Blatt 04: Semantische Analyse

Carsten Gips, BC George (HSBI)

1 Zusammenfassung

Ziel dieses Aufgabenblattes ist die Erstellung einer Symboltabelle und eines einfachen Type-Checkers für eine fiktive statisch typisierte Sprache mit Expressions, Kontrollstrukturen und Funktionen.

2 Methodik

Sie finden im Sample Project eine Grammatik, die (teilweise) zu der Zielsprache auf diesem Blatt passt. Analysieren Sie diese Grammatik und vervollständigen Sie diese bzw. passen Sie diese an.

Erstellen Sie mit dieser Grammatik und ANTLR wieder einen Lexer und Parser. Definieren Sie einen AST und konvertieren Sie Ihren Parse-Tree in einen AST.

Es ist empfehlenswert, den Type-Checker dreiphasig zu realisieren:

1. Aufbauen der Symboltabelle und Prüfen von z.B. Deklaration/Definition vs. Benutzung (Variablen) usw.
2. Prüfen bei Funktionsaufrufen auf vorhandene/sichtbare Funktionsdefinitionen
3. Prüfen der verwendeten Typen

3 Sprachdefinition

Ein Programm besteht aus einer oder mehreren Anweisungen (*Statements*).

3.1 Anweisungen (*Statements*)

Eine Anweisung ist eine Befehlsfolge, beispielsweise eine Deklaration (Funktionen), Definition (Variablen, Funktionen), Zuweisung, ein Funktionsaufruf oder eine Operation. Sie muss immer mit einem Semikolon abgeschlossen werden. Eine Anweisung hat keinen Wert.

```
1  int a;           # Definition der Integer-Variablen a
2  int b = 10 - 5;  # Definition der Integer-Variablen b und Zuweisung des
       Ausdruckes 10-5 (Integer-Wert 5)
3  c = "foo";       # Zuweisung des Ausdrucks "foo" (String) an die Variable c (
       diese muss dazu vorher definiert und sichtbar sein)
4  func1(a, c);     # Funktionsaufruf mit Variablen a und c
```

Kontrollstrukturen und Code-Blöcke sowie **`return`**-Statements zählen ebenfalls als Anweisung.

3.2 Code-Blöcke und Scopes

Code-Blöcke werden in geschweifte Klammern eingeschlossen und enthalten eine beliebige Anzahl von Anweisungen.

Jeder Code-Block bildet einen eigenen Scope - alle Deklarationen/Definition in diesem Scope sind im äußeren Scope nicht sichtbar. Im Scope kann auf die Symbole des/der umgebenden Scopes zugegriffen werden. Symbole in einem Scope können gleichnamige Symbole im äußeren Scope verdecken.

```
# globaler Scope
int a = 7;
int d;
{   # erster innerer Scope
    int b = 7;      # b ist nur in diesem und im zweiten inneren Scope sichtbar
    foo(a);         # Funktionsaufruf mit der Variable a aus dem äußeren Scope
        (Wert 7)
    int a = 42;     # Variable verdeckt das a aus dem äußeren Scope
    foo(a);         # Funktionsaufruf mit der Variable a aus dem aktuellen
        Scope (Wert 42)
    {   # zweiter innerer Scope
        int c = 9;  # dieses c ist nur hier sichtbar
        foo(d);     # d wird im äußeren Scope gesucht, dort nicht gefunden und
            im nächsthöheren Scope gesucht (rekursiv)
    }
}
{   # dritter innerer Scope
    int b;          # dieses b hat mit dem b aus dem ersten inneren Scope
        nichts zu tun
    foo(a);         # Funktionsaufruf mit der Variable a aus dem äußeren Scope
        (Wert 7)
}
```

3.3 Ausdrücke (*Expressions*)

Die einfachsten Ausdrücke sind Integer- oder String-Literale. Variablen und Funktionsaufrufe sind ebenfalls Ausdrücke. Komplexere Ausdrücke werden mit Hilfe von Operationen gebildet, dabei sind die Operanden jeweils auch wieder Ausdrücke.

Ein Ausdruck hat immer einen Wert und einen Typ.

Die Operatoren besitzen eine Rangfolge, um verschachtelte Operationen aufzulösen. Sie dürfen daher nicht einfach von links nach rechts aufgelöst werden. Die Rangfolge der Operatoren entspricht der üblichen Semantik (vgl. Java, C, Python).

Es gibt in unserer Sprache folgende Operationen mit der üblichen Semantik:

3.3.1 Vergleichsoperatoren

Operation	Operator
Gleichheit	==
Ungleichheit	!=
Größer	>

Operation	Operator
Kleiner	<

Die Operanden müssen jeweils beide den selben Typ haben. Dabei sind `string` und **`int`** zulässig. Das Ergebnis ist immer vom Typ `bool`.

3.3.2 Arithmetische Operatoren

Operation	Operator
Addition / String-Literal-Verkettung	+
Subtraktion	−
Multiplikation	*
Division	/

Die Operanden müssen jeweils beide den selben Typ haben. Für + sind `string` und **`int`** zulässig, für die anderen Operatoren (−, *, /) nur **`int`**. Das Ergebnis ist vom Typ der Operanden.

3.3.3 Beispiele für Ausdrücke

```
10 - 5        # Der Integer-Wert 5
"foo"         # Der String "foo"
a             # Wert der Variablen a (Zugriff auf a)
a + b         # Ergebnis der Addition der Variablen a und b
func1(a, b)   # Ergebnis des Funktionsaufrufs
```

Ausdrücke werden nicht mit einem Semikolon abgeschlossen. Sie sind also Teil einer Anweisung.

3.4 Bezeichner

Werden zur Bezeichnung von Variablen und Funktionsnamen verwendet. Sie bestehen aus einer Zeichenkette der Zeichen `a-z`,`A-Z`, `0-9`. Bezeichner dürfen nicht mit einer Ziffer `0-9` beginnen.

3.5 Variablen

Variablen bestehen aus einem eindeutigen Bezeichner (Variablennamen). Den Variablen können Werte zugewiesen werden und Variablen können als Werte verwendet werden. Die Zuweisung erfolgt mithilfe des =-Operators. Auf der rechten Seite der Zuweisung können auch Ausdrücke stehen.

```
int a;          # Definition der Variablen a (Typ: Integer)
int a = 7;      # Definition und Initialisierung einer Variablen
a = 5;          # Zuweisung des Wertes 5 an die Variable a
a = 2 + 3;      # Zuweisung des Wertes 5 an die Variable a
```

Variablen müssen vor ihrer Benutzung (Zugriff, Zuweisung) definiert und im aktuellen Scope sichtbar sein. Die Initialisierung kann zusammen mit der Definition erfolgen.

Variablen können in einem Scope nicht mehrfach definiert werden.

```
int a = 42;

{
    a = 7;  # zulässig - a ist hier sichtbar, Zugriff auf globalen Scope
    b = 9;  # unzulässig - b ist nicht definiert

    int a;  # zulässig - erste Definition in diesem Scope
    int a;  # unzulässig - a ist in diesem Scope bereits definiert
}
```

3.6 Kommentare

Kommentare werden durch das Zeichen # eingeleitet und umfassen sämtliche Zeichen bis zum nächsten Newline.

3.7 Kontrollstrukturen

3.7.1 While-Schleife

While-Schleifen werden mit dem Schlüsselwort **`while`** eingeleitet. Sie bestehen im Weiteren aus einer Bedingung in runden Klammern und einem in geschweiften Klammern formulierten Code-Block.

Die Bedingung besteht aus einem Vergleichsausdruck.

```
while (<Bedingung>) {
    <Anweisung_1>
    <Anweisung_2>
}
```

```
int a = 10;

while (a > 0) {
    a = a - 1;
}
```

3.7.2 Bedingte Anweisung (If-Else)

Eine bedingte Anweisung besteht immer aus genau einer **`if`**-Anweisung, und einer oder keiner **`else`**-Anweisung.

Eine **`if`**-Anweisung wird mit dem Schlüsselwort **`if`** eingeleitet und besteht aus einer Bedingung in runden Klammern und einem in geschweiften Klammern formulierten Code-Block.

Die Bedingung besteht aus einem Vergleichsausdruck.

Eine **`else`**-Anweisung wird mit dem Schlüsselwort **`else`** eingeleitet. Auf das Schlüsselwort folgt in geschweiften Klammern formulierter Code-Block.

```
if (<Bedingung>) {
    <Anweisung_1>
    <Anweisung_2>
}
```

```
if (<Bedingung>) {
    <Anweisung>
} else {
    <Anweisung>
}
```

```
int a = 42;

if (a > 0) {
    a = a - 1;
    if (a > 0) {
        a = a - 1;
    }
} else {
    a = a + 1;
}
```

3.8 Funktionen

3.8.1 Funktionsdefinition

Eine Funktionsdefinition macht dem Compiler die Implementierung einer Funktion bekannt.

Sie gibt zunächst die Signatur der Funktion (den “Funktionskopf”) bekannt: Rückgabetyp, Funktionsname, Parameterliste.

Die Parameterliste ist eine Komma-separierte Liste mit der Deklaration der Parameter (jeweils Typ und Variablenname). Die Parameterliste kann auch leer sein.

Nach dem Funktionskopf folgt der Körper der Funktion als Code-Block.

Funktionen können in einem Scope nicht mehrfach definiert werden.

```
type bezeichner(type param1, type param2) {
    <Anweisung_1>
    <Anweisung_2>
    return <Bezeichner, Wert oder Operation>;
}
```

```
bool func1(int a, string b) {
```

```
    int c = a + f2(b);
    return c == 42;
}
```

3.8.2 Funktionsaufrufe

Funktionsaufrufe bestehen aus einem Bezeichner (Funktionsname) gefolgt von einer in Klammern angegebenen Liste der Argumente, die auch leer sein kann. Als Argumente können alle passend typisierten Ausdrücke dienen.

```
func1(5, var1)
func1(func2(), 1 + 1)
```

Die aufgerufene Funktion muss im aktuellen Scope sichtbar sein, der Funktionsaufruf muss zur Definition passen.

Die aufgerufene Funktion muss (im Gegensatz zum Zugriff auf Variablen) nicht vor dem ersten Aufruf definiert sein. Folgender Code ist also zulässig:

```
int a = 42;
if (a == 42) {
    foo(5);    # zulässig: foo ist erst nach diesem Aufruf definiert, aber in
        diesem Scope sichtbar
}

int foo(int a) {
    return a + 37;
}
```

3.9 Datentypen

Unsere Sprache hat drei eingebaute Datentypen:

Datentyp	Definition der Literale
`int`	eine beliebige Folge der Ziffern `0`–`9`
`string`	eine beliebige Folge von ASCII-Zeichen, eingeschlossen in `"`
`bool`	eines der beiden Schlüsselwörter `T` oder `F`

3.10 Beispiele

```
string a = "wuppie fluppie";
```

```
int a = 0;
if (10 < 1) {
    a = 42;
} else {
```

```
    foo();
}
```

```
int f95(int n) {
    if (n == 0) {
        return 1;
    } else {
        if (n == 1) {
            return 1;
        } else {
            return f95(n - 1) + f95(n - 2) + f95(n - 3) + f95(n - 4) + f95(n -
                5);
        }
    }
}

int n = 10;
f95(n);
```

4 Aufgaben

4.1 Grammatik und AST (2P)

Erstellen Sie eine ANTLR-Grammatik für die Zielsprache. Sie können dabei die Grammatik im Sample Project als Ausgangspunkt nutzen und diese anpassen und vervollständigen.

Definieren Sie einen AST für die Zielsprache. Welche Informationen aus dem Eingabeprogramm müssen repräsentiert werden?

Programmieren Sie eine Traversierung des Parse-Trees, die den AST erzeugt. Testen Sie dies mit den obigen Beispielprogrammen und definieren Sie sich selbst weitere Programme unterschiedlicher Komplexität für diesen Zweck.

4.2 Aufbau der Symboltabelle (2P)

Bauen Sie für den AST eine Symboltabelle auf. Führen Sie dabei die im ersten Lauf möglichen Prüfungen durch, beispielsweise ob referenzierte Variablen tatsächlich bereits definiert und sichtbar sind oder ob eine Variable oder Funktion in einem Scope mehrfach definiert wird oder ob Variablen als Funktion genutzt werden. Geben Sie erkannte Fehler auf der Konsole aus.

4.3 Symboltabelle: Funktionsaufrufe (1P)

Implementieren Sie einen zweiten Lauf. Dabei soll für Funktionsaufrufe geprüft werden, ob diese Funktionen bereits definiert sind und im Scope sichtbar sind. Geben Sie erkannte Fehler auf der Konsole aus.

4.4 Symboltabelle: Typprüfungen (5P)

Implementieren Sie einen dritten Lauf. Führen Sie die Typprüfung durch: Haben die Operanden in Ausdrücken die richtigen Typen, passen die Typen der Funktionsargumente, passen die Typen bei einer Zuweisung, … Geben Sie erkannte Fehler auf der Konsole aus. *Hinweis*: Sie brauchen hier nur die Typprüfung durchführen. Eine Typinferenz oder Typerweiterung o.ä. ist nicht notwendig.

> **Last modified:** f7ac9d2 (reformat using shorter lines, 2025-08-09)