Vorlesungsmitschrift zur Vorlesung Übersetzerbau

Naja v. Schmude
www.najas-corner.de

WS 2009/2010

Inhaltsverzeichnis

Vorlesung 1, 14. Oktober 2009

1 Einführung

Ein Übersetzer (engl. Compiler) übersetzt im Allgemeinen einen Text einer Quellsprache in einen Text der Zielsprache. Dabei soll das ganze bedeutungserhaltend geschehen, so dass die Semantik erhalten bleibt.
In der Informatik hat man stattdessen ein Quellprogramm, der durch den Compiler in ein Zielprogramm übersetzt wird.

Beispiel (Hellsehen) „Michael Baumann läuft 10 m pro Sekunde.“ Gegeben sei eine Position und die Dauer, die er läuft. Dann soll vorausgesehen werden, wo er sein wird, also die neue Position. Die Berechnungsvorschrift würde hier sein:

$$neuePosition = altePosition + Dauer * 10\frac{m}{s}$$

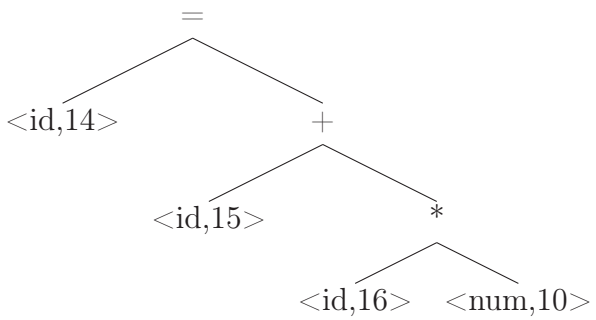
So etwas wäre durch einen Computer nicht ausführbar.
Bei größeren Softwaresystemen ist der Compiler eingebettet. Dabei hat einen Präcompiler, der das Quellprogramm einliest und leicht modifiziert. Dieses modifizierte Quellprogramm wird vom Übersetzer in ein Assembler-Programm übersetzt. Dieses geht dann in den Assembler rein, der verschiebbaren Maschinencode erzeugt. Dieser Code geht wiederum in den Linker (Lader, der den Code mit Bibliotheksdateien koppelt und dadurch das Zielprogramm erzeugt.
Folgende Phasen durchläuft der Compiler und erledigt dabei die aufgeführten Teilaufgaben:

Lexikalische Analyse Es wird eine Symboltabelle der auftretenden Wörter erstellt, die Speichert, wo welches Wort zu finden ist.

```
------------------------
| 14 | neuePosition |  |
| 15 | altePosition |  |
| 16 | dauer        |  |
| .. | ...          |  |
------------------------
```

Die Ausgabe könnte eine Tokentabelle sein, die aus Tokens des Schemas `<Symbol, Verweis auf Symboltabelle>` besteht.

Syntaxanalyse Diese Analyse baut einen Syntaxbaum auf, der hierarchisch die Struktur des Ausdrucks darstellt.



Semantische Analyse Typen usw. werden untersucht, ob sie konform gehen. Als Ausgabe gibt es ein attributierten Syntaxbaum.

Zwischencodeerzeugung Hier wird ein Zwischencode erzeugt. Typischerweise verwendet man einen *Drei-Adress-Code*, bei dem man mit drei temporären Adressen arbeitet. Unser Beispiel könnte dann wie folgt aussehen (`t1` bis `t3` sind dabei die temporären Adressen):

```
t1 = inttofloat 10
t2 = id16 * t1
t3 = id15 + t2
id14 = t3
```

Optimierung Hier wird der 3-Adresss-Code optimiert. Die Ausgabe ist ein (hoffentlich) kürzere Zwischencode.

```
t1 = id16 * 10.0
id14 = id15 + t1
```

Codeerzeugung Die Hauptaufgabe der Codeerzeugung ist die Registerzuweiseung. Es wird Assemberlcode erzeugt.

```
LDF R2 id16          // Load Float
MULF R2 R2 #10.0     // Float Multiplikation R2 * 10.0
LDF R1 id15
ADDF R1 R1 R2        // Float Addition R1 + R2
```

Maschinenabhängige Optimierung Diese zweite Optimierung erzeugt dann letztendlich das Maschinenprogramm.

Die Schritte bis zum Zwischencode sind im *Front-End*, die restlichen Schritte bezeichnet man als *Back-End*. Grob kann man das so unterscheiden, dass im Front-End alle Schritte stecken, die nur von der Quellsprache abhängen. Die im Back-End sind auch (nur?) von der Zielsprache abhängig.

Für jede Computerarchitektur und für jede Programmiersprache benötigt man eigene Compiler. Für N Sprachen und M Architekturen würde man also $N * M$ Compiler benötigen. Wenn man nun nur für jede Programmiersprache einen Compiler fürs Front-End hat, dann braucht man nur noch für jede Architektur einen Compiler, der für das Back-End zuständig ist. Dadurch reduziert sich die benötigte Anzahl an Compiler auf $N + M$.

Vorlesung 2, 15. Oktober 2009

Wir gehen jetzt die einzelnen Arbeitsschritte des Compilers durch (nicht unbedingt in der richtigen Reihenfolge). Wir fangen mit der Syntaxanalyse an.

2 Syntaxanalyse

Wir setzen voraus, das alle Einheiten der lexikalischen Sprache durch ein Symbol (Character) dargestellt werden.
Der Ausgangspunkt für diese Analyse ist eine kontextfreie Grammatik zur Spezifikation der Syntax.

Definition 1 (Kontextfreie Grammatik)**.** *Ein Viertupel* (N, T, S, P) *heißt kontextfreie Grammatik, genau dann wenn gilt:*

- N *ist eine nichtleere, endliche Menge von Symbolen.(Nichtterminale)*
- T *ist eine endliche Menge von Symbolen (Terminale) mi* $N \cap T = \emptyset$*.*
- $S \in N$ *ist das Startsymbol.*
- P *ist eine endliche Menge von Produktionen der Form* $A \to \alpha$ *mit* $A \in N$ *und* $\alpha \in (N \cup T)^*$*.*

(Wir führen hier die Konvention ein, dass Nichtterminale in Schreibschrift geschrieben werden, Terminale unterstrichen (wenn mehr als ein Buchstabe). Als Folge daraus genügt es, die Folge der Produktionen anzugeben, wobei das erste Nichtterminalsymbol als Startsymbol gilt.)

Beispiel Die Grammatik G_1 wird durch folgende Produktionen definiert:

$$
\begin{aligned}
exp &\to exp + digit \\
exp &\to exp - digit \\
exp &\to digit \\
digit &\to 0 \\
&\vdots \\
digit &\to 9
\end{aligned}
$$

Dies impliziert, dass *exp* das Startsymbol ist und $digit \in N$. $T = \{+, -, 0, \ldots, 9\}$. Als weitere Vereinfachung können Produktionen mit derselben linken Seite zusammengefasst werden; durch Trennung der möglichen rechten Seite mit |.

Definition 2 (Herleitung)**.** *Eine Satzform $\alpha \in (N \cup T)^*$ lässt sich in einem Schritt zu einer Satzform β bzgl. einer gegebenen Grammatik herleiten, wenn es eine Produktion $N \to \gamma$ gibt, so dass $\alpha = \alpha_1 N \alpha_2$ und $\beta = \alpha_1 \gamma \alpha_2$ mit beliebigem $\alpha_1, \alpha_2 \in (N \cup T)^*$. Man notiert $\alpha \to_G \beta$ oder $\alpha \to \beta$. Wir sagen, dass α sich bzgl. einer Grammatik G zu β herleiten lässt, genau dann wenn $\alpha \to_G^* \beta$.*

Wir bezeichnen ϵ als das leere Wort.

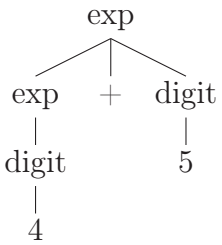
Definition 3 (Sprache zu G)**.** *Sei G eine kontextfreie Grammatik. Die Sprache $L(G)$ ist die Menge aller Wörter über T, die aus S herleitbar sind.*

$$L(G) := \{w | w \in T^*, S \to_G^* w\}$$

Um zu prüfen, ob ein Ausdruck in einer Sprache liegt, muss versucht werden vom Startsymbol über die Produktionen auf den Ausdruck zu kommen.

Beispiel $9 - 5 + 2$ sei der Ausdruck. Ist es in $L(G_1)$? Wir führen folgende Herleitung durch: $exp \to exp + digit \to exp - digit + digit \to digit - digit + digit \to 9 - 5 + 2$. Das Wort ist also in der Sprache.

Definition 4 (Parsebaum)**.** *Jede Produktion der Form $A \to X_1 X_2 \ldots X_n$ ist ein elementarer Baum mit A in der Wurzel und allen X_i in den Blättern zugewiesen. Des weiteren ist $A \to \epsilon$ ebenso darstellbar. Ein Baum B mit Wurzel $A \in N$ und Knoten aus N und Blättern aus $(N \cup T)^*$ heißt Parsebaum zu α, genau dann wenn A Startsymbol ist und für jeden Knoten einschließlich der Wurzel gilt, dass der Teilbaum N zusammen mit allen seinen Kindern zu einer Produktion aus G gehört und α die Hintereinanderhängung der Blätter von B ist.*

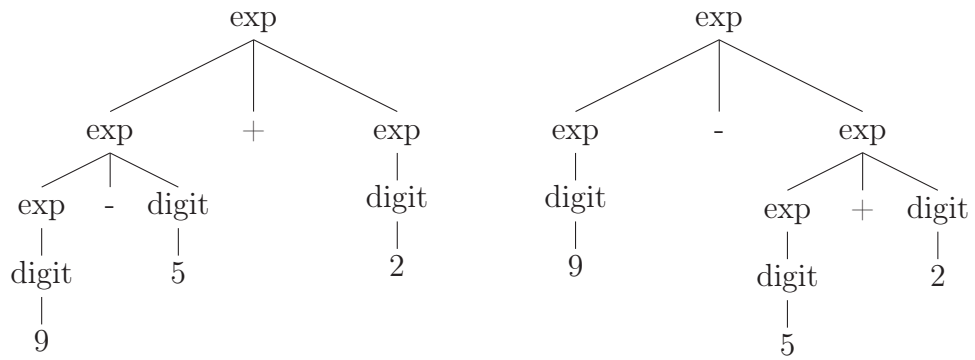


Der Parsebaum entspricht der grammatikalischen Struktur. Der Parsebaum beschreibt eine Klasse von Herleitungen. Z.B. sind bei $exp \to exp + digit \to digit + digit \to 4 + digit \to 4 + 5$ und $exp \to exp + digit \to exp + 5 \to digit + 5 \to 4 + 5$ die Herleitungen unterschiedlich, aber der Parsebaum ist identisch.

Definition 5 (Mehrdeutigkeit)**.** *Eine Grammatik G heißt mehrdeutig, wenn es ein Wort $w \in L(G)$ gibt, dass mehr als einen Parsebaum hat.*

Beispiel Die Grammatik sei

$$e \to e + e | e - e | d$$
$$e \to 0 | \ldots | 9$$



Erzeugung eindeutiger G für Operatoraudrücke. Schreibe die Operatoren mit ihren Assoziativitäten geschickt in eine Tabelle mit aufseigender Priorität.

$+, -$	linksassoziativ
$*, /$	linksassoziativ

Erzeuge ein Nichtterminal für jedes Niveau (hier expr, term) und ein zusätzliches für Grundbausteine (hier factor). Beginne von unten nach oben.

$$factor \to digit | (expr)$$
$$term \to term * factor | term / factor | factor$$
$$expr \to expr + term | expr - term | term$$

Vorlesung 3, 21. Oktober 2009

2.1 Syntaxgerichtete Übersetzung

Diese Übersetzung ist wichtig für die semantische Analyse und die Erzeugung von Code.

Idee Die Idee ist, dass man einer induktiven Definition folgt. Z.B. ist ein Programmkonstrukt, dass aus der Regel $exp \to exp + num$ generiert wurde, wie folgt zu übersetzten:

```
translate exp;
translate num;
handle +;
```

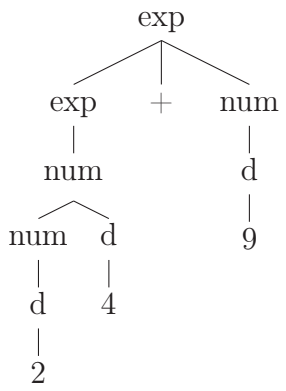
Durch eine einfache Stack-Architektur kann man arithmetische Ausdrücke berechnen. Der Befehlssatz sieht wie folgt aus: `Push n, Add, Mult`. Der Befehl `Push n` packt den Wert n oben auf den Stack rauf. `Add` und `Mult` können nur ausgeführt werden, wenn schon mind. zwei Werte im Stack liegen. Durch diese Befehle werden die obersten zwei Einträge verbraucht und addiert bzw. multipliziert und das Ergebnis wieder auf den Stack gespeichert (beachte, dass der linke Operator der weiter unten im Stack liegende ist).

```
|   |         |   |
|   | Push n  | n |
| c | --->    | c |
| d |         | d |
|___|         |___|
```

Die Grammatik sieht wie folgt aus:

$$\begin{aligned} exp &\to exp + num | num \\ num &\to num - d | d \\ d &\to 0 | \ldots | 9 \end{aligned}$$

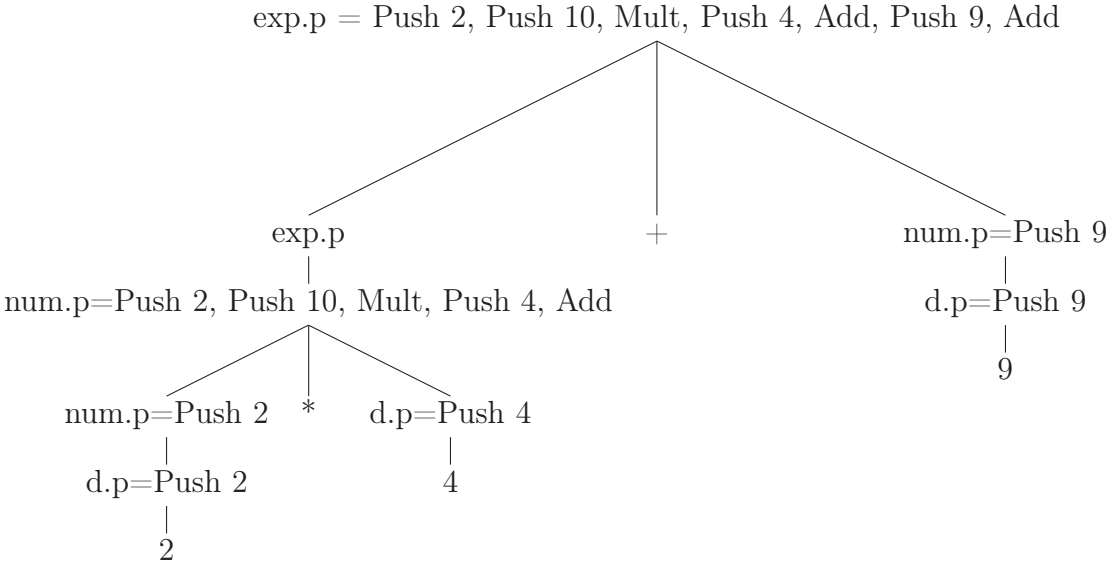
Beispiel Der Parsebaum für $24 + 9$ sieht so aus:



Die syntaxgerichtete Übersetzung hat nun folgende Form (|| steht für die Konkatenation):

Produktion	Semantische Regeln
$exp \to exp_1 + num$	$exp.p = exp_1.p \mid\mid num.p \mid\mid Add$
$exp \to num$	$exp.p = num.p$
$num \to num_1\ d$	$num.p = num_1.p \mid\mid Push\ 10 \mid\mid Mult \mid\mid d.p \mid\mid Add$
$num \to d$	$num.p = d.p$
$d \to 0$	$d.p = Push\ 0$
...	
$d \to 9$	$d.p = Push\ 9$

Finde zu jedem Symbol ein geeignete endliche Menge von Attributen (Platzhaltern). Definiere die semantischen Regeln, die zu jeder Produktion Berechnungsvorschriften für die Attributwerte sind.

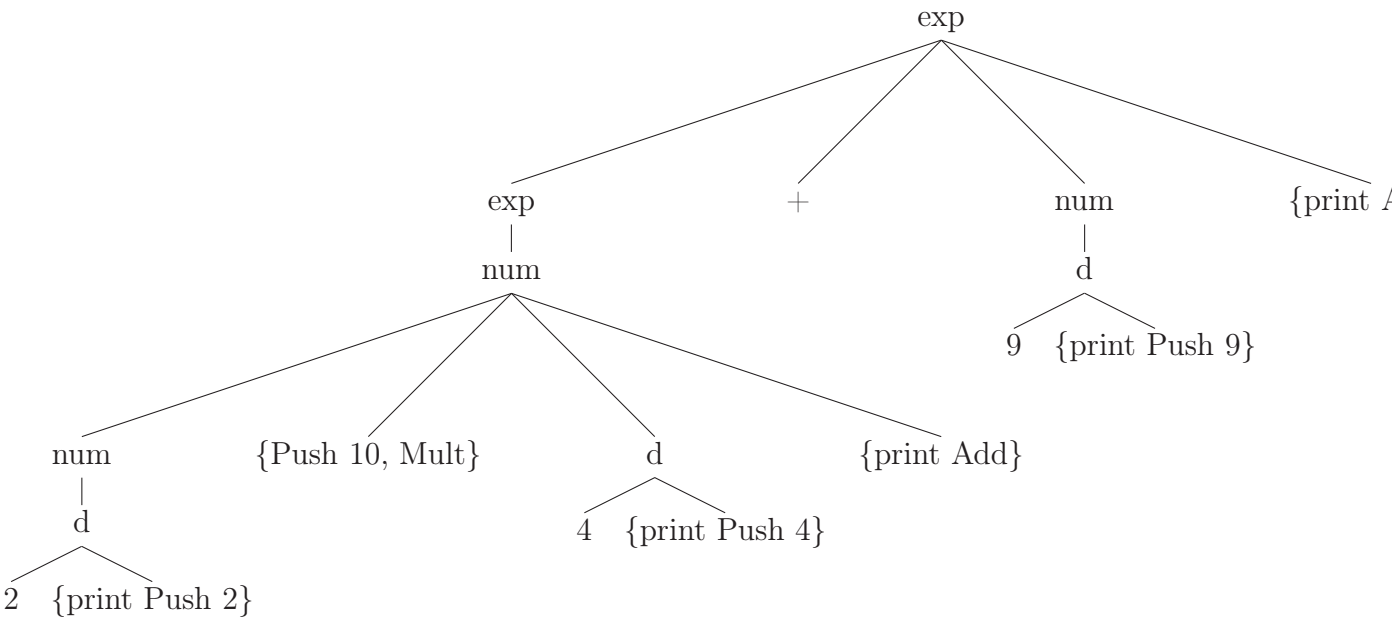


Definition 6. *Eine syntaxgerichtete Definition heißt einfach, wenn Übersetzungen als Attributwerte so erzeugt werden, dass die Übersetzungen der Kinder in der gleichen Reihenfolge wie im Parsebaum Verwendung finden.*

2.2 Syntaxschema

Das Syntaxschema ist immer möglich, wenn eine syntaxgerichtete Definition *einfach* ist. Die Form ist wie folgt: Füge in die verschiedenen Projektionen der Grammatik Code (Psuedoterminale) ein, der ausgeführt wird, wenn die Regel angewendet wird (gemäß depth-first).

$$
\begin{aligned}
exp &\to exp + num\ \{print(Add)\} \\
exp &\to num \\
num &\to num\ \{print(Push10; Mult)\}\ d\{print(Add)\} \\
num &\to d \\
d &\to 0\ \{print(Push\ 0)\} \\
&\dots \\
d &\to 9\ \{print(Push\ 9)\}
\end{aligned}
$$

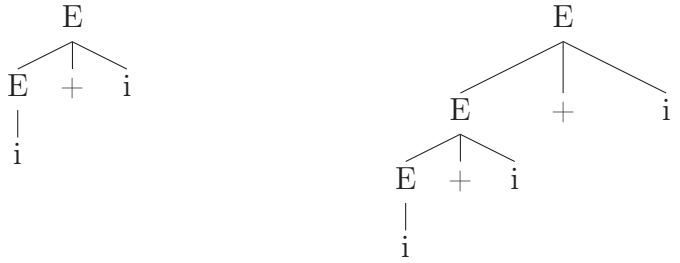


Vorlesung 4, 28. Oktober 2009

2.3 Parser

Problem Gegeben sei eine kontextfreie Grammatik G. Nun ist ein Verfahren aufzustellen, das zu jeder Eingabe $w \in T^*$ ein Parsebaum zu w erzeugt, falls $w \in L(G)$ ist und eine Fehlermeldung produziert, falls $w \notin L(G)$.

Beispiel Die Grammatik G sei $E \to E + i | i$. Die Eingabe sei $i + i$ oder $i + i + i \ldots$.



1. Ausweg Nichtdeterminismus. Man kann durch Backtracking daraus ein Verfahren machen, dies ist aber sehr aufwendig (Laufzeit von $O(n^3)$. Ein Kellerautomat könnte so aussehen, dass unten im Kellerspeicher der Startsymbol liegt und auf dem Eingabeband natürlich die Eingabe. Wie sehen nun die Übergänge aus? Wenn nun ein Nichtterminal A ganz oben im Keller liegt und $A \to \alpha$ liegt in P (der Menge der Produktionen), dann entferne A und schreibe $\bar{\alpha}$ auf den Keller. Wenn ein Terminal $a \in T$ obenauf liegt, dann entferne a vom Keller und rücke den Lesekopf eins vor, falls auf a der Lesekopf aktuell zeigt. Ansonsten soll eine Fehlermeldung geworfen werden. Wenn \$ (das Endsymbol) auf dem Keller liegt und der Lesekopf ebenfalls auf \$ steht, dann wird die Eingabe akzeptiert. Dies ist wie erwähnt recht aufwendig, wie kann man das nun besser bzw. schneller machen?

Idee Die Idee ist nun, dass man die Grammatik G so transformiert, so dass eine deterministische Erkennung möglich wird. Die schlechte Nachricht ist hierbei, das dies nicht immer geht, aber in den meisten Fällen.

Anforderung Zu allen Produktionen der Form $A \to \alpha_1, \ldots, A \to \alpha_n$ gilt $FIRST(\alpha_1)$, $\ldots$, $FIRST(\alpha_n)$ sind wechselseitig disjunkt, mit

$$FIRST(\alpha) = \{a | a \in T, \alpha \to^* aw \wedge w \in (T \cup N)^*\} \cup \{\epsilon | \text{falls } \alpha \to^* \epsilon\}$$

2.4 Prädiktive Syntaxanalyse

Diese Analyse erzeugt einen Parsebaum direkt aus einer gegebenen Grammatik, die der oben genannten Anforderung genügen.
Die Idee ist hierbei, dass Nichtterminale als rekursive Prozeduren betrachtet werden und Terminalsymbole in einem einfachen Matchverfahren behandelt werden.

Beispiel Die Grammatik sei die folgende:

$$stmt \to \underline{exp};\ |\ \underline{if}\ (\underline{exp})\ stmt\ |\ \underline{for}\ (opt; opt; opt)\ start\ |\ \underline{other}$$
$$opt \to \underline{exp}\ |\ \epsilon$$

Eine Eingabe sei $\underline{for}\ (;\underline{exp};\underline{exp})\ \underline{exp};$. Zur passenden Regel gehört der Code:

```
match('for');
match('(');
opt();
match(';');
...
```

In Java könnte das so aussehen:

```
void stmt() {
 switch (lookahead) {
   case exp: {
     match(exp);
     math(';');
     break;
   }
   case if: {
     match(if);
     match('(');
     match(exp);
     match(')');
     stmt();
     break;
   }
}
```

Ein Hindernis ist die Linksrekursion von Parsebäumen der Form $A \to A\alpha$. Die Lösung ist eine systematische Eliminierung. Ein einfacher Fall wäre z.B. $A \to A\alpha|\beta$, mit β beginnt nicht mit A. Mit der nun folgenden abgewandelten Grammatik wird die Linksrekursion verhindert. Mit dieser kann dann eine prädiktive Syntaxanalyse gestartet werden.

$$A \to \beta R$$
$$R \to \alpha R\ |\ \epsilon$$

Wenn man noch mal zum Beispiel vom Anfang zurückkehrt, $(E \to E + i|i)$, könnte man das hier wie folgt umwandeln:

$$E \to iR$$
$$R \to iR \mid \epsilon$$

Vorlesung 5, 29. Oktober 2009

3 Vollständiges Front-End eines Übersetzers

Wir gucken uns ein völlständiges Front-End am Beispiel arithmetischer Ausdrücke an. Wir beziehen uns auf eine Stack-Architektur, damit ein vollständiger Übersetzer vollständig ist. Hier gucken wir uns nur einfache Ausdrücke an, dies ist aber beliebig erweiterbar. Damit ist dann eine zentrale Teilaufgabe jedes Übersetzer gelöst.
Beispielgrammatik G sei

$$\begin{aligned} exp &\to exp + term | exp - term | term \\ term &\to 0 | \ldots | 9 \end{aligned}$$

Die Eingabe sei $9-5+2$. Man bekommt schnell die Einsicht, dass die Prädiktive Syntaxanalyse zu G nicht möglich ist. Hier liegt eine Linksrekursion vor, daher ist es automatisch schon nicht möglich. Als Methode haben wir gesehen, dass man die Linksrekursion eliminieren kann. Die Struktur der Linksrekursion ist hier $A \to A\alpha | A\beta | \gamma$. Ein erzeugtes Wort wäre hier $\gamma(\alpha \backslash \beta)^*$. Wir können uns leicht eine andere Grammatik überlegen, die dasselbe Wort erzeugt:

$$\begin{aligned} A &\to \gamma R \\ R &\to \alpha R | \beta R | \epsilon \end{aligned}$$

Das können wir nun auch mit unserer Beispielgrammatik G machen:

$$\begin{aligned} exp &\to term\ rest \\ rest &\to +term\ rest | -term\ rest | \epsilon \\ term &\to 0 | \ldots | 9 \end{aligned}$$

Jetzt kann die prädiktive Syntaxanalyse gemacht werden

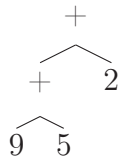
```
void expr() {
  term();
  rest();
}

void rest() {
  if (lookahead == '+') {
```

```
    match('+');
    term();
    rest();
  }
  else if (lookahead == '-') {
    match('-');
    term();
    rest();
  }
}

void term() {
  char t = lookahead;
  if(digit t)
    match(t);
  else
    throw SyntaxError();
}
```

Das Ziel für die semantische Analyse sei die Postfix Notation, die wir erzeugen müssen. Das Erzeugen von Postfixnotation durch synthetisierte Attribute ist nicht möglich. Dazu überlegen wir uns eine abstrakte Syntax, die nur die Operatoren enthält.



Semantische Analyse ? Übersetzung idealerweise aus abstrakter Syntax entwerfen! Hier ist der Entwurf des Übersetzungsschemas möglich.

$$expr \to expr + term\ \{print('+')\} \mid expr - term\ \{print('-')\} \mid term$$
$$term \to 0\ \{print('0')\} \ldots$$

Auch hier müssen wir die Linksrekursion eliminieren.

$$expr \to term\ rest$$
$$rest \to +\{print('+')\}\ rest \mid -\{print('-')\}\ rest \mid \epsilon$$
$$term \to 0\{print('0')\} \ldots$$

Prädiktive Syntaxanalyse erzeugt den Zwischencode als gewünschte Nebenwirkung.

```
void expr() {
  term();
  rest();
}

void rest() {
```

```
  if(lookahead == '+') {
    match('+');
    term();
    print('+');
    rest();
  }
  else if(lookahead == '-') {
    match('-');
    term();
    print('-');
    rest();
  }
}

void term() {
  char t = lookahead;
  match(t);
  print(t);
}
```

3.1 Systematische Optimierung

Als Optimierung kann man die Rekursionen versuchen zu entfernen, in dem man entrekursiert und versucht Schleifen einzuführen. Dies kann man in der Methode `rest()` durchführen. Da `rest()` nur einaml im weiteren Kontext auftritt, ersetzt man man den Methodenaufruf an dieser Stelle durch die Schleife. So spart man einen Methodenaufruf.

Vorlesung 6, 4. November 2009

4 Lexikalische Analyse

Die Aufgabe dieser Analyse ist es ein Quellprogramm, das als Zeichenfolge vorliegt, in eine Tokenfolge zu überführen. Ein Token sei gegeben durch einen eindeutigen Namen („Tag“) und Attributwerte.
Die Eingabe könnte wie folgt aussehen (_ bezeichnet ein Leerzeichen):

```
__count=42_+__a1
```

Die Tokenfolge ergibt dann

$$< \underbrace{\underline{id}}_{Tokenname}, \underbrace{\text{‘count’}}_{Lexeme} > <=> < \underline{num}, 42 > \ldots$$

Die Syntaxanalyse bekommt dann die Tokennamen, die aus der lexikalischen Analyse kommen, als Terminalsymbole.
Es gilt mehrere Teilaufgaben zu bewältigen.

1. Zunächst müssen alle „Whitespaces“ aus dem Quellprogramm entfernt werden.

```
for (;; peak = next input character) {
  if (peak is blank or tab) {
    do nothing;
  }
  else if (peak is newline) {
    line = line + 1;
  }
  else {
    break;
  }
}
```

2. Erkennen natürlicher Zahlen

```
if (peak == digit) {
  v = 0;
```

```
    do {
      v = v * 10 + value of peak;
      peak = next input character;
    } while (peak is digit);
  }
  return <num,v>
```

3. Erkenne Bezeichner (identifier). Hier bestehen Bezeichner aus einer endlichen Anzahl von Buchstaben und Ziffern, wobei er mit einem Buchstaben anfängt. Es gibt natürlich in den Sprachen fest definierte Schlüsselwörter (wie `if`, `for` usw.), die nicht benutzt werden dürfen. Wir legen eine Symboltabelle aller Bezeichner (inkl. Schlüsselwörter) an (Hashtabelle), in der zu jedem Identifier der zugehörige Token gespeichert ist. Diese wird mit `words` bezeichnet.

```
  if(peak is a letter) {
    collect all following letters and digits into buffer b;
    s = content of b;
    w = token returned by words.get(s);
    if (w is not null) {
      return w;
    }
    else {
      enter the key-value pair (s, <id,s>) in words;
    }
  }
```

Ein vollständiger Lexer kann dann aus den Schritten 1. bis 3. konstruiert werden. Dies tun wir in Form einer Methode `scan`, die jeweils den nächsten Token liefert.

```
Token scan() {
  Überspringe whitespaces nach Erstens;
  Behandle Zahlenfolgen nach Zweitens;
  Behandle Bezeichner und Schlüsselwörter nach Drittens;
  Token t = new Token(peak); // Hier ist die Annahme,
    // das der Operator ein Zeichen ist
  peak = ' ';
  return t;
}
```

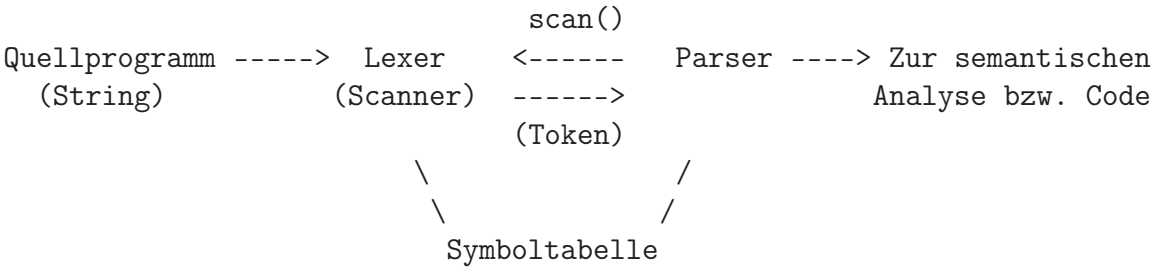
```
    class Token
      int Tag
      /     \
 class Num   class Id
 int num     string lexeme
```

Als Übersetzungsschema ergibt sich

$$
\begin{aligned}
factor &\to expr + term\{print('+')\} \\
term &\to term * factor\{print('*')\} \\
factor &\to num\{print(num.val)\} \\
factor &\to val\{print(id.lexeme)\} \\
factor &\to (expr)
\end{aligned}
$$

Vorlesung 7, 5. November 2009

4.1 Interaktion zwischen Lexer und Parser



Problem Wie geht man mit Gültigkeitsbereiche (scope) von Bezeichnern um. Meist hat man die Konvention der Regel der engsten Umschließung.

$$\begin{aligned} block &\to' \{decks?\ structs'\}' \\ structs &= structs\ struct|\epsilon \\ struct &\to \ldots|block|\ldots \end{aligned}$$

Lösung Der Parser erzeugt eine Baumstruktur der Bezeichner entsprechend der Blockstruktur (lokale Symboltabellen).

4.2 Fehlerbehandlung im Lexer

Man bekommt schnell die Einsicht, dass der Scanner (Lexer) kaum die Chance hat, Schreibfehler zu erkennen (z.B. sowas wie Zeichendreher ...). Ein typisches Beispiel wäre

```
fi (x >=0) {...}
```

Im Lexer wird dies nicht als Fehler erkannt, da es sich bei `fi` um einen gültigen Bezeichner handelt. Erst der Parser bekommt dann hierbei ein Problem. Einige Fehler werden im Lexer gefunden, z.B. Sonderzeichen, die unzulässig sind. Eine typische Fehlerbehandlung ist die *Panische Fortsetzung* (panic reraxy), d.h. man entfernt so lange Zeichen an der verbleibenden Eingabe, bis ein gültiges Lexem gefunden wird. Mögliche Aktionen zur Fehlerbehandlung sind

1. Ein beliebiges Zeichen an der verbleibenden Eingabe löschen

2. Zwei benachbarte Zeichen aus der verbleibenden Eingabe vertauschen.
3. Ein beliebiges Zeichen in der verbleibenden Eingabe einfügen.
4. Ein beliebiges Zeichen in der verbleibenden Eingabe durch ein anderes ersetzten.

Es gilt auch, dass beim Lexerzugriff auf der Festplatte man nur zeichenweise vorgehen kann, was inakzeptabel ist, da zu langsam usw. Man kann nun die Idee verfolgen, dass man blockweise einliest. Dazu benutzt man einen zweigeteilten Puffer. Eine Optimierung wäre folgendes: Vor dem Verschieben ist ein Test auf das Pufferende erforderlich. Kombiniere Test auf Pufferende und den Lesen des aktuellen Zeichens. Dazu setzt man ??zeiger eins weiter (ohne Test). Wenn nun aktives Eingabesymbol `eof` ist, dann ist das Programmende erreicht, ansonsten muss man den anderen Puffer laden.

4.3 Spezifikation von lexikalischen Einheiten (zulässige Lexeme)

Es ist die Frage, wie lexikalische Einheiten spezifiziert sind. Die Formulierung erfolgt durch Muster, gegeben in Form von regulären Ausdrücken.

Operationen und Sprachen

- Vereinigung: $L \cup D = \{w | w \in L\} \cup \{w | w \in D\}$
- Konkatenation: $LD = \{vw | v \in L, w \in D\}$
- Kartesisches Produkt: $L^n = \{w_1 \dots w_n | w \in L\}$
- Folge: $L^* = \bigcup_{i \in \mathbb{N}} L^i$
- nichtleere Folge: $L^+ = \bigcup_{i>0} L^i$

Definition 7. *Reguläre Ausdrücke R_Σ über einem endlichen Alphabet Σ.*

1. $\epsilon \in R_\Sigma$, *wobei* $\epsilon \notin \Sigma$. $L(\epsilon) = \{\epsilon\}$.
2. $a \in R_\Sigma$, *falls* $a \in \Sigma$. $L(a) = \{a\}$.
3. $(A)|(B) \in R_\Sigma$, *falls* $A, B \in R_\Sigma$. $L((A)|(B)) = L(A) \cup L(B)$
4. $(A)(B) \in R_\Sigma$, *falls* $A, B \in R_\Sigma$. $L((A)(B)) = L(A)L(B)$
5. $(A)^* \in R_\Sigma$, *mit* $A \in R_\Sigma$. $L((A)^*) = L(A)^*$

Reguläre Ausdrücke beschreiben lexikalische Muster, die zugehörige Sprache definieren zulässige Lexeme. Klammern kann man nach Prioritäten der Operatoren weglassen. Höchste Priorität haben * und $^+$

Vorlesung 8, 11. November 2009

Begriffsbestimmung

Alphabet Ist eine Menge von Zeichen, Buchstaben (Character). Z.B. das Binäralphabet $\{0, 1\}$ oder der ASCII-Zeichensatz.

Wort (String) Folge von Zeichen (aus dem Alphabet Σ). $w = a_1 \dots a_n$ ist ein Wort über Σ, genau dann wenn $a_i \in \Sigma$ für $1 \leq 1 \leq n$. Die Länge von w wird durch $|w|$ notiert mit $|w| = n$. Die leere Folge wir mit $\epsilon \notin \Sigma$ bezeichnet.

Sprache Eine abzählbare Menge von Wörtern über Σ heißt Sprache. Die Menge ist im Allgemeinen nicht endlich. Die Sprache wird durch Regeln erzeugt.

Die einfachste Sprachklasse sind die regulären Sprachen. Reguläre Ausdrücke definieren reguläre Sprachen. $r \in R_\Sigma$ beschreibt $L(r)$.

Algebraische Gesetze	Erklärung
$r\|s = s\|r$	Vereinigung ist kommutativ
$r\|(s\|t) = (r\|s)\|t$	Konkatenation ist kommutativ
$r(s\|t) = rs\|rt$	Konkatenation ist distributiv bzgl |
$\epsilon r = r = r\epsilon$	ϵ ist neutrales Element bzgl. der Konkatenation
$r^* = (r\|\epsilon)^*$	
$r^{**} = r^*$	

4.4 Reguläre Definitionen

Definition 8. *Sei Σ ein Alphabet und d eine endliche Menge von Namen und $\emptyset \cap \Sigma = \emptyset$. Eine endliche Folge der Form*

$$
\begin{aligned}
d_1 &\to r_1 \\
d_2 &\to r_2 \\
&\vdots \\
d_n &\to r_n
\end{aligned}
$$

heißt reguläre Definition genau dann wenn $d_i \in D$, $r_i \in R_{\Sigma \cup \{d_1, \dots, d_{i-1}\}}$ für alle $1 \leq i \leq n$ und ?

Jede reguläre Definition beschreibt eindeutig eine reguläre Sprache bzw. n reguläre Sprachen, jeweils zu $d_1, \dots, d_n$.

4.5 Unterschiede zur kontextfreien Grammatik

Die kontextfreie Grammatik kann um * erweitert werden. Kontextfreie Grammatik erlaubt direkte und indirekte Rekursion, reguläre Definitionen nicht.

$$d \to r^* \dots$$
$$d \to e \dots$$
$$e \to re|\epsilon$$

Diese Grammatik ist kontextfrei, aber nicht regulär.

Beispiel Bezeichner in C sind alle Wörter über den lateinischen Buchstaben und Ziffern, sowie dem Zeichen _, die nicht mit einer Ziffer beginnen. Wir treffen die Vereinbarung, dass unterstrichene Wörter Namen der Metasprache sind bzgl. der regulären Definition.

$$\begin{aligned} digit &\to 0 \mid 1 \mid \dots \mid 9 \\ digits &\to \underline{digit}\ \underline{digit}^* \\ letter\_ &\to a \mid b \mid \dots \mid A \mid \dots \mid Z \mid \_ \\ id &\to \underline{letter\_}\ (\underline{digit} \mid \underline{letter\_})^* \end{aligned}$$

Wir wollen nun Gleitkommazahlen darstellen. Dafür gibt es mehrere Möglichkeiten. Zum einen kann man sie als Dezimalzahl mit . darstellen. Des weiteren in einer wissenschaftlichen Darstellung, bei der eine Dezimalzahl oder natürliche Zahl gefolgt von E und $opt(+,-)$ gefolgt von einer natürlichen Zahl.

$$\begin{aligned} optFract &\to .digits \mid \epsilon \\ optExp &\to (E(+ \mid - \mid \epsilon)digits) \mid \epsilon \\ nummer &\to digits\ optFract\ optExp \end{aligned}$$

4.6 Erweiterte reguläre Ausdrücke

1. Ein oder mehrere Vorkommen werden durch $^+$ dargestellt. $r^+ = rr^*$
2. Optionales Vorkommen $r? = r \mid \epsilon$
3. Ein aus vielen Zeichen: $[abc] := a \mid b \mid c$. bei kanonischer Reihenfolge $[a_1 - a_n]$.

Dadurch ergibt sich

$$\begin{aligned} digit &\to [0-9] \\ digits &\to digit^+ \\ number &\to digits(.digits)?(E(+ \mid -)?digits)? \end{aligned}$$

Vorlesung 9, 18. Novemer 2009

4.7 Tokenerkennung (endliche Automaten)

Es ist einleuchtend, dass man Sprachen, die durch reguläre Ausdrücke definiert sind, durch endliche Automaten erkennen kann.

Beispiel Wir haben eine pascalähnliche Sprache. In der Grammatik sind die Terminalsymbole unterstrichen.

$$
\begin{aligned}
struct &\rightarrow \underline{if}\ expr\ \underline{then}\ struct \mid \underline{if}\ expr\ \underline{then}\ struct\ \underline{else}\ struct \mid \epsilon \\
expr &\rightarrow term\ \underline{relop}\ term \mid term \\
term &\rightarrow \underline{id} \mid \underline{num}
\end{aligned}
$$

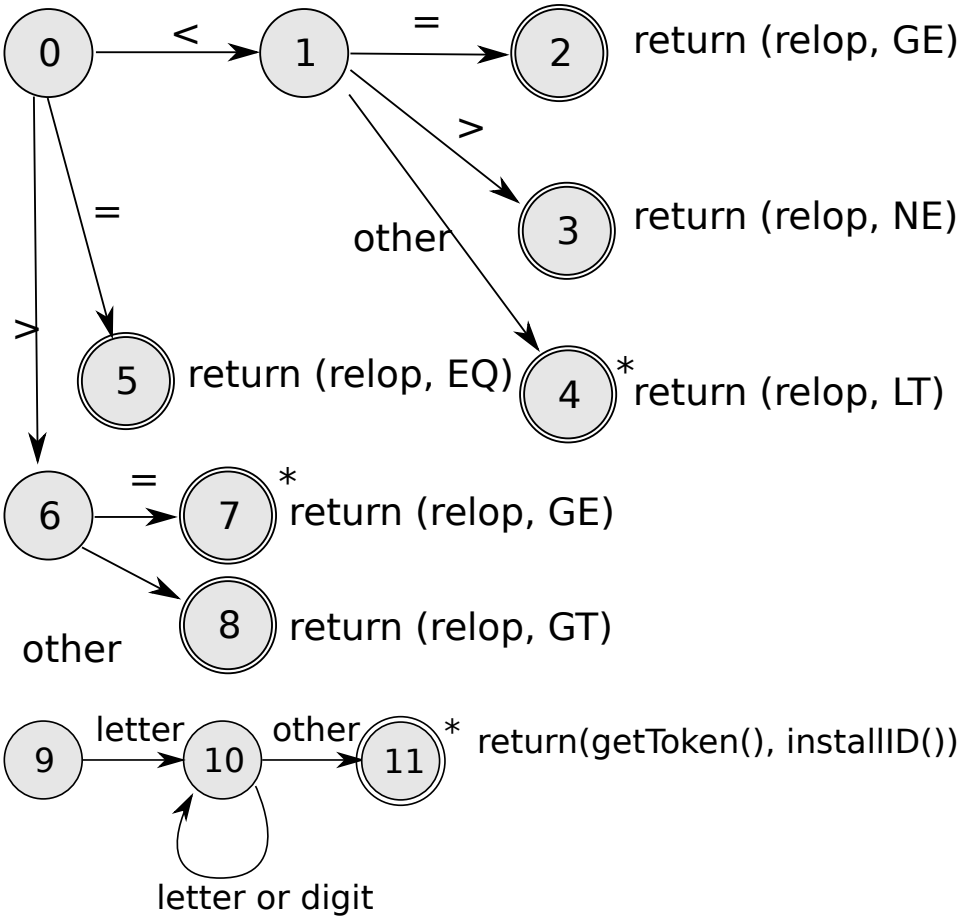
Die reguläre Definition für die lexikalische Struktur sieht wie folgt aus:

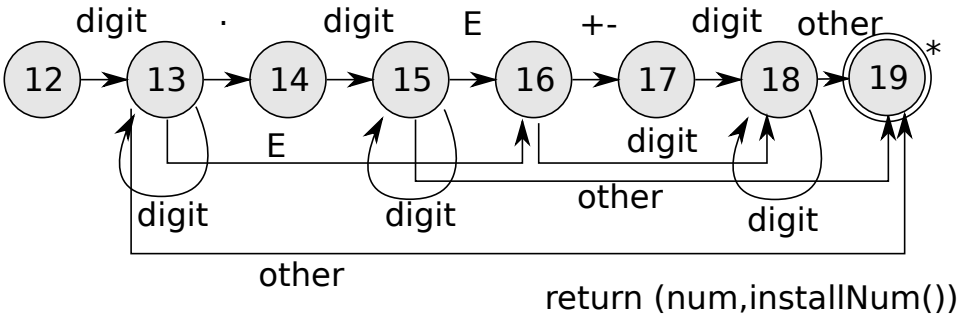
$$
\begin{aligned}
digit &\rightarrow [0-9] \\
digits &\rightarrow digit^{+} \\
\underline{num} &\rightarrow digits\ (.digits)?\ (E[+-]?digits)? \\
letter &\rightarrow [A-Za-z] \\
id &\rightarrow letter\ (letter \mid digit)^{*} \\
\underline{if} &\rightarrow if \\
\underline{then} &\rightarrow then \\
\underline{else} &\rightarrow else \\
\underline{relop} &\rightarrow <|>|<=|>=|<>|=
\end{aligned}
$$

Die Namen der lexikalischen Ebene sind Terminale in der Grammatik (sind zur Verdeutlichung unterstrichen).

Lexem	Tokenname	Attributwert
Whitespace	-	-
if	if	-
then	then	-
else	else	
ein Bezeichner	id	Zeiger in Symboltabelle
eine Zahl	num	Zeiger in Zahltabelle
<	relop	LT
>	relop	GT
<=	relop	LE
>=	relop	GE
<>	relop	NE
=	relop	EQ

Erzeuge zu jedem Tokennamen ein zugehöriges Übergangsdiagramm (also den erkennenden endlichen Automaten), so dass die Zustandsmengen wechselseitig disjunkt sind.





Danke an Lisa für die Bilder!

```
int start = 0; state = 0; Char c;
Token nexttoken() {
  do {
    switch (state) {
      case 0:
        c = next Char();
        while (c == ' ' || c == '\t' || c == '\n') {
          lexemBegin++;
          c = nextChar();
        }
        if ( c == '<') state = 1;
        else if ( c == '=') {
          state = 5;
          return new Token(relop,EQ);
        }
        else if ( c == '>') state = 6;
        else state = fail();
        break;
      ...
      case 8:
        retract(1);
        return new Token(relop,GT)
        break;
    }
  }
}

int fail() {
  forward = lexemBegin;
  switch(start) {
    case 0:
      start = 9; // prüfe auf Bezeichner u. Schlüsselwörter
      break;
```

```
    case 9:
      start = 12; // prüfe auf Zahl
      break;
    default:
      report("Lexikalischer Fehler");
  }
  return start;
}
```

Vorlesung 10, 19. Novemer 2009

4.8 Der Lexer-Generator Lex

(Dieser ist mittlerweile auch frei verfügbar unter dem Namen Flex.) Lex.l sei die Datei, die das Quellprogramm enthält, in der die Struktur der Quellsprache enthalten ist. Dieses wird in den Lex-Compiler geladen. Dieser produziert daraus eine Datei Lex.yyc. In einen normalen C-Compiler kann dies dann wieder reingesteckt werden und man enthält in der generierten a.out ein ausführbares Programm. Wenn man nun ein Quellprogramm durch a.out jagt, dann wird dadurch die Tokenfolge generiert. Im allgemeinen im Kontext der Syntaxanalyse eingesetzt, d.h. eine Funktion `yylex()` ist enthalten, die jeweils das nächste Token liefert.
Vor dem Starten des Lex-Quellprogramms, ist die Tabelle möglicher Lexeme zusammen mit zugehöriger Tokenmenge und Attributwerten aufzustellen.

4.9 Struktur eines Lex-Quellprogramms

```
Deklarationen
%%
Transitionen
%%
Hilfsfunktionen
```

Im Deklarationsteil kann man C-Code ... und reguläre Definitionen. Die Transitionen werden in Form `Muster {Muster}` angegeben. Im letzten Schritt werden dann Angaben zur Verwendung in den Aktionen gemacht.

Beispiel für Lex-Quellprogramm

```
%{ /* Wir erwarten die Verwendung der syntaktischen Konstruktoren
LE,LT, ..., GT, IF, ID, NUM im Parser */
%}
/* Jetzt die regulären Definitionen */
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
```

```
id {letter}({letter} | {digit})*
number {digit}+ (\.{digit}+)?(E[+-]? {digit}+)?
%%
{ws} {/* es ist nichts zu tun bei einem Whitespace*/}
if {return (IF)
else {return (ELSE)}
...
{id} {yyLval = (int) installID(), return (ID);}
{number} {yyLval = (int) installNum(); return (NUM);}
"<" {setval = LT; return (RELOP);}
"<=" ...
...
%%
installID() {...}
installNUM() {...}
```

Der Lookahead-Operator Der Operator wird durch eine Slash / gekennzeichnet. Ein Beispiel zu FORTRAN:

```
IF(1,3)=3 // Feldzugriff
IF(cond)THEN
```

Mit dem Lookahead-Operator kann man dies nun lösen:

```
IF/\(.*\){letter}
```

Vorlesung 11, 25. Novemer 2009

5 Syntaxanalyse (Parser)

Formalisierung der Syntax des Quellprogramms in Form von kontextfreien Grammatiken unterstützt

- das Verständnis der Strukturen
- Präzision und dadurch Fehlererkennung und -behandlung.
- Interpretation und damit Übersetzung

Ziel Strukturanalyse (??) in linearer Zeit. Der erste Ansatz wäre ein nichtdeterministischer Kellerautomat, der allerdings eine Laufzeit von $O(n^3)$ hat, was nicht akzeptabel ist. Also Lösung schränkt man die Menge der Grammatiken so ein, dass nur Gammatiken vorhanden sind, die in Linearzeit zu parsen sind. Diese nennt man LL bzw. LR Grammatiken. LL ist geeignet für ?? Übersetzer, LR ist typisch ??

Vorlesung 12, 26. Novemer 2009

Danke an Lisa fürs zur Verfügungstellen ihrer Mitschrift!!

5.1 Transformationen zur Eliminierung von Nichtdeterminismus

Wir sehen uns nun Transformationen von kontextfreien Grammatiken zur Eliminierung von Nichtdeterminismus an, sodass die Grammatiken von Top-Down-Parsern geparsed werden können.

Eliminierung der Linksrekursion

Definition 9. *Eine Grammatik G heißt linksrekursiv, wenn es eine Herleitung der Form $A \Rightarrow^* A\alpha$ gibt.*

a) Einfacher Fall: Direkte Linksrekursion Als Beispiel betrachten wir

$$A \to A\alpha \mid \beta, \text{ oder allgemein } A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

Jetzt benutzen wir die Transformation $A \to \beta R,\ R \to \alpha R \mid \epsilon$ und können für den allgemeinen Fall schreiben

$$A \to \beta_1 R \mid \ldots \mid \beta_n R$$
$$R \to \alpha_1 R \mid \ldots \mid \alpha_m R \mid \epsilon$$

b) Indirekte Linksrekursion Lässt sich immer eliminieren. Bsp:

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd \mid \epsilon$$

S hat eine indirekte Linksrekursion, das sieht man bei folgender Ableitung.

$$S \to Aa \to Sda$$

Verfahren: Wir nehmen an, dass G keine Zyklen und keine ϵ-Produktionen enthält. Ordne die Nichtterminale der G an: $A_1, \ldots, A_n$. Im Pseudocode sieht das wie folgt aus.

```
for (jedes i von 1 bis n) {
 for (jedes j von 1 bis i-1) {
   ersetze jede Produktion der Form Ai -> Aj gamma durch
   Ai -> delta_1 gamma | .. | delta_n gamma,
   wobei Aj -> delta_1 | .. | delta_n alle Aj-Produktionen sind
 }
 eliminiere die direkte Linksrekursion Ai
```

Beispiel: Ordne S vor A. Äußere Schleife führt zu keiner Transformation, weil S nicht direkt linksrekursiv ist. Der zweite Durchlauf der äußeren Schleife bewirkt die Ersetzung von $A \to S\gamma$, $\eth A \to Sd$ wird durch $A \to Aad \mid bd$ ersetzt. Jetzt haben wir

$$S \to Aa \mid b$$
$$A \to Ac \mid Aad \mid bd \mid e$$

Jetzt müssen wir noch die direkte Linksrekursion in A eliminieren.

$$S \to Aa \mid b$$
$$A \to bdA' \mid A'$$
$$A' \to cA' \mid adA' \mid \epsilon$$

Linksfaktorisierung Entferne gemeinsame Präfixe aus rechten Regelseiten zu jedem Nichtterminal. Unsere erste Einsicht ist, dass $A \to \alpha\beta \mid \alpha\gamma$ mit $\alpha \neq \epsilon$ keine disjunkten Firstmengen hat. Es gilt

$$FIRST(\alpha\beta) \cap FIRST(\alpha\gamma) \neq \emptyset$$

Ersetze durch $A \to \alpha A'$, $A' \to \beta \mid \gamma$. Allgemein bedeutet das, dass man für jedes A den längsten gemeinsamen Präfix α aller rechten Regelseiten zu A auswählt, falls diese ungleich ϵ sind. Dann transformiert man $A \to \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_n$ nach

$$A \to \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_n$$
$$A' \to \beta_1 \mid \ldots \mid \beta_n$$

Nun fährt man mit der Zielgrammatik fort, bis es keine Präfixe mehr gibt.

Erzeuge Übergangsdiagramm für einen prädiktiven Parser Für jedes Nichtterminal gibt es ein Übergangsdiagramm, wobei an den Kanten beliebige Symbole der Grammatik stehen.

```
E  -> TE'        E:  0  -(T)->  1  -(E')->  2
E' -> +TE' | e   E': 3  -(+)->  4  -(T) ->  5 -(E')-> 6,  3 -(e)-> 6
T  -> FT'        T:  7  -(F)->  8  -(T')->  9
T' -> *F | e     T': 10 -(+)-> 11  -(F) -> 12 -(T')-> 13, 10 -(e)-> 13
F  -> (E) | id   F:  14 -(()-> 15  -(E) -> 16 -()) -> 17, 14 -(id)-> 17
```

Jetzt optimieren wir.

```
E': 3 -(+)-> 4, 4 -(T)-> 3, 3 -(e)-> 6
```

Setze neues Übergangsdiagramm E' in E ein:

```
E: 0 -(t)-> 3 -(e)-> 6, 3 -(+)-> 0
```

Vorlesung 13, 2. Dezember 2009

5.2 Deterministische Top-Down-Syntaxanalyse

Eine Grammatik G heißt $LL(k)$, genau dann wenn sich G deterministisch parsen lässt mit einem Lookahead der Länge k.
Was uns interessiert ist ein Verfahren zur Feststellung, ob eine gegebene Grammatik $LL(k)$ (in unserem Fall meist $LL(1)$) ist.

Beispiel Unsere Referenzgrammatik sieht wie folgt aus

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \underline{id}
\end{aligned}
$$

$T = \{+, *, (,), \underline{id}\}$ und $N = \{E, E', T, T', F\}$.
Wie sieht nun ein Top-Down-Parser mit Lookahead von 1 aus?

```
ParseTree tdParser() {
  Token input = nextToken();
  S(); // rufe die Methoden zu S auf
}

void A() {
  wähle eine A-Regel A -> X1 ... Xn aus;
  Erkunde ParseTree entsprechend (depth first);
  for (i = 1 bis n) {
    if (xi in N) Xi(); // rufe Prozedur zu Xi auf
    else if (input == Xi) {
      input = nextToken();
      setze Zeige im ParseTree eins weiter gemäß depth first;
    }
    else Fehlermeldung und -behandlung
  }
}
```

Aufstellen von FIRST- und FOLLOW-Mengen Die FIRST-Menge hatten wir ja wie folgt definiert:

$$FIRST(\alpha) = \{a|\exists\beta\ \alpha \rightarrow^* a\beta\} \cup \{\epsilon|\alpha \rightarrow^* \epsilon\}$$

Die FOLLOW-Menge definieren wir folglich als

$$FOLLOW(A) = \{a|\exists\alpha\beta\ S \rightarrow^* \alpha A a\beta\} \cup \{\$|\exists\alpha\ S \rightarrow^* \alpha A\}$$

Jetzt gucken wir uns ein Verfahren an, wie man die FIRST- und FOLLOW-Mengen aufstellen kann.

1. Stelle FIRST-Menge für alle Symbole der Grammatik auf.
   - $FIRST(a) = \{a\}$ für alle $a \in T$.
   - Für $FIRST(A)$ wende folgende Regeln an, bis nichts mehr hinzukommt: Wenn $A \rightarrow X_1 \dots X_n$ eine Produktion ist mit $n > 1$, dann füge $\{a|\exists i\ a \in FIRST(X_i) \wedge \forall 1 \leq j < i\ \epsilon \in FIRST(X_j)\}$ zu $FIRST(A)$ hinzu.
   - Füge ϵ zu $FIRST(A)$ hinzu, wenn $\forall 1 \leq j \leq n, \epsilon \in FIRST(X_j)$.
2. Stelle FIRST-Mengen für beliebige $X_1 \dots X_n$ auf.
   $FIRST(X_1 \dots X_n) = \{a|\exists i\ a \in FIRST(X_i) \wedge \forall 1 \leq j < i\ \epsilon \in FIRST(X_j)\} \cup \{\epsilon|\forall 1 \leq j \leq n\ \epsilon \in FIRST(X_j)\}$

$FOLLOW(A)$ berechnet sich durch Anwendung folgender Regeln, bis nichts mehr hinzukommt:

1. Füge \$ zu $FOLLOW(S)$ hinzu
2. Füge $FIRST(\beta)\backslash\{\epsilon\}$ zu $FOLLOW(A)$ hinzu, wenn $B \rightarrow \alpha A\beta$ eine Produktion ist
3. Füge $FOLLOW(B)$ zu $FOLLOW(A)$ hinzu, wenn $B \rightarrow \alpha A$ eine Produktion ist oder wenn $B \rightarrow \alpha A\beta$ eine Produktion ist und $\epsilon \in FIRST(\beta)$.

Für unser obiges Beispiel gilt

$$\begin{aligned}
FIRST(F) &= \{(, \underline{id}\} \\
FIRST(T) &= \{(, \underline{id}\} \\
FIRST(E) &= \{(, \underline{id}\} \\
FIRST(E') &= \{+, \epsilon\} \\
FIRST(T') &= \{*, \epsilon\}
\end{aligned}$$

$$\begin{aligned}
FOLLOW(E) &= \{\$,)\} \\
FOLLOW(E') &= \{\$,)\} \\
FOLLOW(T) &= \{+, \$,)\} \\
FOLLOW(T') &= \{+, \$,)\} \\
FOLLOW(F) &= \{*, +, \$,)\}
\end{aligned}$$

Vorlesung 14, 3. Dezember 2009

Definition 10 (LL(1))**.** *Eine Grammatik G heißt LL(1), genau dann wenn zu je zwei Produktionen der Form $A \to \alpha \mid \beta$ gilt*

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. *Wenn $\alpha \to^* \epsilon$, dann gilt $FOLLOW(a) \cap FIRST(\beta) = \emptyset$*

5.3 Aufstellen der Parse-Tabelle

Schreibe Terminalsymbole $\cup\$$ aufs Spaltenarray, Nichtterminale als Zeilen. Trage Produktionen in die Felder von M gemäß folgender Regeln ein. Sei dazu die aktuelle bekannte Produktion $A \to \alpha$.

1. Für jedes $a \in FIRST(\alpha)$ Eintrag von $A \to \alpha$ in Zelle $M[A, a]$
2. Wenn $\epsilon \in FIRST(\alpha)$ und $b \in FOLLOW(A)$, dann Eintrag in $M[A, b]$

	$+$	$*$	$($	$)$	$\underline{id}$	$\$$
E			$E \to TE'$		$E \to TE'$	
E'	$E \to TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to \underline{id}$	

Eine kompaktere Definition für LL(1) lautet so: Eine Grammatik G heißt LL(1), genau dann wenn die Parse-Tabelle zu G an jeder Stelle höchstens einen Eintrag besitzt. Die „Methode Brutale“ entfernt Mehrfacheinträge durch geeignetes Löschen. Dekonstruktive Syntaxanalyse zu $L(G)$ obwohl G mehrdeutig ist?

5.4 Panische Fehlerbehandlung

Die Idee ist die, dass man Synchronisationspunkte festlegt und bei Fehlern ein oder mehrere Symbole entfernt.

1. Trage in de Zellen der FOLLOW-Elemente der einzelnen Nichtterminale das Synkronisationssymbol *sync* ein, wenn die Zelle leer ist. Rücke Eingabe vor bis zum Synchronisationssymbol und entferne Nichttermiale vom Stack.

2. Füge FIRST-Symbole zu A als Synchronisationselemente hinzu. Rücke Eingabe vor, ohne A vom Stack zu entfernen.

3. Wenn ein Terminalsymbol a auf dem Stack liegt, und b auf dem Eingabeband, dann entferne a vom Stack ohne den Zeiger vorzurücken. Das entspricht dann dem Einfügen eines as auf der Eingabe vor dem b (nützlich, wenn der Benutzer z.B. einfach eine Klammer oder sowas vergessen hat).

Die Parse-Tabelle verändert sich dann wie folgt

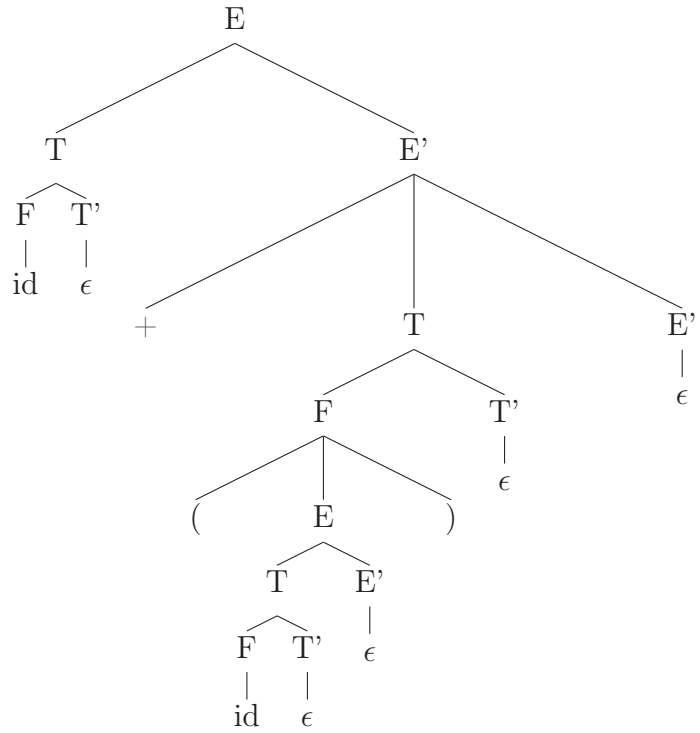
	$+$	$*$	$($	$)$	$\underline{id}$	\$
E			$E \to TE'$	sync	$E \to TE'$	sync
E'	$E \to TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T	sync		$T \to FT'$	sync	$T \to FT'$	sync
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F	sync	sync	$F \to (E)$	sync	$F \to \underline{id}$	sync

Vorlesung 15, 9. Dezember 2009

Beispiel Wir testen die Fehlerbehandlung mit der Eingabe $)\underline{id} + *(\underline{id}\$$.

Stack	Eingabe	Ausgabe
$\$E$	$)\underline{id} + *(\underline{id}\$$	Error! Überflüssiges Symbol wurde entfernt -> Regel 2
$\$E$	$\underline{id} + *(\underline{id}\$$	$E \to TE'$
$\$E'T$	$\underline{id} + *(\underline{id}\$$	$T \to FT'$
$\$E'T'F$	$\underline{id} + *(\underline{id}\$$	$F \to \underline{id}$
$\$FT'\underline{id}$	$\underline{id} + *(\underline{id}\$$	
$\$FT'$	$+ * (\underline{id}\$$	$T' \to \epsilon$
$\$E'$	$+ * (\underline{id}\$$	$E' \to +TE'$
$\$E'T+$	$+ * (\underline{id}\$$	Error! Überflüssiges Symbol $*$ wurde entfernt
$\$E'T$	$(\underline{id}\$$	$T \to FT'$
$\$E'T'F$	$(\underline{id}\$$	$F \to (E)$
$\$E'T')E($	$(\underline{id}\$$	$E \to TE'$
$\$E'T')E'T$	$\underline{id}\$$	$T \to FT'$
$\$E'T')E'T'F$	$\underline{id}\$$	$F \to \underline{id}$
$\$E'T')E'T'\underline{id}$	$\underline{id}\$$	$T' \to \epsilon$
$\$E'T')E'$	$\$$	$E' \to \epsilon$
$\$E'T')$	$\$$	Error!) wird eingefügt
$\$E'T'$	$\$$	$T' \to \epsilon$
$\$E'$	$\$$	$E' \to \epsilon$
$\$$	$\$$	Fertig!

Der Parsebaum sieht wie folgt aus:



Wir haben die Eingabe also interpretiert als $\underline{id} + (\underline{id})$.

5.5 Bottom-Up-Analyse

Analysiere die Eingabe. Suche nach rechten Regelseiten innerhalb der Eingabe. Ersetze gefundene rechte Regelseiten durch das entsprechende Nichtterminal. Dies nennt man „Reduktion“. Fahre dann fort bis man zum Startsymbol gelangt.

Vorlesung 16, 10. Dezember 2009

Wir haben folgende kontextfreie Grammatik G gegeben

$$E \to E + T$$
$$E \to T$$
$$T \to T * F$$
$$T \to F$$
$$F \to (E)$$
$$F \to i$$

Wir betrachten nun die Links- und Rechtsherleitungen des Ausdrucks $i * i$.

Linksherleitung	$E \to T \to T * F \to F * F \to i * F \to i * i$
Rechtsherleitung	$E \to T \to T * F \to T * i \to F * i \to i * i$

Der Parsebaum ist für beide Herleitungen natürlich identisch. Für die Bottom-Up-Analyse benutzen wir einen nichtdeterministischen Kellerautomaten. Wir expandiere die Produktion auf dem Keller. Komplementäre Konstruktion: Reduziere die Eingabe auf das Startsymbol. Für unser Beispiel sieht das folgendermaßen aus: $i * i \to F * i \to T * i \to T * F \to T \to E$.

5.6 Shift-Reduce-Analyse

Dies ist ein nichtdeterministischer Kellerautomat, der auf dem Keller reduziert.

1. Shift: Lies ein Eingabesymbol und schiebe es auf den Keller
2. Reduce: Wenn einige oberste Symbole einer rechten Regelseite entspricht, dann entferne sie vom Stack und push Regelkopf auf Keller
3. Akzeptieren: Wenn Eingabesymbol \$ ist und der Keller gleich $\$S$, dann akzeptiere die Eingabe.
4. Fehler: Wenn auf dem Keller keine rechte Regelseite vorhanden ist, die Eingabe ist leer (\$) und Keller ungleich $\$S$.

?? des Shift-Reduce-Parsers

stack	Eingabe	Aktion, Ausgabe
\$	$i * i\$$	SHIFT
$\$i$	$*i\$$	REDUCE, $F \to i$
$\$F$	$*i\$$	REDUCE, $T \to F$
$\$T$	$*i\$$	SHIFT
$\$T*$	$i\$$	SHIFT
$\$T * i$	\$	REDUCE, $F \to i$
$\$T * F$	\$	REDUCE, $T \to T * F$
$\$T$	\$	REDUCE, $E \to T$
$\$E$	\$	Akzeptieren

Definition 11 (Handle)**.** *Sie $S \to_m^* \alpha Bw \to_m \alpha\beta w$. Die Produktion $B \to \beta$ zusammen mit ihrer Position nach α in der Rechtssatzform $\alpha\beta w$ heißt Handle.*

Lemma Wenn eine Grammatik G eindeutig ist, dann gibt es zu jeder Rechtssatzform genau einen Handle.

Handle-Reduktion (Handle-Pruning, Handle-Stutzung) Die ist die allgemeine Methode eine inverse Rechtsherleitung durch Reduktion aufzubauen. Sei $w \in L(G)$. Es gibt eine Rechtsherleitung $S = \gamma_1 \to \gamma_1 \ldots \to ??w$

Definition 12 (Gültige Vorsilben)**.** *Sei G eine Grammatik und αAw eine Rechtsatzform. Sei $A \to \beta$ und $\alpha\beta w$ mit $A \to \beta$ nach α ist Handle von $\alpha\beta w$. Alle Präfixe von β heißen gültige Vorsilben.*

Definition 13 (Item)**.** *Jede Produktion von G, bei der ein Punkt in die rechte Regelseite eingefügt wurde, heißt Item bezüglich G. Dabei gilt, dass $. \notin T$. Nach der Konvention werden Items in der Form $[A \to .X_1 \ldots X_n]$, $[A \to X_1.X_2 \ldots X_n]$, $\ldots [A \to X_1 \ldots X_n.]$ dargestellt.*

Als Konvention zur Darstellung von Item-Mengen einigen wir uns darauf, dass alle Produktionen der Grammatik nummeriert werden. Der Handle (n, k) entspricht dem Item, der der n-ten Produktion und dem Punkt an der Position k entspricht.

Vorlesung 17, 16. Dezember 2009

5.7 Einfache LR-Syntaxanalyse

(auch bekannt unter Simple LR-Parser, SLR-Parser)
Wir müssen zunächst etwas Vorarbeit leisten. Sei G eine kontextfreie Grammatik, dann erweitern wir G zu G' durch Hinzunahme von $S' \to S$. Zur Steuerung der Shift-Reduce-Analyse konstruieren wir einen endlichen Automaten, den man auch $LR(0)$-Automaten nennt, zu G': Die Zustandsmenge $C = \{I_0, \ldots, I_n\}$ ist die kanonische Sammlung von Item-Mengen für G', die sich wie folgt generieren lässt:

1. $I_0 := Closure(\{[S' \to .S]\})$. Zu Eintrag $Closure(I) = \{[A \to .\alpha] \mid [B \to \gamma.A\beta] \in I$ und $A \to \alpha$ ist Produktion $\}$
2. Füge den Item-Mengen $I_0, I_1, \ldots, I_k \in C$ für jedes $0 \leq i \leq k$ und jedes Grammatiksymbol $X \in N \cup T$ die Item-Menge $Goto(I_i, X)$ zu C hinzu, falls diese Menge nicht leer ist und nicht schon in C enthalten ist.

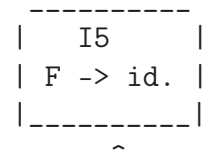
Definition 14 (Funktion Goto)**.** *Sei I eine Item-Menge und $X \in N \cup T$.*

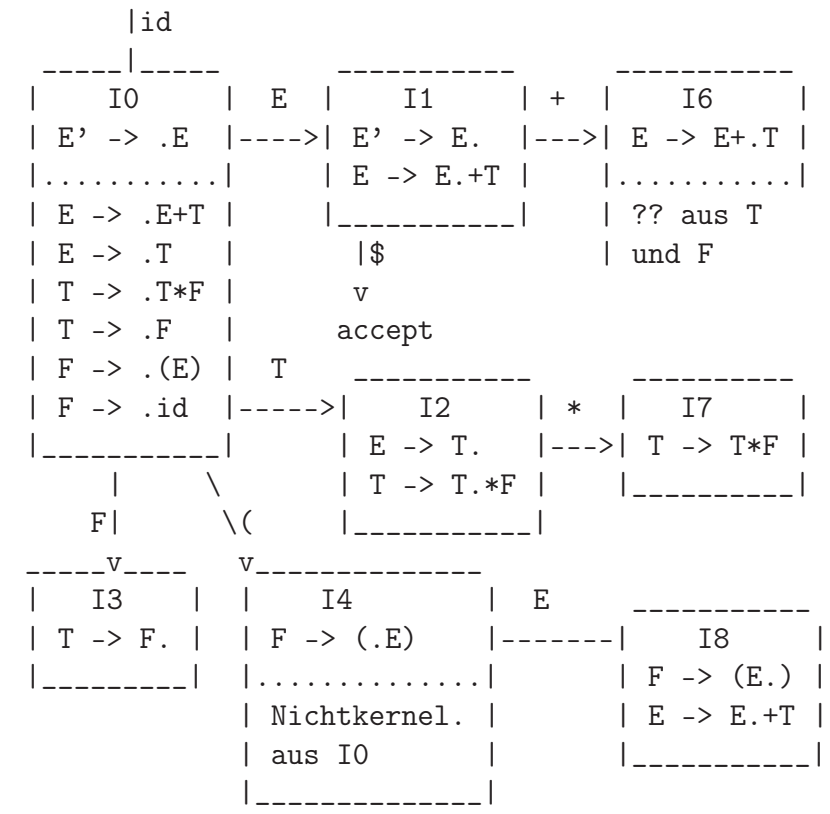
$$Goto(I, X) := Closure(\{[A \to \alpha X.\beta] \mid [A \to \alpha.X\beta] \in I\})$$

Eine intuitive Erklärung dafür wäre, dass die Goto-Funktion den Zustandsübergang beschreibt, wenn X als Eingabesymbol gelesen wurde (shift) bzw. wenn eine rechte Regelseite γ auf X reduziert wurde (reduce).

Beispiel

$$E' \to E$$
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \underline{id}$$




id
I0
E’ -> .E
E -> .E+T
E -> .T
T -> .T*F
T -> .F
F -> .(E)
F -> .id
E
I1
E’ -> E.
E -> E.+T
$
accept
+
I6
E -> E+.T
?? aus T
und F
T
I2
E -> T.
T -> T.*F
*
I7
T -> T*F
F
(
I3
T -> F.
I4
F -> (.E)
Nichtkernel.
aus I0
E
I8
F -> (E.)
E -> E.+T

Vorlesung 18, 17. Dezember 2009

Jedem Zustand i ist eindeutig ein Grammatiksymbol zugeordnet. $g := Goto(I_i, X)$. In der SLR-Syntaxanalyse hat man eine zweigeteilte Parse-Tabelle, bestehend aus einem ACTION und dem GOTO-Teil.

5.8 Struktur der Parse-Tabelle

1. Die ACTION-Komponente hat zu jedem LR(0)-Automatenzustand eine Zeile und zu jedem Terminalsymbol sowie \$ eine Spalte. Jeder Eintrag $ACTION[i, a]$ hat eine der folgenden Formen:
   a) sj mit der Bedeutung Shift Zustand j auf den Stack
   b) rj mit der Bedeutung Reduce $A \to \alpha$, wenn diese Produktion die Nummer j trägt. Der Effekt hierbei ist, dass k Ziffern vom Stack entfernt werden, wobei α die Länge k hat. Dann legt man $GOTO(m, A)$ auf den Stack, wenn in die oberste Zahl nicht Entfernt war.
   c) *Accept*
   d) *Error*

Die Parserkonfigurationen werden in der Form $0s1 \dots sr, aj \dots a_k\$$ notiert (Rechtssatzform).

	id	+	*	(	)	\$	E	T	F
0	sS			s4			1	2	3
1		s6				*acc*			
2			s7						
3									
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9			s7						
10									
11									

5.9 Aufbau der Parse-Tabelle

1. Konstuiere die LR(0)-Zustandmenge $I_0, \ldots, I_n$
2. Zu jedem Terminalsymbol u:
   a) Wenn $[A \to \alpha.a\beta] \in I_i$ und $Goto(I_i, a) = Ij$, dan trage sj in $ACTION[i, a]$ ein.
   b) Wenn $[A \to \alpha.] \in I_i$ mit $A \to \alpha$ ist die k-te Produktion, dann trage rk in $ACTION[i, a]$ für alle $a \in FOLLOW(A)$ ein.

Definition 15. *Eine Grammatik G heißt SLR-Grammatik genau dann wenn die Einträge nach Regel 2 keine Kollisionen hervorrufen.*

Vorlesung 19, 6. Januar 2010

6 Semantische Analyse

Es sollen kontextabhängige Informationen festgestellt und dargestellt werden. Nach der semantischen Analyse endet die so genannte Analysephase. Allerdings ist dies noch nicht das Ende des Frontends, dazu gehört nämlich noch die Zwischencode-Erzeugung.

6.1 Syntaxgerichtete Definitionen

(Syntax Directed Definitions - SDD)

Synthetisierung von Definitionen zu Teilbäumen aus den Definitionen der zugehörigen Unterbäume.

Aufgaben des Übersetzerbauers

1. Finde geeignete Attribute für die Symbole der Grammatik (z.B. typ, code ...) sowie zugehörige Berechnungs?? für die Werte der Attribute.
2. ??

Produktion	semantische Regel
$E \to E_1 + T$	$E.code = E_1.code \mid\mid T.code \mid\mid '+'$

Definition 16 (SDD)**.** *Eine SDD ist eine kontextfreie Grammatik zusammen mit endlich vielen Attributen und semantischen Regeln, wobei Attribute den Symbolen der Grammatik zugeordnet sind und semantische Regeln den Produktionen.*

Synthetisierte und ererbte Attribute Ein Attribut heißt *synthetisiert*, wenn die zugehörigen semantischen Regeln nur auf Attributwerte der „Kinderknoten“ oder auf Attributwerte des Knotens selbst zugreifen.
Ein Attribut heißt *ererbt*, genau dann wenn die zugehörigen semantischen Regeln auf „Geschwisterknoten“, den „Vaterknoten“ oder auf eigene Attribute zugreifen.
Ein Beispiel: $X_2.a = X_1.b * 3 + X_2.c$

Produktion	semantische Regel
$F \to F_1 * digit$	$F.val = F_1.val * digit.lexval$
$F \to digit$	$F.val = digit.lexval$

Beispiel für Bedarf ererbter Attribute

Produktion	semantische Regeln
$F \to digit\ F'$	$F'.inh = digit.lexval$; $F.val = F'.syn$
$F' \to *\ digit\ F'$	$F'_1.inh = F'.inh * digit.laxval$; $F'.syn = F'_1.syn$
$F' \to \epsilon$	$F'.syn = F'.inh$

Im Parsebaum an der Stelle, wo man die Regel $F' \to *digitF'$ benutzt, muss man ein ererbtes Attribut verwenden.

Vorlesung 20, 7. Januar 2010

Definition 17 (Abhängigkeitsgraph)**.** *Gegeben sei eine SDD und ein zugehöriger Parsebaum P.*

1. *Zu jedem Knoten aus P mit Markierung X und zugeordnetem Attribut a erzeuge einen Knoten $X.a$ im Abhängigkeitsgraphen A_p.*
2. *Wenn eine Regel zur Berechnung on $X.a$ den Wert $Y.b$ benötigt, dann füge eine Kante von $Y.b$ nach $X.a$ in A_p ein. Achtung: Vorkommen unterscheiden!*

6.2 Topologische Sortierung

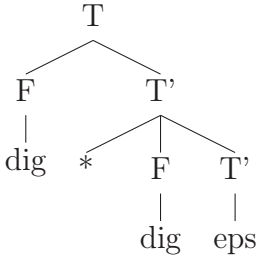
Wenn A_p keinen Zyklus enthält, dann lassen sich seine Knoten $N_0, \dots N_k$ derart sortieren, dass folgende Eigenschaft erfüllt ist: Wenn es eine Kante von N_i nach N_j gibt, dann gilt $i < j$. Eine solche Sortierung heißt topologische Sortierung.

Lemma 1. *Wenn A_p einen Zyklus enthält, dann gibt es keine topologische Sortierung.*

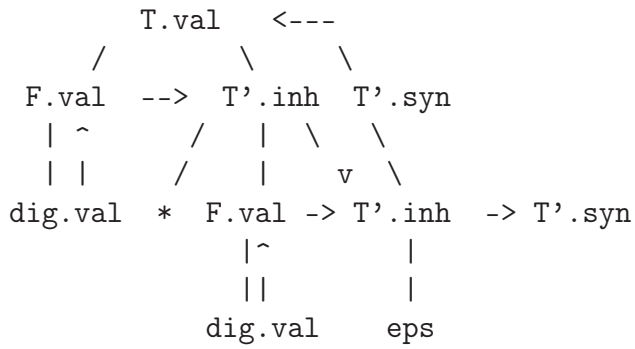
Konstruktion Wenn A_p keine Zyklen enthält, dann gibt es mindestens einen Knoten in A_p, der keine Eingangskante besitzt. Nimm einen solchen Knoten als jeweils nächsten und entferne ihn zusammen mit seinen ausgehenden Kanten aus A_p und fahre fort.
Eine topologische Sortierung liefert eine lineare Attributierungsmöglichkeit in P.
Wir haben folgende SDD gegeben

Produktion	semantische Regel
$T \to FT'$	$T.val = T'.syn$, $T'.inh = F.val$
$T' \to *FT'_1$	$T'_1.inh = T'.inh * F.val$, $T'.syn = T'_1.syn$
$T' \to \epsilon$	$T'.syn = T'.inh$
$F \to digit$	$F.val = digit.lexval$

Für $3 * 5$ sieht der Parsebaum so aus:



Jetzt bauen wir den Abhängigkeitsgraphen (überlagt den Parsebaum)



S-Attributierung Eine SDD heißt S-Attributierung, genau dann wenn alle Attribute synthetisiert sind.

Lemma 2. *Eine S-Attributierung kann wie bei der Bottom-Up-Analyse erstellt werden.*

Definition 18 (L-Attributierung)**.** *Eine SDD heißt L-Attributierung, wenn jedes Attribut entweder synhetisiert ist oder wenn es ererbt ist, nur folgende Eigenschaften besitzt: Sei $A \to X_1 \dots X_n$ die Regel und $X_i.a$ das benutzte Attribut, dann darf zur Berechnung von $X_i.a$ nur erstens auf ererbte Attribute von A zugegriffen werden und zweitens nur auf Attribute von $X_1, \dots, X_i$ zugegriffen werden, wobei innerhalb von X_i keine lokalen Zyklen entstehen dürfen.*

6.3 Syntaxgerichtete Definition mit Nebenwirkungen (side effects)

Typzuweisung an Variablen Der Symboltabelle der lexikalischen Analyse will man den Variablen nun Typen zuweisen.

Produktion	semantische Regeln
$D \to \underline{int}\ L$	$L.typ = int$
$D \to \underline{float}\ L$	$L.typ = float$
$L \to \underline{id}, L$	$L_1.typ = L.typ$; $addType(id.entry, L.typ)$
$L \to \underline{id}$	$addType(id.entry, L.typ)$

Achtung Ausführungsreihenfolge kann zu Inkonsistenzen führen.
Man hat nun zwei Möglichkeiten, wie man das auflösen kann.

1. Durch Festlegung der Reihenfoge z.B. depth-first, links nach rechts, post order usw.

2. Sorgfältige Analyse bzgl. zulässiger Reihenfolge

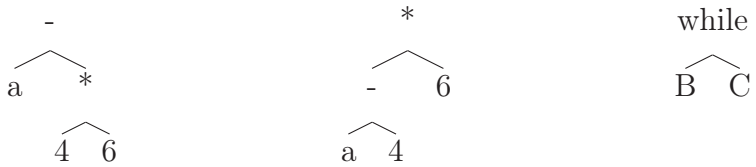
Vorlesung 21, 13. Januar 2010

6.4 Weitere Anwendungen syntaxgerichteter Definitionen

Heute gucken wir uns die Entwicklung von Syntaxbäumen an. Dies ist eine kompakte Darstellung der abstrakten Syntax.

Definition 19. *Ein Quellprogramm wird durch einen (abstrakten) Syntaxbaum repräsentiert, wobei syntaktischer Zucker ignoriert wird. In der Wurzel hat man Zugriff aus das Gesamtprogramm und induktiv repräsentieren in jedem inneren Knoten K die Kinder aller unmittelbaren, wesentlichen Komponenten von K.*

Beispiele $a - 4 * 6$, $(a - 4) * 6$, $\underline{while}\ B\ \underline{do}\ C$



Je nach Grammatik entspricht der Parsebaum besser oder schlechter dem abstrakten Syntaxbaum. Syntaxgerichete Definition zur Erzeugung des abstrakten Syntaxbaums geeignet für Bottom-Up-Analyse. Implementierung, Konstruktorfunktionen, Leaf und Node jeweils mit ?? Stelligkeiten.

Produktion	semantische Regeln
$E \to E_1 + T$	$E.t = new\ Node('+', E_1.t, T.t)$
$E \to E_1 - T$	$E.t = new\ Node('-', E_1.t, T.t)$
$E \to T$	$E.t = T.t$
$T \to \underline{id}$	$T.t = new\ Leaf(\underline{id}, \underline{id}.entry)$
$T \to \underline{num}$	$T.t = new\ Leaf(\underline{num}, \underline{num}.val)$
$T \to (E)$	$T.t = E.t$

Die S-Attributierung liefert hier bei der Bottom-Up-Analyse den abstrakten Syntaxbaum.

Produktion	semantische Regeln
$E \to T\ E'$	$E'.i = T.t;\ E.t = E'.s$
$E' \to +T\ E'_1$	$E'_1.i = new\ Node('+', E'.iT.t);\ E'.s = E'_1.s$
$E' \to -T\ E'_1$	$E'_1.i = new\ Node('-', E'.i, T.t)$
$E' \to \epsilon$	$E'.s = E'.i$
$T \to \underline{id}$	$T.t = new\ Leaf(\underline{id}, id.entry)$
$T \to \underline{num}$	$T.t = new\ Leaf(\underline{num}, num.val)$
$T \to (E)$	$T.t = E.t$

Die L-Attributierung für die Top-Down-Analyse erzeugt denselben abstrakten Syntaxbaum. Syntaktische Strukturen können bereits im Sprachentwurf schlecht sein. Ein Beispiel wären Felder über Felder über ... In C schreibt man `int[2][3]`. Die Syntax wäre dazu wie folgt:

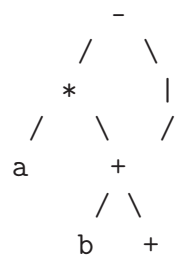
$T \to BC$
$C \to [num]C$
$C \to \epsilon$
$B \to \underline{int}$
$B \to \underline{float}$

Vorlesung 22, 14. Januar 2010

6.5 Verbesserung der Implementierung der abstrakten Syntax

Gerichtete azyklische Graphen (engl. DAG)

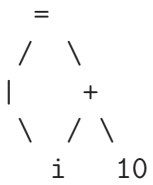
Idee Operatorbäume aus konkreter Syntax $\underline{if}\ B\ \underline{then}\ C_1\ \underline{else}\ C_2$. Gemeinsame Unterstrukturen verwenden, wie z.B. hier bei $a * (b + 4) - (b + 4)$. Dies soll dargestellt werden als



Für die Code-Erzeugung ist dies unproblematisch.
Zur Berechnung von Werten (Auswertung von Ausdürcken) muss auf Nebenwirkungen geachtet werden.

Vorgehensweise Anstelle von `new Leaf` und `new Node` werden Funktionen `mkLeaf` bzw. `mkNode` verwendet, die vor Erzeugung eines neuen Knotens testen, ob ein solcher nicht schon existiert.

Implementierung Stelle Blätter und ihre Knoten als Records dar und organisiere diese in einem Array. Verwende die Indizes der Arrays zur Referenzierung der Knoten.
Ein Beispiel: $i = i + 10$



1	$\underline{id}$	27
2	$\underline{num}$	10
3	$'+'$	1 2
4	$'='$	1 3
	$\vdots$	

Produktion	semantische Regeln
$A \to L = E$	$A.n = mkNode('=', L.n, E.n)$
$E \to E_1 + T$	$E.n = mkNode('+', E_1.n, T.n)$
$E \to T$	$E.n = T.n$
$T \to \underline{num}$	$T.n = mkLeaf(\underline{num}, num.val)$
$T \to \underline{id}$	$T.n = mkLeaf(\underline{id}, id.entry)$
$T \to (E)$	$T.n = E.n$
$L \to \underline{id}$	$L.n = mkLeaf(\underline{id}, id.entry)$

6.6 Übersetzungselemente

Man will semantische Aktionen in die rechten Regelseiten einfügen. Wir gewinnen die Einsicht, dass zu jedem SDD man eine ?? Übersetzungselement finden kann.

1. Fall Die Grammatik sei LR und S-attributiert. Dann konstruiere äquivalentes Übersetzungsschema.

Produktion	Semantische Regeln
$E \to E_1 + T$	$\{E.val = E_1.val + T_1.val\}$
$E \to T$	$\{E.val = T.val\}$
$T \to d$	$\{T.val = d.lexval\}$
$T \to (E)$	$\{T.val = E.val\}$
$L \to E.n$	$\{print(E.val)\}$

Das ist jetzt der Übergang zum Üersetzungsschema SDT. Verbindung zur Syntax-Analyse wichtig! Bezug zu Vorkommen der Nichtterminale.
Verwalte im Stack die Symbole zusammen mit den Attributen.

Vorlesung 23, 20. Januar 2010

7 Zwischencode-Erzeugung

Es gibt nicht den Zwischencode, allerdings Standards, z.B. Drei-Adress-Code, der sich für große Klassen von Quellprogrammen und Zielarchitekturen eignet.
Der erste C++-Compiler erstellte so C-Code als Zwischencode und nutze dann den C-Compiler für das Back-End.

7.1 Drei-Adress-Code

Die Idee ist hierbei, dass jeder Befehl (Instruktion, Anweisung) aus jeweils einem Operator, maximal zwei Operanden und einem Register besteht. Es gibt drei Adressarten:

1. Namen (Bezeichner aus dem Quellprogramm, meist gegeben durch Verweis in die Symboltabelle)
2. Konstanten (i.a. Familie)
3. Hilfsvariablen (temporäre Adressen), z.B. zur Auflösung zwischen arithmetischen Ausdrücken

Befehlssatz gegeben durch endlich viele Befehlsketten.

1. Wertzuweisungen der Form $x = y\ op\ z$, wobei x eine Variable ist, y und z beliebige Anweisungen und op ein zweistelliger arithmetischer oder boolscher Operator
2. Werzuweisungen der Form $x = op\ y$, wobei hier op ein einstelliger Operator ist.
3. Wertzuweisungen der Form $x = y$
4. Unbedingte Sprünge der Form $goto\ x$, wobei x eine beliebige Adresse ist
5. Bedingte Sprünge der Form $if\ x\ goto\ L$, wobei L eine Zahl bzw. Adresse ist
6. Bedingte Sprünge der Form $if\ False\ x\ goto\ L$
7. Bedingte Sprünge der Form $if\ x\ relop\ y\ goto\ L$ und $relop$ ein Vergleichsoperator
8. $param\ x$
9. $call\ p,\ n$
10. $y = call\ p,\ n$
11. $return\ y$

12. *return*
13. Befehle der Form $x = y[i]$, wobei x, y Variablen sind und i eine Adresse
14. Befehle der Form $x[i] = y$
15. Befehle der Form $x = \&y$, wobei x und y Variablen sind. x enthält den L-Wert von y, d.h. die Adresse die zu y gehört.
16. Befehle der Form $x = *y$

7.2 Darstellung von Drei-Adress-Code

Beispiel $a = b * -c + b * -c$

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
```

Quadrupel-Darstellung

	op	arg1	arg2	result
0	minus	c		t1
1	*	b	t1	t2
2	minus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5

Tripel-Darstellung Anstatt der Result-Spalte der Quadrupel referenziert man andere Zeilen in der Tabelle.

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)

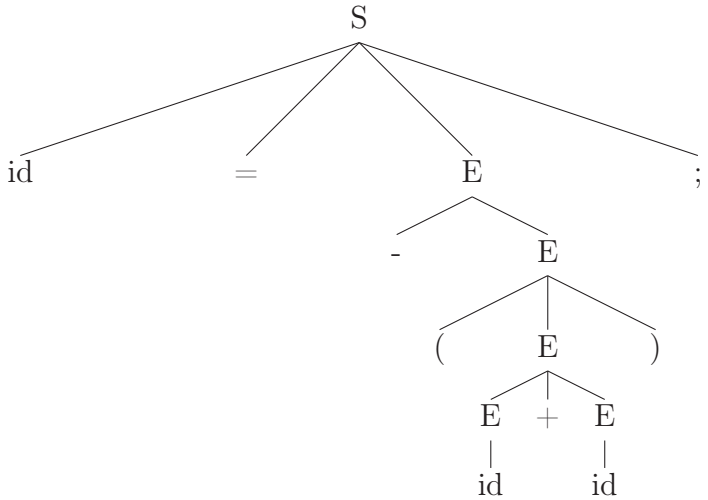
Vorlesung 24, 21. Januar 2010

7.3 Erzeugung von Drei-Adress-Code für eine imperative Programmiersprache

Sei `new T()` eine Funktion, die bei jedem Aufruf eine neue Hilfsvariable erzeugt, z.B. $t_1, t_2 \ldots$. Welche zu E ?? Attribute, `addr` und `code`, wobei der Wert von `addr` diejenige Adresse sein soll, unter der der Wert des Ausdrucks E abgelegt wird und der Wert von `code` zur Berechnung des Werts von E direkt? Zur Darstellung von Drei-Address-Code verwendet wir Strings, mit `||` bezeichnen wir die Konkatenation. Wir verwenden auch Quotierungen z.B. `E.code = new T();`. `E1.code || E.addr = x+ y || E.code`. Alternativ ist die Verwendung Konstruktionsfunktion `gen(E.addr '=' x '+' y)`.
Syntaxgerichtete Definition für einfache Wertzuweisungen der Form $S \to id = E;$.

Produktion	semantische Regeln
$S \to id = E$	S.code = E.code || id.lexeme = E.addr
$E \to E_1 + E_2$	E.addr = new T(); E.code = E1.code || E2.code || E.addr = E1.addr + E2.addr
$E \to id$	E.addr = id.lexeme; E.code = ”
$E \to -E_1$	E.addr = new T(); E.code = E1.code || E.addr = minus E1.addr
$E \to -(E_1)$	E.addr = E1.addr; E.code = E1.code

Beispiel $X = -(y + z)$. Wir betrachten nun den Parsebaum.



Die Attributierung liefert hier dann das Drei-Adress-Code-Programm

```
t0 = y + z;
t1 = minus t0;
tx = t1;
```

Das können wir natürlich auch über ein Übersetzungsschema erzeugen lassen.

$$S \to id = E\{println(id.L = E.addr);\}$$
$$E \to E_1 + E_1\{E.addr = newT(); println(E.addr = E_1.code + E_2.code);\}$$
$$E \to id\{E.addr = id.lexeme\}$$
$$E \to -E_1\{E.addr = newT(); println(E.addr = minusE_1.addr;\}$$
$$E \to (E_1)\{E.addr = E_1.addr\}$$

7.4 Syntaxgerichtete Definition für Konstrollstrukturen

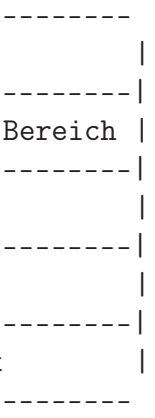
Zu S wählen wir zwei Attribute: `code` und `next`, wobei `next` ein Label ist, mit dem wir die Stelle im Programm hinter S markierten. Sei `new Label()` eine Funktion, die jeweils neue Label liefert.

Produktion	semantische Regeln
$P \to S$	S.next = new L(); P.code = S.code || Label S.rest: Stop
$S \to S_1 S_2$	S1.next = S.next; S2.next = new L(); S.code = S1.code || Label S1.next: S2.code

Vorlesung 25, 27. Januar 2010

8 Laufzeitumgebung

Wir schauen uns das ganze auf einer eher konzeptionellen Ebene an.
Wir müssen überlegen, wie man den Speicher verwaltet, d.h. wie man abstrakte Strukturen auf Speicherplätze abbildet. Wir benutzen einen logischen Adressraum, d.h. eine Zuordnung zu physikalischen Adressraum, die durch das Betriebssystem geschieht.
Den logischen Adressraum teilen wir in fünf große Bereiche auf. In den ersten Block packen wir den ganzen Code rein. In den zweiten Block lassen wir hinreichend viel Platz für statische Variablen, Objekte usw. Dieser Bereich kann zur Compilierzeit in der Größe festgelegt werden. Im dritten Segment findet der Heap, d.h. der dynamische Speicherbereich platz. Der Heap wächst zur Laufzeit nach unten. Im fünften Segment lassen wir Platz für den Stack. Auch dies ist ein dynamischer Bereich, der nach oben hin wächst. Auf dem Stack werden die Prozeduraufrufe verwaltet.



Man vermischt Heap und Stack nicht, da es ohnehin schwierig ist nicht mehr benötigten Speicherplatz auf dem Heap wieder freizugeben (Stichwort Garbage Collection). Durch eine Vermischung wäre dies noch schwieriger. Beim Stack hingegen funktioniert dies ganz strukturiert, sobald eine Prozedur fertig ist, kann der zugehörige Platz freigegeben werden.
Ein Aktivierungssegment ist Speicher, der beim Aufruf einer Prozedur benötigt wird und beim Beenden kann er wieder freigegeben werden. Mit Aktivierungsbäume kann der Ablauf von Prozeduraufrufen dargestellt werden. Die Funktion `main` bildet dabei die Wurzel und jeder Knoten, der beim Ablauf Prozeduren erstellt, hat entsprechend dann wieder Kinder.

Beispiel Quicksort für Arrays der Länge 11.

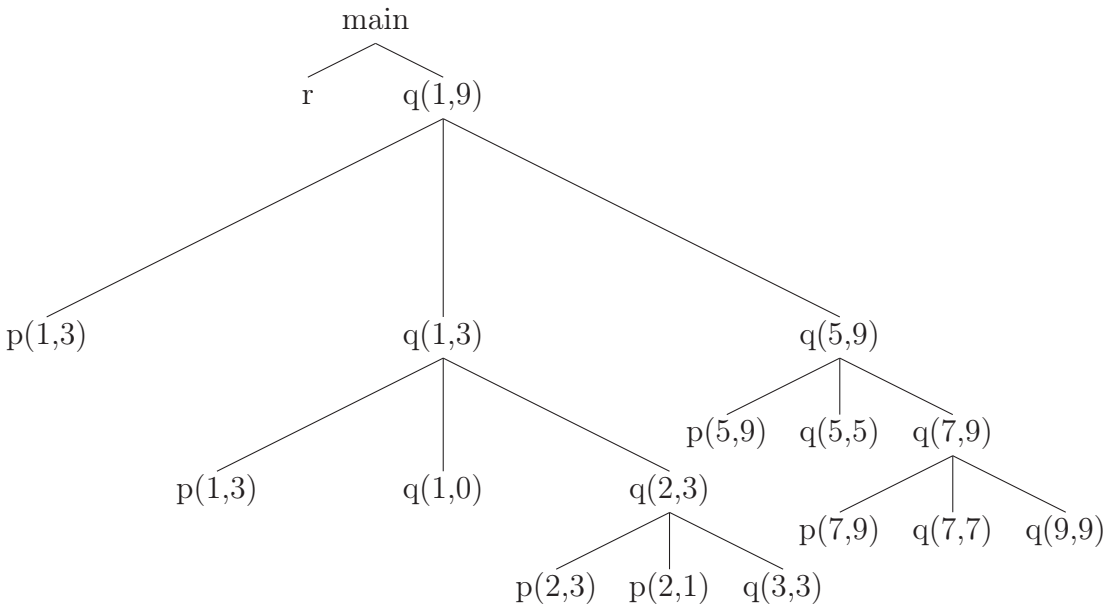
```
int a[11];
void readArray() { /* lies 9 Werte in a[1] bis a[9] */
  int i;
  ...
}
int partition(int m, int m) { /* zu einem gewaehlten Pivot-
   Element v, schreibe a so um, dass a[n..i-1] < v und a[i] = v
   und a[i+1..n] >= 0 gilt */
  ...
}
void quicksort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m,n);
    quicksort(m, i - 1);
    quicksort(i+1, n);
  }
}

main() {
  readArray();
  quicksort(1,9);
}
```

Annahmen bezüglich konkreter Eingabe

Aufruf `partition`	i
p(1,9)	4
p(1,3)	1
p(2,3)	2
p(5,9)	6
p(7,9)	8

Der zugehörige Aktivierungsbaum sieht dann wie folgt aus:



Folgende Erkenntnisse können wir daraus gewinnen:

1. Prozeduraufrufe entsprechen einem pre-Order Durchlauf
2. Prozedurterminierungen entsprechen einem Post-Order-Durchlauf
3. Wenn sich die Kontrolle in einem Methodenaufruf, der `N` entspricht, befindet, dann sind alle Aufrufe von `N` aufwärts zur Wurzel lebend, d.h. diese Aktivierungselemente müssen gehalten werden.

Wir betrachten nuun das Aktivierungssegment.

```
 ---------------------------
| aktuelle Parameter        |
|---------------------------|
| Rückgabewerte             |
|---------------------------|
| Kontrollverweis           |
|---------------------------|
| Zugriffsverweis           |
|---------------------------|
| Gerettete Maschinenzustand|
|---------------------------|
| Lokale u. temporäre Daten |
 ---------------------------
```

Vorlesung 26, 28. Januar 2010

9 Codeerzeugung

IR (Intermediate Representations), z.B. '"Drei-Adress-Code'in eine gewisse Darstellung oder lineare Darstellungen (Postfix-Code), graphische Darstellungen (Syntaxbaum, DAG) oder Befehle einer abstrakten Maschine (JVM).

Zielsprache ist geprägt durch die Zielarchitektur, wie z.B. RISC, CISC, Stack-Architektur

9.1 Wesentliche Aufgaben der Codeerzeugung

Befehlsauswahl, Registervergabe, Befehlsanordnung Es gibt mehrere Kriterien, die die Codeerzeugung erfüllen muss.

1. Korrektheit
2. Effizienz

Wir schauen uns die Befehlsauswahl an. Eine erste naive Herangehensweise wäre hier, Code-Skelette (Code-Templates) zu verwenden. Ein Beispiel wäre hier, Drei-Address-Befehle der Form `x = y +z` wie folgt zu übersetzen:

```
LD R0, y // Lade y ins Register 0
ADD R0, R0, z // Addiere  R0 und z und speichere das Ergebnis nach R0
ST x, R0 // Speichere R0 nach x
```

Dies ist natürlich recht ineffizient, da man viele redundante Speicherzugriffe hat. Eine Verbesserung wäre das Entfernen von überflüssigen Speicherzugriffen oder die Verwendung spezifischer Maschinenbefehle, z.B. `INC a`. Des Weiteren sollte man den Kontext berücksichtigen.

Beispiel Wir schauen uns die Multiplikation und Division auf Integer-Werten an. Dies wird in der Maschinensprache häufig in Registerpaaren erledigt. `MULT x,y`. Dabei liegt x in der ungeraden Komponente eines gerade/ungerade-Registerpaars und das Ergebnis liegt in der geraden Komponente. Bei `DIV x y` liegt der Dividend im geraden Teil vom gerade/ungerade-Registerpaar. Der Rest des Ergebnis liegt in der geraden, der Quotient in der ungeraden Komponente.
Wir haben folgenden Drei-Adress-Code

```
t = a + b
t = t * c
t = t / d
```

Das Übersetzen wir in folgenden Maschinencode:

```
LD R1, a
ADD R1, b
MULT R0, c
DIV R0, d
ST R1, t
```

```
t = a + b
t = t + c
t = t / d
```

```
LD R0, a
ADD R0, b
ADD R0, c
SRDA R0, 32 // Shift reduce double arithmetic
DIV R0, d
ST R1, t
```

9.2 Einfache Zielsprache

Wir können verschiedene Adressierungsarten haben

- abstrakte Adressen
- Variablen (symbolische Adressen)
- indizierte Adressen der Form `a(r)`, wobei `a` eine Adresse und `r` ein Register ist
- indirekte Adressierung der Form `*r`, wobei `r` ein Register oder eine abstrakte Adresse ist. Ein Beispiel wäre `LD R0, *p`
- Konstante der Form `#k`, für positive Integerwerte k

Jetzt haben wir verschiedene Befehlsarten

- Ladebefehl der Form `LD r addr`
- Speicherbefehl der Form `ST x, r`
- Arithmetische/ logische Befehle der Form `OP dst src1 src2` bzw. `OP dst src1`, wobei `dst`, `src1` und `src2` Register sind
- Unbedingte Sprünge `BR L`, wobei `L` eine Marke ist
- Bedingte Sprünge `BR cond r L`, wobei `cond` eine Bedingung ist (Test auf Integerwerte), die erfüllt sein muss, damit der Sprung ausgeführt wird. Typisches Beispiel wäre `BR LTZ r L`

Beispiel `a` sein ein Array, dessen Elemente 8-Bit-Worte sind. Wir übersetzten nun `b = a[i]` in Maschinencode.

```
LD R1, i
MUL R1, R1, #8
LD R2, a(R1)
ST b, R2
```

9.3 Kosten von Befehlen und Programmen

Als Näherung nehmen wir an, dass die Kosten eines Befehls 1 sind plus die Anzahl der Kosten für die Adressart. Dabei kosten Register nichts, aller anderen Adressarten jeweils 1. Die Kosten korrelieren mit der Befehlslänge.

Befehl	Kosten
`LD R0, R1`	1
`LD R0, x`	2
`LD R0, *100(R2)`	2

Vorlesung 27, 3. Februar 2010

10 Grundblöcke und Flussgraphen

Das Augenmerk liegt hier bei uns bei Schleifen. Die Idee ist, dass man gegebene IR-Programme P_I vollständig in Grundblöcke zerlegt, und zwar so, dass jeder Grundblock nur an seinem ersten Befehl betreten wird und von keinem Befehl innerhalb eines Grundblock (außer dem letzten) kann der Grundblock verlassen werden. Konstruiere einen Flussgraphen F_p zu P_I, der die Grundblöcke von P_I als Knoten (plus zwei zusätzliche Knoten *entry* und *exit*) enthält und dessen Kanten den möglichen Kontrollfluss darstellen.

10.1 Zerlegung eines IR-Programms in Grundblöcke

Bei dem Algorithmus ist die Eingabe die Befehlsfolge $c_1, \ldots, c_r$. Die Ausgabe sei die vollständige Zerlegung von P_I in $B_1, \ldots, B_n$, so dass $P_I = B_1 \,||\, \ldots \,||\, B_n$.

1. Bestimme die jeweils ersten Befehle $b_1, \ldots, b_n$ aus den Grundblöcken wie folgt:
   - Der erste Befehl von P_I ist b_1
   - Jeder Befehl, der Ziel einer Sprunganweisung ist, wird zu einem Blockanfang.
   - Jeder Befehl, der einer Sprunganweisung folgt, wird zu einem Blockanfang.
2. Jeder Grundblock B_i besteht aus b_i und allen Befehlen zwischen b_i und b_{i+1}.

Definition 20. *Flussgraph F_p zu gegebenen Programm P_I mit Zerlegung $B_1, \ldots, B_n$. Die Knotenmenge sei $\{entry, end, B_1, \ldots, B_n\}$. Konstruiere die Kantenmenge wie folgt:*

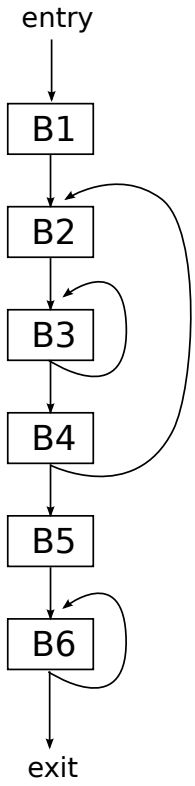
- *Füge eine Kante von entry nach B_1 ein.*
- *Zu jedem Block $B_i \in \{B_1, \ldots, B_n\}$ der ein Befehl enthält, da möglicherweise als letze in einem Programmausführung ausgeführt wird, füge eine Kante von B nach exit hinzu.*
  1. *Falls der letzte Befehl von B_n kein unbedingter Sprung ist, dann füge eine Kante von B_n nach exit ein*
  2. *Wenn ein Block B ein Sprungbefehl zu einer Marke außerhalb von P_I hat und einen Haltebefehl enthält, dann füge eine Kante vonB nach exit ein*

- *Füge eine Kante von A nach B ein, wenn*
  - *A ein Sprung zu einer Marke in B enthält oder*
  - *A nicht mit einem unbedingten Sprung endet und B auf A in der Reihenfolge $B_1, \ldots, B_n$ folgt.*

Beispiel Transformiere eine 10×10 Matrix zu einer Einheitsmatrix.

```
for i = 1 to 10 do
  for j = 1 to 10 do
    a[i,j] = 0.0;
for i = 1 to 10 do
  a[i,i] = 1.0;
```

```
1   i = 1                       B1
-----
2   j = 1                       B2
-----
3   t1 = 10 * i
4   t2 = t1 + j
5   t3 = 8 * t2
6   t4 = t3 - 88
7   a[t4] = 0.0                 B3
8   j = j+1
9   if j <= 10 goto (3)
-----
10  i = i+1                     B4
11  if i <= 10 goto (2)
-----
12  i = 1                       B5
-----
13  t5 = i - 1
14  t6 = 88 * t5
15  a[t6] = 1.0                 B6
16  i = i+1
17  if i <= 10 goto (13)
```



Definition 21. *Eine Knotenmenge L aus F heißt Schleife, genau dann wenn*

1.

Vorlesung 28, 4. Februar 2010

10.2 Lebendigkeit und nächste Verwendung

Es folgen einige Infos zur „Lebendigkeit“ und „nächsten Verwendung“ von Variablen direkt an IR-Befehlen. Dadurch werden Optimierungen ermöglicht.

Definition Sei $i : x = \ldots$ ein IR-Befehl. Wenn in einer anderen Anweisung $j : \ldots = \ldots x \ldots$ die Variable x als Operand verwendet wird und der Kontrollfluss von i nach j verlaufen kann, ohne weitere Wertzuweisungen an x, dann ist die „nächste Verwendung“ von x bzgl. Anweisung i die Anweisung j. x heißt *lebendig* in i, wenn es ein j gibt, sodass die „nächste Verwendung“ von x j ist.

Verfahren zur Erstellung dieser Infos: Rückwärtsdurchlauf unter Verwendung der Symboltabelle. Hier: Analyse innerhalb von Grundblöcken B.

- Trage initial in die Symboltabelle zu jeder Variable die Info „lebendig“ und „keine nächste Verwendung“ ein.
- Beginne mit der letzten Auswertung von B und behandle alle Anweisungen aus B rückwärts. Zu jeder Anweisung der Form `i: x = y+z` erledige folgendes:
  1. Trage hinter Anweisung i alle Infos zu x, y und z aus der Symboltabelle ein.
  2. Trage in die Symboltabelle zu x „nicht lebendig“ und „keine nächste Verwendung“ ein.
  3. Trage zu y und z in die Symboltabelle „lebendig“ und „nächste Verwendung in i“ ein.
  4. Verfahre zu Befehlen mit anderen Operatoren analog.

Beispiel

```
1  t1 = 10 * i
2  t2 = t1 * j
3  y = j + 1
```

Die Symboltabelle

	lebendig	nächste Verwendung
i	ja	(1)
j	ja	(2)
$t1$	nein	keine
$t2$	nein	keine

10.3 Lokale Optimierungen

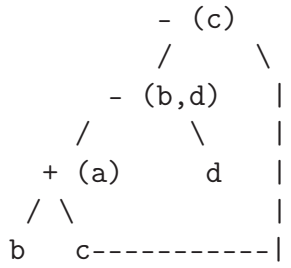
Darstellung der IR-Blöcke als DAG.

1. Zu jeder Variable x und Konstante k gib es ein Blatt x_0 bzw. k (Eingangswerte)
2. Zu jeder Anweisung c in B erzeuge einen Knoten N in G mit Kindern, die genau den letzten Zuweisungen an die Operanden von c gehören. Markiere N mit dem Operanden von c.
3. Füge die Markierung alle Variablen hinzu, deren letzte Wertzuweisung in c erfolgt
4. Die Knoten, deren Variablen am Blockausgang lebendig sind, heißen Ausgangsknoten. Die Info kommt von der globalen Analyse.

Beispiel

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Dazu kann folgender DAG konstruiert werden:



Daraus kann man jetzt wieder optimierten Code extrahieren.

```
a = b + c
b = a - d
d = b
c = b - c
```

Falls d am Ausgang nicht lebendig ist, kann man die Zuweisung $d = b$ auch streichen. Es bestehen folgende Optimierungsmöglichkeiten

1. Eliminierung gemeinsamer Teilausdrücke (Neuzuweisung beachten)
2. Eliminierung von passivem Code, d.h. Wertzuweisungen an Variablen, die nicht mehr verwendet werden.
3. Änderung der Befehlsreihenfolge
4. Ausnutzung algebraischer Gesetze

Bei Verweisung von Arrays und Dereferenzierungen muss man aufpassen.