

Max Wisniewski , Alexander Steen

Tutor: Ansgar Schneider

Aufgabe 1

Ändern Sie die Syntax von *WHILE*, indem Sie *INTEGER* und *REAL* Zahlen unterscheiden.

Lösung:

Wir müssen zunächst die Menge *ZAHL* in die Mengen *INTEGER* und *REAL* aufteilen. Haben wir dies erledigt, muss man noch alle Operationen so aufteilen, dass man nicht einfach *REAL* und *INTEGER* Zahlen mischen kann. Will man dies, geben wir noch eine explizite Konversion von *INTEGER* nach *REAL* dazu.

Syntax

Elementare Einheiten:

$$\begin{aligned} \underline{\text{INTEGER}} &::= \underline{0} \mid \underline{1} \mid \underline{-1} \mid \underline{2} \mid \underline{-2} \mid \dots \\ \underline{\text{ZIFFERNFOLGE}} &::= \underline{0\text{ZIFFERNFOLGE}} \mid \dots \mid \underline{9\text{ZIFFERNFOLGE}} \mid \varepsilon \\ \underline{\text{REAL}} &::= \underline{\text{INTEGER}.\text{ZIFFERNFOLGE}} \\ \underline{Z} &::= \underline{\text{INTEGER}} \mid \underline{\text{REAL}} \end{aligned}$$

An den restlichen Definitionen muss nichts geändert werden, damit wir syntaktisch *REAL* und *INTEGER* unterscheiden können. Nun ist allerdings auch syntaktisch korrekt, die beiden Zahlenarten miteinander zu verrechnen. Dies könnte man zwar auch syntaktisch verbieten, in dem man z.B. die Terme *T* und die Operationen *BOP* und *OP* in zwei verschiedene Ausprägungen (je eine für *REAL* und *INTEGER*) unterteilt. Dann müsste man aber noch künstlichen Einschränkungen an die Variablenbezeichner anlegen, sodass hier ebenfalls unterschieden werden kann. Da dies allerdings nicht wirklich sinnvoll erscheint, müsste man gemischte Terme auf semantischer Ebene überprüfen und dort entscheiden, welches Verhalten gewünscht ist.

Aufgabe 2

Definieren Sie eine konkrete Syntax, die eindeutig ist.

Lösung:

Die elementaren Einheiten können diesmal unverändert bleiben.

Induktive Einheiten:

$$\begin{aligned} T &::= Z \mid I \mid (T_1 \text{ OP } T_2) \mid \text{read} \\ B &::= W \mid \text{not } B \mid T_1 \text{ BOP } T_2 \mid \text{read} \\ C &::= \text{skip} \mid I := T \mid (C_1; C_2) \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi} \mid \\ &\quad \text{while } B \text{ do } C \text{ od} \mid \text{output } T \mid \text{output } B \\ P &::= C \end{aligned}$$

Wir müssen nur jede Produktion, die aus mehr als einem Nicht-Terminal Symbol besteht, entweder mit Klammern oder mit einem Abschlusschlüsselwort versehen, sollte dies nötig sein. So werden arithmetische Operationen nun auf jedenfall durch eine Klammer abgeschlossen. Das *if* wird nach dem *else* durch ein Abschließendes *fi* ersetzt. Bei

while wird das *do* durch ein *od* geschlossen. Da man den Gültigkeitsbereich nun explizit angeben muss, wird die konkrete Syntax eindeutig.

Die booleschen Operationen müssen nicht abgeschlossen werden, da wir in unserer Syntax boolesche Werte nicht weiter kombinieren können. So schließt eine boolesche Operation den Ausdruck an sich ab. Die Komposition sieht im ersten Moment komisch aus, wenn wir diese erzeugen, da wir nun viele unnütze Klammern haben, aber nur auf diese Weise können wir verhindern, dass $c_1; c_2; c_3$ auf 2 Arten interpretiert werden kann. Bei arithmetischen Ausdrücken und der Komposition ist dies in den meisten Fällen nicht die gewünscht Lösung. Dort sollte eher auf Semantikebene eine Bindungsstärke, bzw. Ausführungsreihenfolge angegeben werden, aber in der Syntax ist dies nicht anders eindeutig machbar.

Aufgabe 3

Formulieren Sie informell eine Präzisierung der angegebenen WHILE-Semantik, die die genannte Fehlerquellen:

1. Bereichsüberschreitungen
2. Division durch Null
3. Berechnung von *read* bei leerer Eingabedatei
4. Typkonflikte

behandelt.

Lösung:

Da wir der Aufgabe nach nichts weiter erfüllen müssen könnten wir uns für die Praxis unsinnige Ergebnisse ausdenken. So könnte in einem Fehlerfall z.B. immer das literal 7 zurückgegeben werden, oder wir verlangen, dass nach einem Fehler im Programm heiloses Chaos herrschen soll (wie ein Programmierer in realer Semantik einmal spezifiziert hat). Wir haben uns für ein etwas handhabbares Model entscheiden, bei dem man noch die Variable *errno* braucht und REAL die Werte *NaN*, *INFTY*, *-INFTY* annehmen kann.

Bereichsüberlauf: Wir könnten an dieser Stelle in *errno* z.B. 1 schreiben. Dies wird für alle Anweisungen, die nach dieser ausgeführt werden bedeuten, dass wir einen Overflow hatten. Im Falle von Integer, werden wir bei einem Überlauf bei *MinValue* starten und von dort aus den Rest addieren, bei einem Unterlauf, werden wir von *MaxValue* den Rest abziehen. Bei REAL Zahlen werden wir bei einem Überlauf *INFTY* und bei einem Unterlauf *-INFTY* ausgeben.

Division by zero: Wir werden hier in *errno* die Zahl *0* schreiben. Diese kann als Identifier benutzt werden um zu testen, ob durch 0 geteilt wurde. Als Wert werden wir die REAL Zahl *NaN* zurückgeben.

read bei leerer Datei: Wir vergeben den Fehlercode *2* und schreiben bei Typ T eine *0* und bei B ein *false*. An dieser Stelle ist das erste mal wirklich notwendig auf den *errno* zurückzugreifen um den Fehler konkret zu verarbeiten.

Typkonflikte: Wir geben wie beim leeren lesen vor und schreiben diesmal *3* in *errno*. Konflikte können nur bei T und B auftreten, daher können wir so verfahren. Bei C in der Zuweisung kann es keine Probleme geben, da es keine statisch getypten Variablen gibt und es so zu keinem Konflikt kommen kann.