Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 5: Syntaxdiagramme und Grammatikregeln

- Syntaxdiagramme
- Grammatikregeln (kontextfrei)
- Beispiele: Lexikalische Syntax

Inhalt

1. Syntax-Formalismen
2. Syntaxdiagramme
3. Beispiel: Zahlkonstanten in C++

Syntax-Formalismen (1)

- Ein C++-Programm ist eine Folge von Zeichen, aber nicht jede Folge von Zeichen ist ein C++-Programm.
- Ein Alphabet ist eine endliche, nicht-leere Menge, deren Elemente Zeichen heißen.
- Ein Wort über einem Alphabet ist eine endliche Folge von Zeichen des Alphabets.
- Eine formale Sprache (über einem Alphabet) ist eine (meist unendliche) Menge von Worten (über dem Alphabet).

Syntax-Formalismen (2)

- Es gibt verschiedene Formalismen, um klar und eindeutig eine formale Sprache (Menge von Zeichenfolgen) zu definieren, z.B.
  - ◇ Syntax-Diagramme
  - ◇ (Kontextfreie) Grammatik-Regeln

    BNF (Backus-Naur-Form) ist eine spezielle Syntax für kontextfreie Grammatikregeln (recht verbreitet).
- Syntaxdiagramme und kontextfreie Grammatiken haben die gleiche Ausdruckskraft, d.h. können die gleichen Mengen von Zeichenfolgen beschreiben.

Syntax-Formalismen (3)

- Als professioneller Programmierer sollte man die Definition der Programmiersprache lesen können.
- Die Grammatik ist eine Referenz in unklaren Fällen, hilft aber auch zum Verständnis der Sprache.

  Die syntaktischen Kategorien sind oft nützliche Konzepte.
- Es könnte ja sein, dass der Compiler einen Fehler enthält.

  Man sollte sich nicht zum "Sklaven" eines einzelnen Compilers machen. Der Compiler hat nicht immer recht (aber meistens schon). Es gibt ein unabhängiges Gesetz (den ISO-Standard). Zumindest hilft die Aufklärung eines Fehlers, ihn zukünftig zu vermeiden. Herumprobieren ("wie hätte er es denn gern?") ist alleine keine richtige Lösung.

Syntax-Formalismen (4)

- Mit kontextfreien Grammatiken definiert man nur eine Obermenge der C++-Programme, und stellt dann weitere einschränkende Forderungen, z.B.
  - ◇ jede Variable muss vor ihrer Verwendung deklariert sein.
- Kontextfreie Grammatiken (und damit auch Syntaxdiagramme) können solche Bedingungen nicht ausdrücken.

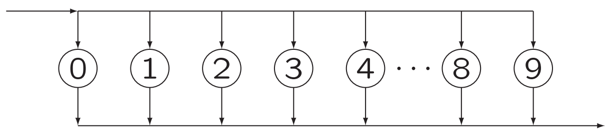
  Die Trennung dieser Aufgaben in die Compiler-Phasen Parser und semantische Analyse ist aber auch für die Modularisierung nützlich.

Inhalt

1. Syntax-Formalismen
2. Syntaxdiagramme
3. Beispiel: Zahlkonstanten in C++

Syntaxdiagramme (1)

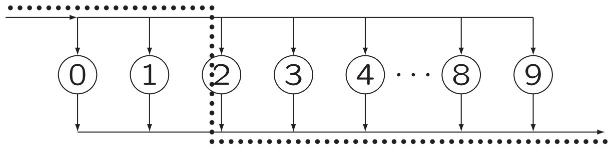
- Digit:



- Ein Syntaxdiagramm besteht aus:
  - ◇ Namen der definierten syntaktischen Kategorie,
  - ◇ Startknoten: Pfeil von außen in das Diagramm.
  - ◇ Zielknoten: Pfeil, der das Diagramm verlässt.
  - ◇ Kreise/Ovale und Rechtecke, die mit gerichteten Kanten verbunden sind.

Syntaxdiagramme (2)

- Die formale Sprache, die durch dieses Diagramm definiert wird, ist die Menge der Dezimalziffern $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
- Um zu prüfen, ob eine Eingabe, z.B. “2”, zu der Sprache “Digit” gehört, die durch das Diagramm definiert wird, muss man einen Weg durch das Diagramm finden, der der Eingabe entspricht:



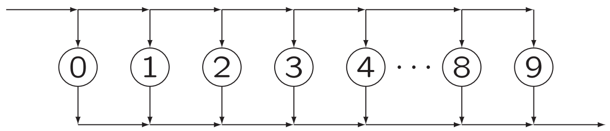
Syntaxdiagramme (3)

- Solche Diagramme können auf zwei Arten verwendet werden:
  - ◇ Um ein gültiges Wort der Sprache zu erzeugen, folgt man einem Pfad durch das Diagramm vom Start zum Ziel und druckt jedes Symbol in einem Kreis/Oval aus, den man durchläuft.
  - ◇ Um festzustellen, ob eine gegebene Eingabe syntaktisch korrekt ist, muss man einen Pfad durch das Diagramm finden, so dass beim Durchlaufen eines Kreises/Ovals das Zeichen darin das nächste Eingabezeichen ist.

Syntaxdiagramme (4)

- Jede Kante hat nur eine mögliche Richtung.

  Manchmal ist es für den Anfänger etwas schwierig, die Richtung einer Kante zu erkennen.
- Wenn alle Richtungen explizit gemacht sind, sieht das Diagramm so aus:



- Normalerweise wird der Pfeilkopf aber nicht in jedem Segment einer Kante wiederholt.

Syntaxdiagramme (5)

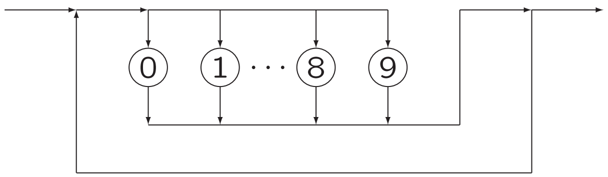
- Es gibt Verzweigungsknoten, an denen man verschiedene ausgehende Kanten benutzen kann.

  Z.B., nach der Eingangskante könnte man nach unten gehen, und die Ziffer 0 ausgeben/einlesen, oder man kann nach rechts gehen, um eine der Ziffern 1 bis 9 zu erhalten.
- Um eine gegebene Eingabe zu prüfen, hilft es meist, das nächste Eingabesymbol anzuschauen, um den richtigen Pfad auszuwählen.

  Dies ist, was der Compiler auch macht. Man versucht sicherzustellen, dass an jeder Verzweigung, die Kreise/Ovale, die man in verschiedenen Richtungen erreichen kann, disjunkt sind.
- Gibt es keinen passenden Pfad: Syntaxfehler.

Syntaxdiagramme (6)

- Syntaxdiagramme können Zyklen enthalten (es gibt ja unendlich viele gültige C++-Programme).
- Digit Sequence:



- Aufgabe: Finden Sie einen Pfad durch das Diagramm, um zu zeigen, dass "81" korrekt ist.

  Man darf die gleiche Kante mehrere Male durchlaufen.

Syntaxdiagramme (7)

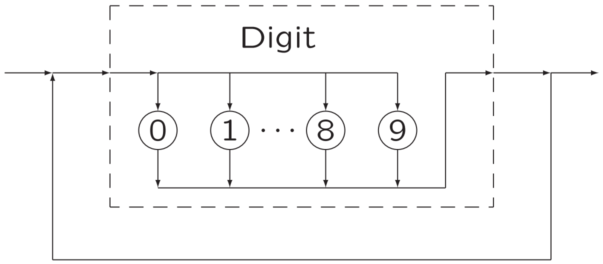
- Man kann an anderer Stelle definierte Diagramme als Module benutzen.
- Dies wird als Rechteck dargestellt, das den Namen des anderen Diagramms (def. synt. Kat.) enthält:

  Digit Sequence:

- Ein Rechteck steht also für eine syntaktische Kategorie (wie "Subjekt", "Prädikat", "Objekt").

Syntaxdiagramme (8)

- Ein Kasten kann durch das Diagramm ersetzt werden, für das er steht (er hat wie das Diagramm eine eingehende und eine ausgehende Kante).
- Digit Sequence:



Syntaxdiagramme (9)

- Natürlich muss man nicht explizit das Diagramm der benutzten syntaktischen Kategorie einfügen:
  - ◇ Man merkt sich einfach, wo man im übergeordneten Diagramm war (bei welchem Rechteck),
  - ◇ durchläuft dann das Diagramm für die syntaktische Kategorie, die in dem Rechteck steht,
  - ◇ anschließend (wenn man mit diesem Diagramm fertig ist) kehrt man zum übergeordneten Diagramm zurück, und folgt dort dem Pfeil, der das Rechteck verläßt.

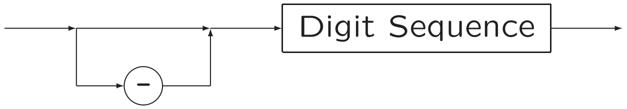
Syntaxdiagramme (10)

- Natürlich weiß man nach einiger Zeit, wofür z.B. die syntaktische Kategorie “Digit” steht, und braucht das zugehörige Diagramm dann nicht mehr nachzuschlagen.
- Jedes Diagramm definiert eine formale Sprache.

  D.h. eine Menge von Zeichenketten.
- Wenn man ein Rechteck durchläuft, kann man jedes Element der zugehörigen Sprache ausdrucken bzw. einlesen.

Syntaxdiagramme (11)

- Kreise/Ovale und Rechtecke können beide im gleichen Diagramm benutzt werden:
- Zahlkonstante: (nur Beispiel, nicht Teil von C++)



- Dieses Diagramm definiert eine Sprache, die z.B. folgende Zeichenfolgen enthält: `123`, `-45`, `007`.
- Dagegen gehören folgende Zeichenfolgen nicht zu der definierten Sprache: `+89`, `--5`, `23-42`, `0.56`.

Syntaxdiagramme (12)

- Syntaxdiagramme können “rekursiv” definiert sein, d.h. das Diagramm für eine synt. Kat. X kann selbst einen mit X beschrifteten Kasten enthalten.

  Entsprechend könnten sich auch zwei syntaktische Kategorien X und Y gegenseitig aufrufen. Es ist nicht verlangt, dass eine syntaktische Kategorie vor ihrer Verwendung definiert ist. Es müssen nur am Ende alle verwendeten syntaktischen Kategorien auch wirklich definiert sein.
- Der Pfad, den man für ein Wort der Sprache durch die Diagramme geht, muss immer endlich sein.

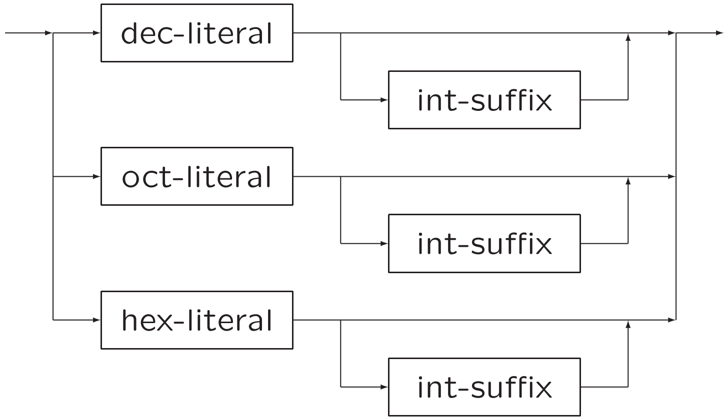
  Man kann jetzt zwar nicht mehr vorab die Diagramme für die Kästen vollständig einsetzen, aber man könnte noch die tatsächlich durchlaufenen Kästen nach Bedarf “expandieren”.

Inhalt

1. Syntax-Formalismen
2. Syntaxdiagramme
3. Beispiel: Zahlkonstanten in C++

Zahlkonstanten in C++ (1)

- integer-literal:



Zahlkonstanten in C++ (2)

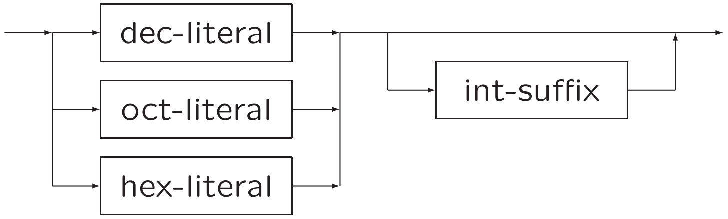
- Im C++ Buch von Stroustrup (dem Erfinder der Sprache) werden keine Syntaxdiagramme verwendet, sondern Grammatik-Regeln.
- Dort finden sich auf Seite 796 folgende Definition, die obigem Diagramm entspricht:

  integer-literal:
  decimal-literal *integer-suffix*$_{\text{opt}}$
  octal-literal *integer-suffix*$_{\text{opt}}$
  hexadecimal-literal *integer-suffix*$_{\text{opt}}$

- Es ist also eine Alternative pro Zeile angegeben, "$_{\text{opt}}$" markiert optionale Teile.

Zahlkonstanten in C++ (3)

- Die Grammatik-Notation ist natürlich wesentlich kompakter. Die Diagramme sind am Anfang anschaulicher. Hier könnte man wenigstens den Suffix noch zusammenfassen (äquivalentes Diagramm):
- integer-literal:

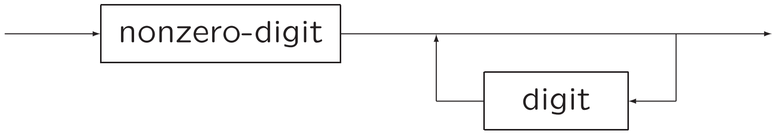


Zahlkonstanten in C++ (4)

- Da in C++ die Oktalschreibweise (zur Basis 8) dadurch gekennzeichnet ist, dass die Zahl mit 0 beginnt, dürfen normale Dezimalzahlen nicht mit 0 beginnen: `010` bedeutet 8.

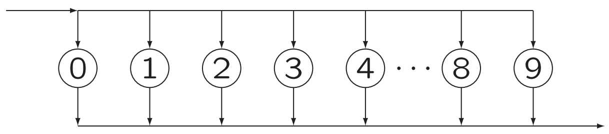
  Formal ist `0` auch Oktalschreibweise, aber in diesem Fall ist das Ergebnis das gleiche wie bei Dezimalschreibweise.

- dec-literal:

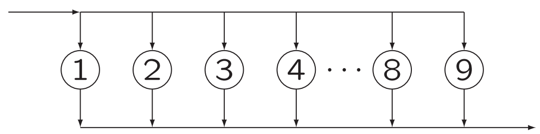


Zahlkonstanten in C++ (5)

- digit:



- nonzero-digit:



Zahlkonstanten in C++ (6)

- Grammatik-Regeln aus dem Buch von Stroustrup:

  digit: *one of*
  0 1 2 3 4 5 6 7 8 9

  nonzero-digit: *one of*
  1 2 3 4 5 6 7 8 9

- Normalerweise müßten die verschiedenen Alternativen untereinander geschrieben werden.

  Jede der eingerückten Zeilen definiert eine mögliche Ersetzung für die syntaktische Kategorie, die darüber genannt ist.

- Durch die Schlüsselworte “*one of*” wird markiert, dass hier jedes Zeichen einzeln eine Alternative ist.

Zahlkonstanten in C++ (7)

- Bemerkung/Exkurs: Auch in dieser Notation für Grammatik-Regeln wird unterschieden zwischen
  - ◇ syntaktischen Kategorien (kursiv) und
  - ◇ den letztendlich erzeugten Zeichen der definierten Sprache (normale Schrift bzw. “teletype”)

  (entspricht Rechtecken und Kreisen/Ovalen).
- Beispiel (gekürzt):

  selection-statement:
  `if` (*condition*) *statement*
  `if` (*condition*) *statement* `else` *statement*

Zahlkonstanten in C++ (8)

- Bemerkung/Exkurs, Forts.: Die Symbole für die syntaktischen Kategorien werden auch “Nichtterminalsymbole” genannt, die Zeichen der erzeugten Sprache “Terminalsymbole”.

  Weil die Zeichen der Zielsprache nicht weiter durch Anwendung von Grammatikregeln ersetzt werden können (die Ersetzung endet).

- Es gibt viele Notationen für Grammatikregeln:
  - ◇ Z.B. kann man syntaktischen Kategorien auch in spitze Klammern ⟨ . . . ⟩ schreiben.
  - ◇ Manche Autoren unterstreichen auch die Zeichen der erzeugten Sprache, oder schreiben sie in ’. . .’.

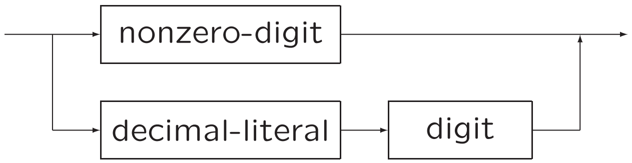
Zahlkonstanten in C++ (9)

- Zahlkonstanten in Dezimalschreibweise sind durch folgende Grammatikregeln definiert:

  decimal-literal:
  nonzero-digit
  decimal-literal digit

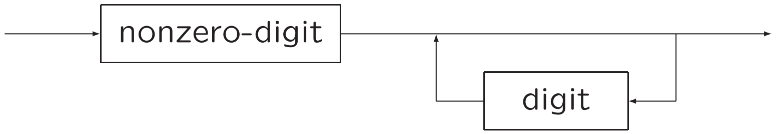
- Dies entspricht folgendem Syntaxgraphen:

  decimal-literal:



Zahlkonstanten in C++ (10)

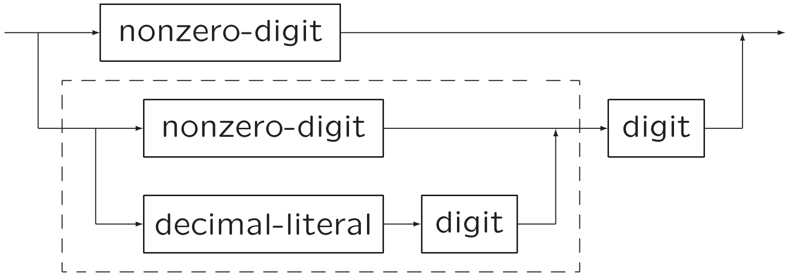
- Wie oben gesagt, sind rekursive Definitionen (d.h. die Verwendung einer syntaktischen Kategorie in ihrer eigenen Definition) durchaus erlaubt, aber in Syntaxdiagrammen recht selten.
- Im Beispiel ist der Syntaxgraph mit einem Zyklus äquivalent (d.h. definiert die gleiche Sprache) und übersichtlicher:



Zahlkonstanten in C++ (11)

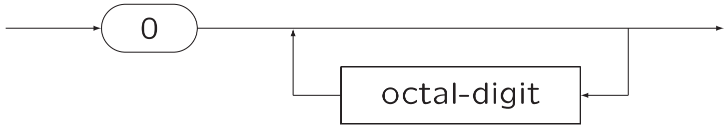
- In Grammatik-Regeln kann man dagegen keine Zyklen schreiben, und setzt intensiv Rekursion ein.
- Rekursives Diagramm nach einmaliger Einsetzung:

  decimal-literal:

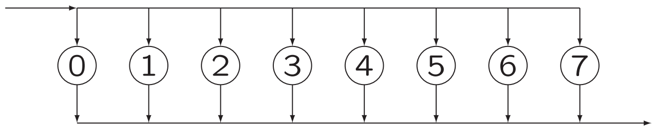


Zahlkonstanten in C++ (12)

- oct-literal:



- octal-digit:



Zahlkonstanten in C++ (13)

- Die entsprechenden Grammatik-Regeln sind:

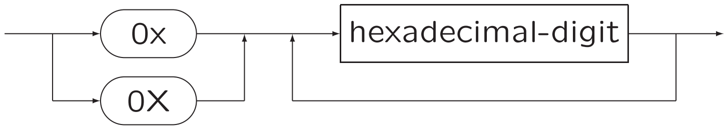
  octal-literal:
  0
  octal-literal octal-digit

  octal-digit: *one of*
  0 1 2 3 4 5 6 7

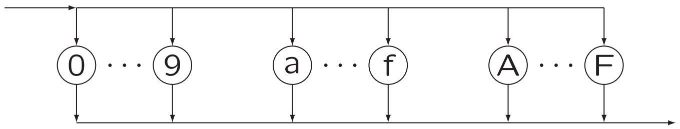
- Aufgabe: Zeichnen Sie einen Syntaxgraphen, der die Grammatikregeln für "octal-literal" direkt abbildet. Expandieren Sie den rekursiven Aufruf von "octal-literal" anschließend einmal (d.h. setzen Sie eine Kopie des Graphens für den Kasten ein).

Zahlkonstanten in C++ (14)

- hex-literal:

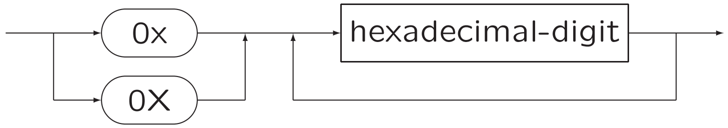


- hexadecimal-digit:

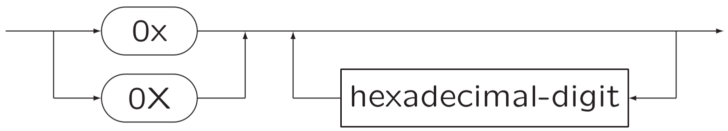


Zahlkonstanten in C++ (15)

- Aufgabe: Gibt es einen wesentlichen Unterschied zwischen den beiden folgenden Syntaxdiagrammen?
- hex-literal-1:



- hex-literal-2:



Zahlkonstanten in C++ (16)

- Die entsprechenden Grammatik-Regeln sind:

  hexadecimal-literal:
  `0x` *hexadecimal-digit*
  `0X` *hexadecimal-digit*
  hexadecimal-literal hexadecimal-digit

  hexadecimal-digit: *one of*
  `0 1 2 3 4 5 6 7 8 9`
  `a b c d e f`
  `A B C D E F`

Zahlkonstanten in C++ (17)

- Die Grammatik-Regeln für "integer-suffix" sind:

  integer-suffix:

  unsigned-suffix *long-suffix*$_{\text{opt}}$
  long-suffix *unsigned-suffix*$_{\text{opt}}$

  unsigned-suffix: *one of*

  `u U`

  long-suffix: *one of*

  `l L`

- Aufgabe: Entwickeln Sie ein äquivalentes Syntax-Diagramm.