen**
Die Offsets der Argumente bzw. Parameter werden sowohl für die Aufrufstelle als auch für den Zugriff innerhalb des Callees benötigt. Daher müssen sie in der lokalen Symboltabelle (für Callee) und in der Parametertypliste (für Caller) abgespeichert werden. Dies lässt sich z.B. in einer Schleife, ausgehend vom AST-Knoten für die Parameterdeklarationen bewerkstelligen. Am Ende der Schleife ist die Gesamtgröße für die Parameter bekannt und wird in der Symboltabelle der Prozedur gespeichert.

c) **Variablen-Offsets bestimmen**
Die Offsets für die Variablen müssen in die lokale Symboltabelle eingetragen werden. Dies lässt sich z.B. in einer Schleife, ausgehend vom AST-Knoten für die Variablendeklarationen bewerkstelligen. Am Ende der Schleife ist die Gesamtgröße für die Variablen bekannt und wird gespeichert.

d) **Speicherbedarf für Argumentbereich bestimmen**
In einem zweiten Durchgang kann dann zu jeder Prozedur die Größe des Argumentbereichs am Rahmenende bestimmt werden: Anhand des AST-Knotens für den Prozedurrumpfs ermittelt der Compiler alle darin vorkommenden Aufrufe. Zu jedem Aufruf wird aus dem Symboltabelleneintrag des Callee die benötigte Speichergröße für die Argumente entnommen. Das Maximum für alle Aufrufe wird bestimmt und für den Caller gespeichert.
Wenn keine Aufrufe gefunden werden, muss der Compiler sich das merken (z.B. outgoingAreaSize = -1), denn in diesem Fall wird später vom Code-Generator das kurze Rahmen-Layout verwendet (ohne RETURN-alt).

Für die Implementierung der am AST orientierten Algorithmen bietet sich wieder das Visitor-Muster an.

9 Assembler-Code-Generator für SPL

Der SPL-Compiler erzeugt Assembler-Code für eine RISC-Maschine. Diese Maschine nutzt bei Berechnungen ihre Mehrzweckregister für die Speicherung von Operanden und Berechnungsergebnissen. Eines der Probleme bei der Implementierung des Code-Generators liegt in der Registerbewirtschaftung: Nach welchem Verfahren werden die Register für abzuspeichernde Werte zugeordnet? Im Praktikum wird eine einfache Strategie dafür verfolgt, nämlich die Simulation einer Stackmaschine.

In diesem Abschnitt wird zuerst die ECO32-Zielplattform näher spezifiziert. Danach wird das Stackmaschinen-Prinzip erläutert und anschließend die Stackmaschinen-Simulation durch die RISC-Maschine dargestellt.

Schließlich wird für ausgewählte SPL-Sprachkonstrukte die Codeerzeugung diskutiert.

9.1 Zielplattform

Zielplattform ist die an der Technische Hochschule Mittelhessen unter Federführung von Prof. Dr. Hellwig Geisse entwickelte RISC-Plattform ECO32 (https://github.com/hgeisse/eco32).

9.1.1 ECO32 Registernutzung

Die 32-Bit RISC-Maschine hat 32 Mehrzweckregister ($0, . . . , $31) der Größe 32 Bit. Folgende Register werden von der Codegenerierung für spezielle Zwecke genutzt:

- $0 hat immer den Wert 0 und kann nicht verändert werden,
- $29 speichert den Stackpointer (SP).
- $25 speichert den Framepointer (FP),
- $31 enthält die Rücksprungadresse (RETURN)

Die Mehrzweckregister $8-$23 werden für die Speicherung von Zwischenwerten genutzt.

Integer Registers

$0	always zero	$16	temporary variable
$1	DO NOT USE	$17	temporary variable
$2	DO NOT USE	$18	temporary variable
$3	DO NOT USE	$19	temporary variable
$4	DO NOT USE	$20	temporary variable
$5	DO NOT USE	$21	temporary variable
$6	DO NOT USE	$22	temporary variable
$7	DO NOT USE	$23	temporary variable
$8	temporary variable	$24	DO NOT USE
$9	temporary variable	$25	frame pointer
$10	temporary variable	$26	DO NOT USE
$11	temporary variable	$27	DO NOT USE
$12	temporary variable	$28	DO NOT USE
$13	temporary variable	$29	stack pointer
$14	temporary variable	$30	DO NOT USE
$15	temporary variable	$31	return address

9.1.2 Register und Unterprogrammaufrufe

Bei Prozeduraufrufen ist der Caller für die korrekte Belegung von RETURN verantwortlich. Der Callee muss SP und FP beim Aufruf aktualisieren (Prolog) und vor dem Rücksprung auf die alten Werte zurücksetzen (Epilog).

9.1.3 ECO32 Instruktionssatz

Format	Description
N	no operands
R	one register
RH	one register and the lower 16 bits of a word
RHh	one register and the upper 16 bits of a word
RRH	two registers and a zero-extended halfword

Format	Description
RRS	two registers and a sign-extended halfword
RRR	three registers
RRX	three registers, or two registers and a zero-extended halfword
RRY	three registers, or two registers and a sign-extended halfword
RRB	two registers and a sign-extended 16 bit offset
J	no registers and a sign-extended 26 bit offset

Mnemonic	Operands	Description	Format
add	dst, op1, op2	dst := op1 + op2	RRY
sub	dst, op1, op2	dst := op1 - op2	RRY
mul	dst, op1, op2	dst := op1 * op2, signed	RRY
mulu	dst, op1, op2	dst := op1 * op2, unsigned	RRX
div	dst, op1, op2	dst := op1 / op2, signed	RRY
divu	dst, op1, op2	dst := op1 / op2, unsigned	RRX
rem	dst, op1, op2	dst := remainder of op1/op2, signed	RRY
remu	dst, op1, op2	dst := remainder of op1/op2, unsigned	RRX
and	dst, op1, op2	dst := bitwise AND of op1 and op2	RRX
or	dst, op1, op2	dst := bitwise OR of op1 and op2	RRX
xor	dst, op1, op2	dst := bitwise XOR of op1 and op2	RRX
xnor	dst, op1, op2	dst := bitwise XNOR of op1 and op2	RRX
sll	dst, op1, op2	dst := shift op1 logically left by op2	RRX
slr	dst, op1, op2	dst := shift op1 logically right by op2	RRX
sar	dst, op1, op2	dst := shift op1 arithmetically right by op2	RRX
ldhi	dst, op1	dst := op1 shifted left by 16 bits	RHh
beq	op1, op2, offset	branch to PC+4+offset*4 if op1 == op2	RRB
bne	op1, op2, offset	branch to PC+4+offset*4 if op1 != op2	RRB
ble	op1, op2, offset	branch to PC+4+offset*4 if op1 <= op2 (signed)	RRB
bleu	op1, op2, offset	branch to PC+4+offset*4 if op1 <= op2 (unsigned)	RRB
blt	op1, op2, offset	branch to PC+4+offset*4 if op1 < op2 (signed)	RRB
bltu	op1, op2, offset	branch to PC+4+offset*4 if op1 < op2 (unsigned)	RRB
bge	op1, op2, offset	branch to PC+4+offset*4 if op1 >= op2 (signed)	RRB
bgeu	op1, op2, offset	branch to PC+4+offset*4 if op1 >= op2 (unsigned)	RRB
bgt	op1, op2, offset	branch to PC+4+offset*4 if op1 > op2 (signed)	RRB
bgtu	op1, op2, offset	branch to PC+4+offset*4 if op1 > op2 (unsigned)	RRB
j	offset	jump to PC+4+offset*4	J
jr	register	jump to register	R
jal	offset	jump to PC+4+offset*4, store PC+4 in $31	J
jalr	register	jump to register, store PC+4 in $31	R
trap	-/-	cause a trap, store PC in $30	N
rfx	-/-	return from exception, restore PC from $30	N
ldw	dst, reg, offset	dst := word @ (reg+offset)	RRS
ldh	dst, reg, offset	dst := sign-extended halfword @ (reg+offset)	RRS
ldhu	dst, reg, offset	dst := zero-extended halfword @ (reg+offset)	RRS
ldb	dst, reg, offset	dst := sign-extended byte @ (reg+offset)	RRS
ldbu	dst, reg, offset	dst := zero-extended byte @ (reg+offset)	RRS
stw	src, reg, offset	store src word @ (reg+offset)	RRS
sth	src, reg, offset	store src halfword @ (reg+offset)	RRS
stb	src, reg, offset	store src byte @ (reg+offset)	RRS
mvfs	dst, special	dst := contents of special register	RH
mvts	src, special	contents of special register := src	RH
tbs	-/-	TLB search	N
tbwr	-/-	TLB write random	N
tbri	-/-	TLB read index	N
tbwi	-/-	TLB write index	N

9.2 Stackmaschine

Eine Stackmaschine ist ein Prozessor, dessen Mehrzweckregister als Stack organisiert sind. Um Verwechslungen zu vermeiden, bezeichnen wir diesen Stack im folgenden auch als Registerstack (RSTACK) und den für die Prozeduraufrufe im Hauptspeicher genutzten Stack der Aktivierungsrahmen als Laufzeitstack (LZSTACK). Für den Registerstack ist ein Spe-

zialregister vorhanden, das auf das letzte belegte Register von RSTACK verweist: Der Registerstack-Pointer (RSP). Die Ausführung einer Maschineninstruktion lässt sich in verschiedene Teilschritte untergliedern. Dabei werden viele Maschineninstruktionen RSP nutzen und auch verändern.

Beispiel:

Der Ausdruck `3*5+2*6` erfordert zunächst zwei Multiplikationen und am Ende eine Addition. Jede dieser Rechenoperationen entspricht einer Maschineninstruktion (ADD, MUL). Diese Instruktionen erwarten ihre Operanden oben auf dem RSTACK, genauer: Der linke Operand steht in `RSTACK[RSP-1]` der rechte in `RSTACK[RSP]`. Bei der Ausführung der Instruktion werden implizit die Operanden vom Registerstack entfernt und durch das Ergebnis ersetzt:

```
PUSH 3       ; RSTACK[++RSP]<--3
PUSH 5       ; RSTACK[++RSP]<--5
MUL          ; RSTACK[RSP-1]<--RSTACK[RSP-1]*RSTACK[RSP], RSP--
PUSH 2       ; RSTACK[++RSP]<--2
PUSH 6       ; RSTACK[++RSP]<--6
MUL          ; RSTACK[RSP-1]<--RSTACK[RSP-1]*RSTACK[RSP], RSP--
ADD          ; RSTACK[RSP-1]<--RSTACK[RSP-1]+RSTACK[RSP], RSP--
```

Der Codegenerator für eine binäre Operation verwendet also immer das gleiche Muster:

1. Code zur Berechnung des linken Operanden generieren (Rekursion)
2. Code zur Berechnung des rechten Operanden generieren (Rekursion)
3. Maschinenbefehl für die Rechenoperation anhängen

9.3 Simulation der Stackmaschine durch RISC-Prozessor

Ein einfaches Prinzip für die Nutzung der Mehrzweckregister unserer ECO32 VM ist die Simulation der Stackmaschine. Der Codegenerator nutzt die 16 Register $8-$23 in Form eines Register-Stacks. Beispielsweise könnte man diese Register gemäß aufsteigender Nummerierung benutzen. Um das RSP-Register der Stackmaschine zu ersetzen, merkt sich der SPL-Compiler immer das letzte belegte Register in einer internen Variable und generiert den Code so, dass die aktuelle Instruktion immer die Register nutzen, die den „obersten“ des simulierten Registerstacks entsprechen. Das heißt, wenn ein Speicherplatz benötigt wird, wird immer das freie Register mit der kleinsten Nummer verwendet. Sobald ein Wert nicht mehr gebraucht wird, steht dieses Register wieder zur Verfügung.

Die Simulation der Berechnung von `3*5+2*6` im Vergleich mit der Stackmaschine unter der Annahme, dass zu Beginn das Register $8 schon belegt und $9 das erste freie Register ist:

```
Stack-Maschine              ECO32

 PUSH 3                     add $9,$0,3      ;  $9 <-- 3
 PUSH 5                     add $10,$0,5     ; $10 <-- 5
 MUL                        mul $9,$9,$10    ;  $9 <-- $9*$10
 PUSH 2                     add $10,$0,2     ; $10 <-- 2
 PUSH 6                     add $11,$0,6     ; $11 <-- 6
 MUL                        mul $10,$10,$11  ; $10 <-- $10*$11
 ADD                        add $9,$9,$10    ;  $9 <-- $9+$10
```

Man beachte, dass man eine *add*-Instruktion mit Register `$0` (enthält immer 0) verwendet, um einen Direktwert in einem Register zu speichern.

9.4 Lokale Variablen und Parameter

Lokale Variablen und Parameter stehen im Laufzeitstack und werden in der Form *FP + Offset* adressiert. Ein lesender Zugriff (*VariableExpression*-Knoten im AST) wird durch eine `ldw`-Instruktion („load word“) umgesetzt, ein schreibender Zugriff (z. B. linke Seite der Wertzuweisung) durch ein `stw`-Instruktion („store word“).

Bei den Hauptspeicherzugriffen können die Adressen in der Form *Registerinhalt+Offset* angegeben werden. Diese Möglichkeit kann der Codegenerator in einigen Fällen nutzen, in anderen dagegen nicht.

Beispiel:

```
proc p(){
  var i: int;
  var j: int;
  i := 5;
```

```
  j := i+1;
}
```

Eine einfache Strategie für Wertzuweisungen ist:

- Zieladresse in `$8` berechnen
- Wert der rechten Seite in `$9` berechnen
- Speicherinstruktion `stw` anhängen

Betrachten wir zuerst die Wertzuweisung `i := 5;` :

Da *Offset(i)* = -4, FP=$25, ergibt sich folgender Code

```
add $8,$25,-4   ; $8 <-- Adresse(i) = FP-4
add $9,$0,5
stw $9,$8,0
```

Natürlich ist das gleiche Ergebnis auch mit nur zwei Instruktionen erreichbar:

```
add $8,$0,5
stw $8,$25,-4
```

Auf die separate Berechnung der Zieladresse wird hier verzichtet, da die `stw`-Instruktion die erforderliche Berechnung ebenso durchführen kann. Aber: Beim Zugriff auf Array-Komponenten ist die separate Adressberechnung in jedem Fall notwendig. Der Codegenerator müsste also für einfache Variablen und Arrays unterschiedlich vorgehen und würde dadurch komplexer. Zugunsten der Einfachheit folgen wir daher in den Beispielen der zuerst aufgezeigten Strategie.

Die zweite Wertzuweisung: Da *Offset(j)* = -8, ist der Assemblercode zur Wertzuweisung `j := i+1;` wie folgt untergliedert:

1. Zieladresse in `$8` berechnen:
   ```
   add $8,$25,-8   ; $8 <-- Adresse(j)
   ```
2. Wert der rechten Seite berechnen und in Register 9 speichern.
   ```
                     ; linker Operand der Addition
   add  $9,$25,-4  ; $9 <-- Adresse(i)
   ldw  $9,$9,0    ; $9 <-- Wert(i)
                     ; rechter Operand der Addition
   add  $10,$0,1
                     ; Maschineninstruktion entsprechend SPL-Operator
   add $9,$9,$10
   ```
3. Mit `stw` den Inhalt von Register 9 in den Hauptspeicher kopieren.
   ```
   stw $9,$8,0
   ```

9.4.1 Referenzparameter

Da der Wert eines Referenzparameters des Callee die Adresse einer Variable des Callers ist, muss die im SPL-Quelltext nicht sichtbare zusätzliche Dereferenzierung auf der Maschinenebene durch eine zusätzliche ldw-Instruktion umgesetzt werden.

Beispiel:

```
proc p(ref i:int){
  i := i+1;
}
```

Da *Offset(i)* = 0, sieht der Code für die linke Seite der Wertzuweisung wie folgt aus:

```
add  $8,$25,0   ; $8 <-- Adresse(i)
ldw  $8,$8,0    ; $8 <-- Adresse der Variable, auf die i verweist
```

Wert der rechten Seite berechnen und in Register 9 speichern.

```
add  $9,$25,0   ; $9 <-- Adresse(i)
ldw  $9,$9,0    ; $9 <-- Adresse der Variable, auf die i verweist
ldw  $9,$9,0    ; $9 <-- Wert der Variable, auf die i verweist
add  $10,$0,1
add  $9,$9,$10
```

9.4.2 Array-Komponenten

Bei Array-Komponenten wird die Adresse wie folgt berechnet:

Feldanfangsadresse $+$ *Indexwert* $*$ *Bytegröße der Feldkomponenten*

Die Feldanfangsadresse wird wie bei einfachen Variablen aus *FP+Offset* bestimmt. Die Bytegröße der Feldkomponenten findet der Compiler im Basistyp des Array-Typs des Feldes. Der Compiler soll durch zusätzliche Instruktionen sicher stellen, dass zur Laufzeit eine Indexprüfung erfolgt.

Tipp: Zwei Bedingungen müssen erfüllt sein, damit ein Array-Index i im erlaubten Bereich liegt:

1. $i \geq 0$
2. $i < Komponentenanzahl$

Anstatt Maschinencode für zwei Vergleichsoperationen zu erzeugen, kann der Codegenerator auch i als vorzeichenlose Zahl interpretieren und mit der Komponentenanzahl vergleichen. Dabei würden negative Werte wegen der 1 im höchstwertigen Bit der Integer-Representation immer als zu groß angesehen werden.

9.5 Bedingte Anweisungen und Schleifen

Zu den SPL-Vergleichsoperationen gibt es in der ECO32 VM keine äquivalenten Machineninstruktionen. Stattdessen kann man nach der Berechnung der beiden Operanden mit bedingten Sprunganweisungen abhängig vom Testausgang zu einer anderen Stelle im Code springen. Die Sprungadressen werden im Assembler durch Labels repräsentiert.

Ein Beispiel: IF-Anweisung ohne ELSE

```
if ( 3 < 5 )
  STMT
```

Zunächst wird der Maschinencode für den Ausdruck `(3 < 5)` erzeugt.

```
add $8,$0,3
add $9,$0,5
```

Anschließend erfolgt ein bedingter Sprung zum Ausgang der IF-Anweisung mit der Instruktion `bge` („branch if greater or equal“):

```
bge $8,$9,label_001
```

Ein Label-Bezeichner wie `label_001` repräsentiert eine Instruktionsadresse auf der Assemblerebene, die vom Assembler in eine Hauptspeicheradresse umgesetzt wird. Das Sprungziel wird durch eine Markierung definiert, die aus einer Zeile besteht, in der der Label-Bezeichner gefolgt von einem Doppelpunkt steht. Im Beispiel muss diese Markierung hinter dem letzten Maschinenbefehl von STMT stehen, so dass der gesamte Code wie folgt aussieht:

```
    add  $8,$0,3
    add  $9,$0,5
    bge  $8,$9,label_001  ; falls $8 >= $9, springe zu label_001
       .
       .         Code für STMT
       .
label_001:
```

Das Sprungziel ist also die Adresse der ersten Maschineninstruktion, die hinter dem Code der IF-Anweisung folgt. Die Labels werden vom Compiler nach Bedarf generiert und können beliebige Bezeichner sein.

9.6 Unterprogrammaufrufe

9.6.1 Caller

Im Caller muss zum Aufruf zunächst Code für die Übergabe der Argumente erzeugt werden. Zunächst müssen die Argumentwerte in einem Register (immer `$8`) berechnet und danach an das Ende Caller-Stackframe kopiert werden. Es handelt sich also um Hauptspeicher-Schreibzugriffe, die mit `stw`-Instruktionen implementiert werden.

Die Übergabe eines Arguments ist bei einem Wertparameter auf den ersten Blick vergleichbar mit einer Wertzuweisung: Zieladresse berechnen, Wert berechnen, `stw`-Instruktion. Allerdings ist hier die Adressberechnung immer sehr einfach:

$$Parameteradresse = SP + Offset(Parameter)$$

Die Adressberechnung der Zieladresse kann in der `stw`-Instruktion erfolgen. *Offset* kann aus der Parametertypliste des Callee entnommen werden.

Bei Referenzparametern muss die Adresse der Caller-Variablen übergeben werden, bei Wertparametern ist zuerst die Adresse zu berechnen und danach (mit `ldw`) der Wert zu bestimmen.

Nach der Argumentübergabe muss der Einsprung in den Callee und die Übergabe der Rücksprungadresse im Register `$31` erfolgen. Dies erledigt die `jal`-Instruktion („jump and link“), die das Label für den ersten Maschinenbefehl des Callee als Argument bekommt. Vor dem Sprung wird die Adresse der auf `jal` folgenden Instruktion in `$31` gespeichert.

Beispiel:

```
proc p(i:int, ref j:int) {
  var m: int;
  j := i;
  printi(j);
}

proc q(){
   var k: int;
   k := 1;
   p(k, k);
}
```

Die Offset-Werte sind 0 für das erste und 4 für das zweite Argument von p. Der Code für `p(k,k)`:

```
                  ; Wert des 1. Arguments berechnen: Wert(k)
add  $8,$25,-4    ; $8 <-- Adresse(k)
ldw  $8,$8,0      ; $8 <-- Wert(k)

stw  $8,$29,0     ; 1. Argument im Stackframe speichern

                  ; Adresse 2. Arguments berechnen: Adresse(k)
add  $8,$25,-4    ; $8 <-- Adresse(k)
                  ; kein ldw hier, da ein ref-Parameter vorliegt
stw  $8,$29,4     ; 2. Argument im Stackframe speichern
jal  p            ; Einsprung mit Übergabe der Rücksprungadresse in $31
```

Für die `jal`-Instruktion muss vor dem ersten Befehl einer Prozedur ein entsprechendes Label in den Assemblercode eingefügt werden (siehe Prolog von *q* im nächsten Abschnitt).

9.6.2 Callee

Im Callee müssen SP und FP aktualisiert werden, so dass diese auf Ende bzw. Anfang des neuen Stackframe verweisen. Der alte FP muss vor dem Überschreiben gerettet werden. Falls im Callee-Rumpf selbst wieder Aufrufe anderer Prozeduren vorkommen, muss auch Register 31 gerettet werden:

Prolog von q aus dem Beispiel von oben:

```
q:                  ; Prozedurname als Label für den jal-Befehl
                    ; SP aktualisieren
    sub  $29,$29,20 ; SP <-- SP - Framesize(q)
                    ; FP retten und danach aktualisieren
    stw  $25,$29,12 ; FP alt speichern
    add  $25,$29,20 ; FP neu <-- SP neu + Framesize(q)
    stw  $31,$25,-12 ; RETURN retten
```

Vor dem Rücksprung müssen alle Aktionen aus dem Prolog wieder rückgängig gemacht, also FP, SP und RETURN wieder auf die alten Werte gesetzt werden. Der Rücksprung erfolg mit jr.

Epilog von q:

```
ldw   $31,$25,-12 ; RETURN restaurieren
ldw   $25,$29,12  ; FP alt restaurieren
add   $29,$29,20  ; SP <-- SP + Framesize(q)
jr    $31         ; Rücksprung
```

Hinweis: Man beachte, dass bei Prozeduren, die selbst keine anderen Prozeduren aufrufen, die Rettung und das Restaurieren von `$31` entfallen. Der Codegenerator muss also für Prolog und Epilog eine entsprechende Fallunterscheidung machen. Dazu ist entweder für jede Prozedur ein zusätzliches boolesches Attribut (z.B. *isCaller*) nötig, oder die *VarAlloc*-Komponente vergibt z.B. einen negativen Wert für *outgoingArea*.

9.6.3 Direktiven

Für den Linker müssen in Ergänzung zu den Maschineninstruktionen und Labels noch bestimmte Assembler-Direktiven in den Code eingefügt werden:

- Import der Prozeduren aus der Standardbibliothek:

  ```
  .import printi
  .import printc
  .import readi
  .import readc
  .import exit
  .import time
  .import clearAll
  .import setPixel
  .import drawLine
  .import drawCircle
  .import _indexError
  ```

  Vor der ersten Instruktion:

  ```
  .code
  .align  4
  ```

  Vor jeder Prozedur:

  ```
  .export <Prozedur-ID>
  ```