Script generated by TTT

Title: Simon: Compilerbau (06.06.2012)

Date: Wed Jun 06 14:01:49 CEST 2012

Duration: 77:31 min

Pages: 59

Kapitel 2:
Symboltabellen

Symboltabellen

Betrachte folgenden Java Kode:

```
void foo() {
  int A;
  void bar() {
    double A;
    A = 0.5;
    write(A);
  }
  A = 2;
  bar();
  write(A);
}
```

- innerhalb des Rumpfs von `bar` wird die Definition von `A` durch die *lokale Definition* verdeckt
- für die Code-Erzeugung benötigen wir für jede Benutzung eines Bezeichners die zugehörige *Definitionsstelle*
- *statische Bindung* bedeutet, dass die Definition eines Namens `A` an allen Programmpunkten innerhalb des gesamten Blocks *gültig* ist.
- *sichtbar* ist sie aber nur in denjenigen Teilbereichen, in denen keine weitere Definition von `A` gültig ist

Gültigkeitsbereiche von Bezeichnern

```
void foo() {
  int A;
  void bar() {
    double A;
    A = 0.5;
    write(A);
  }
  A = 2;
  bar();
  write(A);
}
```

Gültigkeitsbereich von `int A`

Gültigkeitsbereiche von Bezeichnern

```
void foo() {
  int A;
  void bar() {
    double A;
    A = 0.5;
    write(A);
  }
  A = 2;
  bar();
  write(A);
}
```

Gültigkeitsbereich von **double** A

Gültigkeitsbereiche von Bezeichnern

```
void foo() {
  int A;
  void bar() {
    double A;
    A = 0.5;
    write(A);
  }
  A = 2;
  bar();
  write(A);
}
```

Gültigkeitsbereich von **double** A

Verwaltungs von Bezeichnern kann kompliziert werden...

Sichtbarkeitsregeln in objektorientierten Prog.spr.

```
1  public class Foo {
2    protected int x = 17;
3    protected int y =  5;
4    private int z = 42;
5    public int b() { return 1; }
6  }
7  class Bar extends Foo {
8    protected double y = 0.5;
9    public int b(int a) { return a; }
10 }
```

Beobachtung:

Sichtbarkeitsregeln in objektorientierten Prog.spr.

```
1  public class Foo {
2    protected int x = 17;
3    protected int y =  5;
4    private int z = 42;
5    public int b() { return 1; }
6  }
7  class Bar extends Foo {
8    protected double y = 0.5;
9    public int b(int a) { return a; }
10 }
```

Beobachtung:

- `private` Members sind nur innerhalb der aktuellen Klasse sichtbar
- `protected` Members sind innerhalb der Klasse, in den Unterklassen sowie innerhalb des gesamten package sichtbar
- Methoden `b` gleichen Namens sind stets verschieden, wenn ihre Argument-Typen verschieden sind ⇝ overloading

Sichtbarkeitsregeln in objektorientierten Prog.spr.

```
1  public class Foo {
2    protected int x = 17;
3    protected int y =  5;
4    private int z = 42;
5    public int b() { return 1; }
6  }
7  class Bar extends Foo {
8    protected double y = 0.5;
9    public int b(int a) { return a; }
10 }
```

b, ∅
b, int
double

Beobachtung:

- `private` Members sind nur innerhalb der aktuellen Klasse sichtbar
- `protected` Members sind innerhalb der Klasse, in den Unterklassen sowie innerhalb des gesamten package sichtbar
- Methoden `b` gleichen Namens sind stets verschieden, wenn ihre Argument-Typen verschieden sind ⇝ overloading

class Cell {...}

```
public class Queue {
    protected Cell head;
    protected Cell tail;
}

public class PriorityQueue extends Queue {
    public void merge (PriorityQueue pq) {
        ... pq.head ... pq.tail
    }
    public void merge (Queue q) {
        ... q.head ... q.tail
    }
}
```

Fehler

Dynamische Auflösung von Funktionen

```
public class Foo {
  protcted int foo() { return 1; }
}
class Bar extends Foo {
  protected int foo() { return 2; }
  public int test(boolean b) {
    Foo x = (b) ? new Foo() : new Bar();
    return x.foo();
  }
}
```

Beobachtungen:

Dynamische Auflösung von Funktionen

```
public class Foo {
  protcted int foo() { return 1; }
}
class Bar extends Foo {
  protected int foo() { return 2; }
  public int test(boolean b) {
    Foo x = (b) ? new Foo() : new Bar();
    return x.foo();
  }
}
```

Beobachtungen:

- der Typ von `x` ist `Foo` oder `Bar`, je nach Wert von `b`
- `x.foo()` ruft entweder `foo` in Zeile 2 oder Zeile 5 auf

34 / 184

Dynamische Auflösung von Funktionen

```
public class Foo {
  protcted int foo() { return 1; }
}
class Bar extends Foo {
  protected int foo() { return 2; }
  public int test(boolean b) {
    Foo x = (b) ? new Foo() : new Bar();
    return x.foo();
  }
}
```

Beobachtungen:

- der Typ von `x` ist `Foo` oder `Bar`, je nach Wert von `b`
- `x.foo()` ruft entweder `foo` in Zeile 2 oder Zeile 5 auf
- Entscheidung wird zur *Laufzeit* gefällt (hat nichts mit Namensauflösung zu tun)

Überprüfung von Bezeichnern

Aufgabe: Finde zu jeder Benutzung eines Bezeichners die zugehörige *Deklaration*

Idee:

1. ersetze Bezeichner durch *eindeutige* Nummern
2. finde zu jedem Benutzung eines Bezeichners die *Deklaration*

Überprüfung von Bezeichnern

Aufgabe: Finde zu jeder Benutzung eines Bezeichners die zugehörige *Deklaration*

Idee:

1. ersetze Bezeichner durch *eindeutige* Nummern
   - das Vergleichen von Nummern ist schneller
   - das Ersetzen von gleichen Namen durch eine Nummer spart Speicher
2. finde zu jedem Benutzung eines Bezeichners die *Deklaration*

Überprüfung von Bezeichnern

Aufgabe: Finde zu jeder Benutzung eines Bezeichners die zugehörige *Deklaration*

Idee:

1. ersetze Bezeichner durch *eindeutige* Nummern
   - das Vergleichen von Nummern ist schneller
   - das Ersetzen von gleichen Namen durch eine Nummer spart Speicher
2. finde zu jedem Benutzung eines Bezeichners die *Deklaration*
   - für jede Deklaration eines Bezeichners muss zur Laufzeit des Programs Speicher reserviert werden
   - bei Programmiersprachen ohne explizite Deklaration wird implizit eine Deklaration bei der ersten Benutzung eines Bezeichners erzeugt

(1) Ersetze Bezeichner durch eindeutige Namen

Idee für Algorithmus:

Eingabe: Folge von Strings
Ausgabe:
1. Folge von Nummern
2. Tabelle, die zu Nummern die Strings auflistet

Wende diesen Algorithmus im Scanner auf jeden Bezeichner an.

Beispiel für die Anwendung des Algorithmus

Eingabe:

das	schwein	ist	dem	schwein	was	

	das	schwein	dem	menschen	ist	wurst

Ausgabe:

0	1	2	3	1	4	0	1	3	5	2	6

und

0	das
1	schwein
2	ist
3	dem
4	was
5	menschen
6	wurst

Spezifikation der Implementierung

Idee:

- benutze eine *partielle Abbildung*: S : **String** $\rightharpoonup$ **int**
- verwalte einen Zähler **int** count $= 0$; für die Anzahl der bereits gefundenen Wörter

Damit definieren wir eine Funktion **int** getIndex(**String** w):

```
int getIndex(String w) {
    if (S(w) ≡ undefined) {
        S = S ⊕ {w ↦ count};
        return count++;
    else return S(w);
}
```


Datenstrukturen für partielle Abbildungen

Ideen:

- Liste von Paaren $(w, i) \in$ **String** $\times$ **int** :
  $\mathcal{O}(1)$
  $\mathcal{O}(n)$ $\rightsquigarrow$ zu teuer

Datenstrukturen für partielle Abbildungen

Ideen:

- Liste von Paaren $(w, i) \in \textbf{String} \times \textbf{int}$:
  $\mathcal{O}(1)$
  $\mathcal{O}(n)$ $\rightsquigarrow$ zu teuer
- balancierte Bäume :
  $\mathcal{O}(\log(n))$
  $\mathcal{O}(\log(n))$ $\rightsquigarrow$ zu teuer

Datenstrukturen für partielle Abbildungen

Ideen:

- Liste von Paaren $(w, i) \in \textbf{String} \times \textbf{int}$:
  $\mathcal{O}(1)$
  $\mathcal{O}(n)$ $\rightsquigarrow$ zu teuer
- balancierte Bäume :
  $\mathcal{O}(\log(n))$
  $\mathcal{O}(\log(n))$ $\rightsquigarrow$ zu teuer
- Hash Tables :
  $\mathcal{O}(1)$
  $\mathcal{O}(1)$ zumindest im Mittel

Datenstrukturen für partielle Abbildungen

Ideen:

- Liste von Paaren $(w, i) \in \textbf{String} \times \textbf{int}$:
  $\mathcal{O}(1)$
  $\mathcal{O}(n)$ $\rightsquigarrow$ zu teuer
- balancierte Bäume :
  $\mathcal{O}(\log(n))$
  $\mathcal{O}(\log(n))$ $\rightsquigarrow$ zu teuer
- Hash Tables :
  $\mathcal{O}(1)$
  $\mathcal{O}(1)$ zumindest im Mittel

Caveat: In alten Compilern war der Scanner das langsamste Glied in der Kette. Heute der Zeitverbrauch des Scanners eher gering.

Implementierung mit Hash-Tabellen

- lege ein Feld M von hinreichender Größe m an
- wähle eine *Hash-Funktion* H : $\textbf{String} \to [0, m-1]$ mit den Eigenschaften:
  - $H(w)$ ist leicht zu berechnen
  - H streut die vorkommenden Wörter gleichmäßig über $[0, m-1]$

Mögliche Wahlen ($\vec{x} = \langle x_0, \ldots x_{r-1} \rangle$):

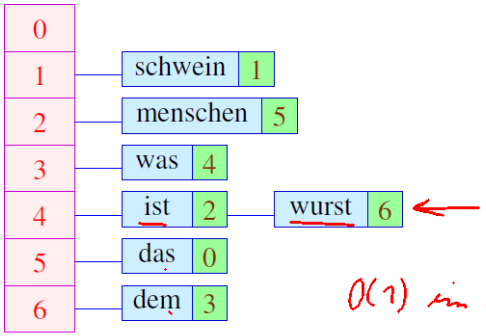
$$
\begin{aligned}
H_0(\vec{x}) &= (x_0 + x_{r-1}) \,\%\, m \\
H_1(\vec{x}) &= \left(\textstyle\sum_{i=0}^{r-1} x_i \cdot p^i\right) \% \, m \\
H_2(\vec{x}) &= (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m
\end{aligned}
$$

für eine Primzahl p (z.B. 31)

- Das Argument-Wert-Paar (w, i) legen wir dann in $M[H(w)]$ ab

Berechnung einer Hash-Tabelle am Beispiel

Mit $m = 7$ und H_0 erhalten wir:



Um den Wert des Worts w zu finden, müssen wir w mit allen Worten x vergleichen, für die $H(w) = H(x)$

Überprüfung von Bezeichnern: (2) Symboltabellen

Überprüfe die korrekte Benutzung von Bezeichnern:

- Durchmustere den Syntaxbaum in einer geeigneten Reihenfolge, die
  - jede Definition vor ihren Benutzungen besucht
  - die jeweils aktuell sichtbare Definition zuletzt besucht
- für jeden Bezeichner verwaltet man einen *Keller* der gültigen Definitionen
- trifft man bei der Durchmusterung auf eine *Deklaration* eines Bezeichners, schiebt man sie auf den Keller
- verlässt man den *Gültigkeitsbereich*, muss man sie wieder vom Keller werfen
- trifft man bei der Durchmusterung auf eine Benutzung, schlägt man die letzte Deklaration auf dem Keller nach
- findet man keine Deklaration, haben wir einen Fehler gefunden

Beispiel: Keller von Symboltabellen

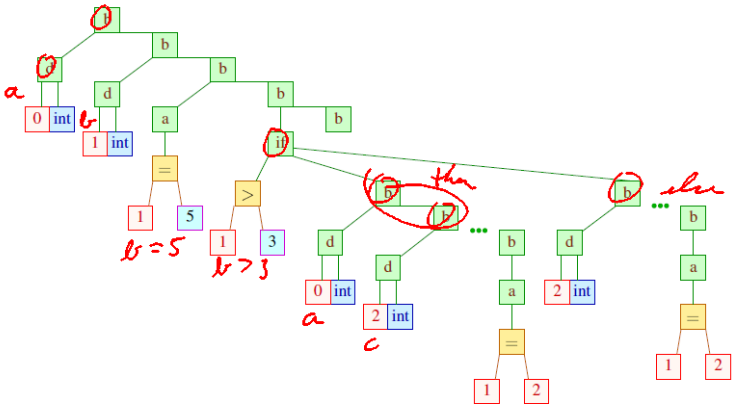
```
{
  int a, b;
  b = 5;
  if (b>3) {
    int a, c;
    a = 3;
    c = a + 1;
    b = c;
  } else {
    int c;
    c = a + 1;
    b = c;
  }
  b = a + b;
}
```

Abbildung von Namen auf Zahlen:

0	a
1	b
2	c

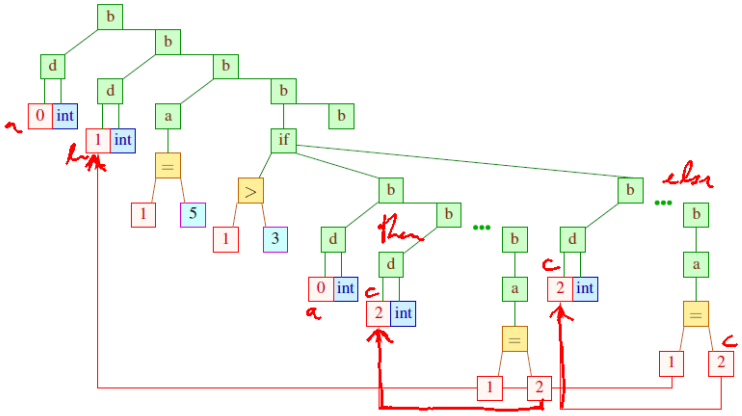
Der zugehörige Syntaxbaum

- **d** Deklaration
- **b** Basis-Block
- **a** Zuweisung



Der zugehörige Syntaxbaum

d Deklaration
b Basis-Block
a Zuweisung



Praktische Implementierung: Sichtbarkeit

- Das Auflösen von Bezeichnern kann durch L-attributierte Grammatik erfolgen

Praktische Implementierung: Sichtbarkeit

- Das Auflösen von Bezeichnern kann durch L-attributierte Grammatik erfolgen
- Benutzt man eine Listen-Implementierung der Symboltabelle und verwaltet man Marker für Blöcke, kann man auf das Entfernen von Deklarationen am Ende eines Blockes verzichten

vor if-Anweisung

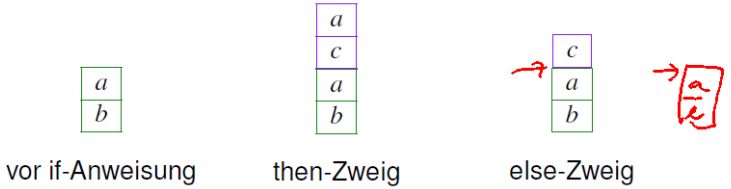
Praktische Implementierung: Sichtbarkeit

- Das Auflösen von Bezeichnern kann durch L-attributierte Grammatik erfolgen
- Benutzt man eine Listen-Implementierung der Symboltabelle und verwaltet man Marker für Blöcke, kann man auf das Entfernen von Deklarationen am Ende eines Blockes verzichten

vor if-Anweisung then-Zweig

Praktische Implementierung: Sichtbarkeit

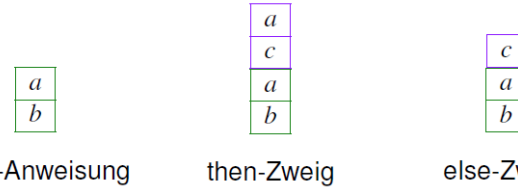
- Das Auflösen von Bezeichnern kann durch L-attributierte Grammatik erfolgen
- Benutzt man eine Listen-Implementierung der Symboltabelle und verwaltet man Marker für Blöcke, kann man auf das Entfernen von Deklarationen am Ende eines Blockes verzichten



vor if-Anweisung | then-Zweig | else-Zweig

Praktische Implementierung: Sichtbarkeit

- Das Auflösen von Bezeichnern kann durch L-attributierte Grammatik erfolgen
- Benutzt man eine Listen-Implementierung der Symboltabelle und verwaltet man Marker für Blöcke, kann man auf das Entfernen von Deklarationen am Ende eines Blockes verzichten



vor if-Anweisung | then-Zweig | else-Zweig

- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben
  - Anstatt ein eindeutige Nummern zu vergeben wird in der Praxis auch gerne ein Zeiger zum Deklarationsknoten gespeichert

Mehrfach- und Vorwärtsdeklarationen

Manche Programmiersprachen verbieten eine *Mehrfachdeklaration* des selben Namens innerhalb eines Funktionsrumpfes.

Mehrfach- und Vorwärtsdeklarationen

Manche Programmiersprachen verbieten eine *Mehrfachdeklaration* des selben Namens innerhalb eines Funktionsrumpfes.

- weise jedem Block eine eindeutige Nummer zu
- speichere für jede Deklaration auch die Nummer des Blocks, zu der sie gehört

Mehrfach- und Vorwärtsdeklarationen

Manche Programmiersprachen verbieten eine *Mehrfachdeklaration* des selben Namens innerhalb eines Funktionsrumpfes.

- weise jedem Block eine eindeutige Nummer zu
- speichere für jede Deklaration auch die Nummer des Blocks, zu der sie gehört
- gibt es eine weitere Deklaration des gleichen Namens mit derselben Blocknummer, muss ein Fehler gemeldet werden

Mehrfach- und Vorwärtsdeklarationen

Manche Programmiersprachen verbieten eine *Mehrfachdeklaration* des selben Namens innerhalb eines Funktionsrumpfes.

- weise jedem Block eine eindeutige Nummer zu
- speichere für jede Deklaration auch die Nummer des Blocks, zu der sie gehört
- gibt es eine weitere Deklaration des gleichen Namens mit derselben Blocknummer, muss ein Fehler gemeldet werden

Wie zuvor können eindeutige Block Nummern auch durch Zeiger auf die Blöcke implementiert werden.

Um rekursive Datenstrukturen zu ermöglichen erlauben Sprachen *Vorwärtsdeklarationen:*

```
struct list1;
struct list0 {
  int info;
  struct list1* next;
}
```

```
struct list1 {
  int info;
  struct list0* next;
}
```


Deklaration von Funktionsnamen

Auf gleiche Weise müssen auch Funktionsnamen verwaltet werden.

- bei einer *rekursiven Funktion* muss der Name vor der Durchmusterung des Rumpfes in die Tabelle eingetragen werden

```
int fac(int i) {
  return i*fac(i-1);
}
```


Deklaration von Funktionsnamen

Auf gleiche Weise müssen auch Funktionsnamen verwaltet werden.

- bei einer *rekursiven Funktion* muss der Name vor der Durchmusterung des Rumpfes in die Tabelle eingetragen werden

```
int fac(int i) {
  return i*fac(i-1);
}
```

- bei *wechselseitig rekursiven Funktionsdefinitionen* gilt dies für alle Funktionsnamen. Beispiel in ML und C:

```
fun odd 0 = false
  | odd 1 = true
  | odd x = even (x-1)
and even 0 = true
  | even 1 = false
  | even x = odd (x-1)
```

```
int even(int x);
int odd(int x) {
  return (x==0 ? 0 :
    (x==1 ? 1 : even(x-1)));
}
int even(int x) {
  return (x==0 ? 1 :
    (x==1 ? 0 : odd(x-1)));
}
```

Überladung von Namen

Ähnliche Abhängigkeiten gelten für objekt-orientierte Sprachen:

- bei objekt-orientierter Sprache mit Vererbung zwischen Klassen sollte die übergeordnete Klasse vor der Unterklasse besucht werden

Überladung von Namen

Ähnliche Abhängigkeiten gelten für objekt-orientierte Sprachen:

- bei objekt-orientierter Sprache mit Vererbung zwischen Klassen sollte die übergeordnete Klasse vor der Unterklasse besucht werden
- bei Überladung muss simultan die Signatur verglichen werden
  - eventuell muss eine Typüberprüfung vorgenommen werden

Sobald Namen von Bezeichnern aufgelöst sind, kann die semantische Analyse fortsetzen mit der *Typprüfung* (oder *Typinferenz*, siehe Vorlesung „Programmiersprachen").

48 / 184

Mehrere Sorten von Bezeichnern

Einige Programmiersprachen unterscheiden verschiedene Bezeichner:

- C: Variablennamen und Typnamen
- Java: Klassen, Methoden, Felder
- Haskell: Typnamen, Variablen, Operatoren mit Prioritäten

Mehrere Sorten von Bezeichnern

Einige Programmiersprachen unterscheiden verschiedene Bezeichner:

- C: Variablennamen und Typnamen
- Java: Klassen, Methoden, Felder
- Haskell: Typnamen, Variablen, Operatoren mit Prioritäten

In einigen Fällen *verändern* Deklarationen in der Sprache die Bedeutung eines Bezeichners:

- Der Parser informiert den Scanner über eine neue Deklaration
- Der Scanner generiert verschiedene Token für einen Bezeichner
- Der Parser generiert einen Syntaxbaum, der von den bisherigen Deklarationen abhängt

49 / 184

Mehrere Sorten von Bezeichnern

Einige Programmiersprachen unterscheiden verschiedene Bezeichner:

- C: Variablennamen und Typnamen
- Java: Klassen, Methoden, Felder
- Haskell: Typnamen, Variablen, Operatoren mit Prioritäten

In einigen Fällen *verändern* Deklarationen in der Sprache die Bedeutung eines Bezeichners:

- Der Parser informiert den Scanner über eine neue Deklaration
- Der Scanner generiert verschiedene Token für einen Bezeichner
- Der Parser generiert einen Syntaxbaum, der von den bisherigen Deklarationen abhängt

Interaktion zwischen Parser und Scanner: problematisch!

Fixity-Deklaration in Haskell

Haskell erlaubt *beliebige* binäre Operatoren aus $(?!\hat{}\&|=+-\_*/)^+$.
In Haskell Standard-Bibliothek:

```
infixr 8 ^
infixl 7 *,/
infixl 6 +,-
infix  4 ==,/=
```

Grammatik ist *generisch*:

$$
\begin{array}{lcl}
Exp_0 & ::= & Exp_0\ LOp_0\ Exp_1 \\
 & | & Exp_1\ ROp_0\ Exp_0 \\
 & | & Exp_1\ Op_0\ Exp_1 \\
 & | & Exp_1 \\
 & & \vdots \\
Exp_9 & ::= & Exp_9\ LOp_9\ Exp \\
 & | & Exp\ ROp_9\ Exp_9 \\
 & | & Exp\ Op_9\ Exp \\
 & | & Exp \\
Exp & ::= & \textbf{ident} \mid \textbf{num} \\
 & | & \textbf{(}\ Exp_0\ \textbf{)}
\end{array}
$$

Fixity-Deklaration in Haskell

Haskell erlaubt *beliebige* binäre Operatoren aus $(?!\hat{}\&|=+-\_*/)^+$.
In Haskell Standard-Bibliothek:

```
infixr 8 ^
infixl 7 *,/
infixl 6 +,-
infix  4 ==,/=
```

Grammatik ist *generisch*:

$$
\begin{array}{lcl}
Exp_0 & ::= & Exp_0\ LOp_0\ Exp_1 \\
 & | & Exp_1\ ROp_0\ Exp_0 \\
 & | & Exp_1\ Op_0\ Exp_1 \\
 & | & Exp_1 \\
 & & \vdots \\
Exp_9 & ::= & Exp_9\ LOp_9\ Exp \\
 & | & Exp\ ROp_9\ Exp_9 \\
 & | & Exp\ Op_9\ Exp \\
 & | & Exp \\
Exp & ::= & \textbf{ident} \mid \textbf{num} \\
 & | & \textbf{(}\ Exp_0\ \textbf{)}
\end{array}
$$

- Parser trägt Deklarationen in Tabelle ein
- Scanner erzeugt:
  - Operator `-` wird zum Token LOp_6.
  - Operator `*` wird zum Token LOp_7.
  - Operator `==` wird zum Token Op_4.
  - etc.
- Parser erkennt `3-(4*5)-6` als `(3-(4*5))-6`

Fixity-Deklarationen in Haskell: Beobachtungen

Nicht ohne Probleme:

- Scanner hat einen *Zustand*, der vom Parser bestimmt wird
  $\rightsquigarrow$ nicht mehr *kontextfrei*, braucht globale Datenstruktur

Fixity-Deklarationen in Haskell: Beobachtungen

Nicht ohne Probleme:

- Scanner hat einen *Zustand*, der vom Parser bestimmt wird
  $\rightsquigarrow$ nicht mehr *kontextfrei*, braucht globale Datenstruktur
- ein Stück Kode kann mehrere Semantiken haben
- syntaktische Korrektheit hängt evtl. von importierten Modulen ab

Fixity-Deklarationen in Haskell: Beobachtungen

Nicht ohne Probleme:

- Scanner hat einen *Zustand*, der vom Parser bestimmt wird
  $\rightsquigarrow$ nicht mehr *kontextfrei*, braucht globale Datenstruktur
- ein Stück Kode kann mehrere Semantiken haben
- syntaktische Korrektheit hängt evtl. von importierten Modulen ab
- Fehlermeldungen des Parsers schwer verständlich

Im GHC Haskell Compiler werden alle Operatoren als LOp_0 gelesen und der AST anschliessend transformiert.

Typbezeichner und Variablen in C

Die C Grammatik unterscheidet zwischen Terminal `typedef-name` und `identifier`.
Betrachte folgende Liste von Deklarationen:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevante C Grammatik:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` $\cdots$ `typedef`
	|	`void` | `char` | `char` $\cdots$ `typedef-name`
declarator	$\rightarrow$	`identifier` | $\cdots$

Typbezeichner und Variablen in C

Die C Grammatik unterscheidet zwischen Terminal `typedef-name` und `identifier`.
Betrachte folgende Liste von Deklarationen:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevante C Grammatik:

declaration	$\rightarrow$	(declaration-specifier)$^+$ declarator ;
declaration-specifier	$\rightarrow$	`static` | `volatile` $\cdots$ `typedef`
	|	`void` | `char` | `char` $\cdots$ `typedef-name`
declarator	$\rightarrow$	`identifier` | $\cdots$

Problem:

- Parser trägt `point_t` in Typen-Tabelle ein wenn die declaration Regel reduziert wird

Typbezeichner und Variablen in C

Die C Grammatik unterscheidet zwischen Terminal `typedef-name` und `identifier`.
Betrachte folgende Liste von Deklarationen:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevante C Grammatik:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Problem:

- Parser trägt `point_t` in Typen-Tabelle ein wenn die declaration Regel reduziert wird
- Parserzustand hat mindestens ein Lookahead-Token

Typbezeichner und Variablen in C

Die C Grammatik unterscheidet zwischen Terminal `typedef-name` und `identifier`.
Betrachte folgende Liste von Deklarationen:

```
typedef struct { int x,y } point_t;
point_t origin;
```

Relevante C Grammatik:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Problem:

- Parser trägt `point_t` in Typen-Tabelle ein wenn die declaration Regel reduziert wird
- Parserzustand hat mindestens ein Lookahead-Token
- Scanner hat `point_t` in zweiter Zeile also schon als identifier gelesen

Typbezeichner und Variablen in C: Lösungen

Relevante C Grammatik:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Lösung schwierig:

- versuche, Lookahead-Token im Parser zu ändern
- füge folgende Regel zur Grammatik hinzu:
  typedef-name → `identifier`

Typbezeichner und Variablen in C: Lösungen

Relevante C Grammatik:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Lösung schwierig:

- versuche, Lookahead-Token im Parser zu ändern
- füge folgende Regel zur Grammatik hinzu:
  typedef-name → `identifier`
- registriere den Typnamen früher

Typbezeichner und Variablen in C: Lösungen

Relevante C Grammatik:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Lösung schwierig:

- versuche, Lookahead-Token im Parser zu ändern
- füge folgende Regel zur Grammatik hinzu:
  typedef-name → `identifier`
- registriere den Typnamen früher
  - separiere Regel für `typedef` Produktion

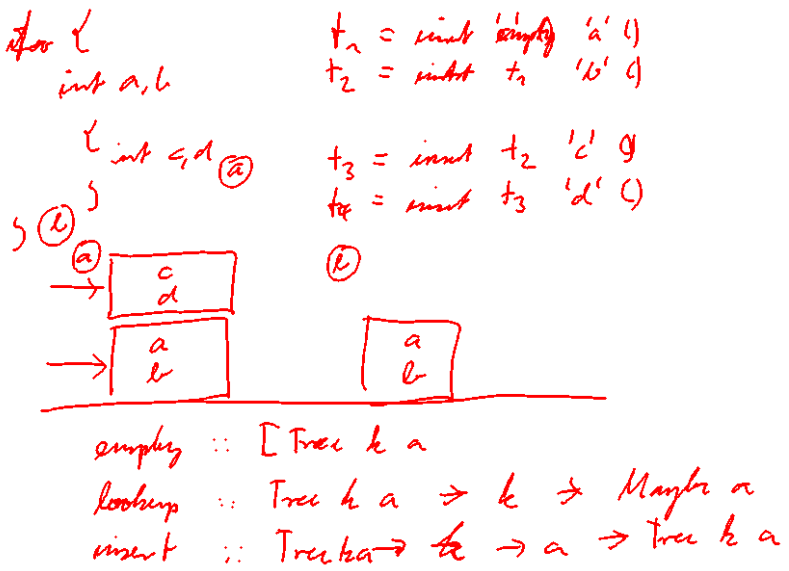
Typbezeichner und Variablen in C: Lösungen

Relevante C Grammatik:

declaration	→	(declaration-specifier)$^+$ declarator ;
declaration-specifier	→	`static` | `volatile` ⋯ `typedef`
	|	`void` | `char` | `char` ⋯ `typedef-name`
declarator	→	`identifier` | ⋯

Lösung schwierig:

- versuche, Lookahead-Token im Parser zu ändern
- füge folgende Regel zur Grammatik hinzu:
  typedef-name → `identifier`
- registriere den Typnamen früher
  - separiere Regel für `typedef` Produktion
  - rufe alternative declarator Produktion auf, die `identifier` als Typenamen registriert



Ausblick

- Implementierung von Symboltabellen für C
- nächste Woche: Überprüfung von Typen