Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 4: Lexikalische Syntax

- Compiler, Phasen, Scanner vs. Parser
- Leerplatz, Kommentare
- Bezeichner, Schlüsselworte
- Konstanten/Literale

Inhalt

Aufgabe eines Compilers (1)

- Übersetzung von Programmen aus einer Programmiersprache A in eine Programmiersprache B.
  - ◇ Eine Maschine für B existiert (Hardware, Interpreter, Compiler in implementierte Sprache C).
  - ◇ A ist für den Programmierer bequemer als B.

    Zumindest für bestimmte Aufgaben. Je nach Problem können unterschiedliche Sprachen besonders geeignet sein.
- Der Computer versteht die Sprache A über den Compiler.

  Der Compiler $A \rightarrow B$ macht aus einer Maschine für B eine Maschine für A.

Aufgabe eines Compilers (2)

Aufgabe:

- An einem Compiler sind drei Sprachen beteiligt:
  - ◇ Die Eingabesprache A (z.B. C++)
  - ◇ Die Ausgabesprache B (z.B. Maschinencode für Intel Pentium CPUs).
  - ◇ Die Sprache C, in der der Compiler selbst geschrieben wurde.
- Könnte es vielleicht Sinn machen, den Compiler für eine Sprache in dieser Sprache selbst zu schreiben, z.B. einen C++-Compiler in C++ zu schreiben?

Phasen eines Compilers (1)

- Compiler sind große Programme. Üblich: Einteilung in möglichst unabhängige Module/Arbeitsphasen:
  - ◇ Lexikalische Analyse (Scanner)
  - ◇ Syntaktische Analyse (Parser)
  - ◇ Semantische Analyse (u.a. Typprüfung)
  - ◇ Erzeugung von Zwischencode
  - ◇ Codeoptimierung
  - ◇ Codeerzeugung

    Besonders praktisch ist hier, daß man die von einem Modul zum nächsten übergebenen Daten leicht ausgeben kann.

Phasen eines Compilers (2)

- Trennung lexikalische Syntax (Scanner) vs. kontextfreie Syntax (Parser):
  - ◇ Der Scanner kondensiert mit einem einfachem (sehr effizienten) Verfahren die Eingabe (Zeichenfolge) in kürzere Folge von Wortsymbolen.
  - ◇ Das komplexere Analyseverfahren im Parser muß dann nur noch eine kürzere Eingabe verarbeiten.
  - ◇ Einfache Regel für Leerplatz: Kann zwischen je zwei lexikalischen Einheiten eingestreut werden.

Lexikalische Analyse (1)

- Eingabe: Programm als Folge von Zeichen.
- Ausgabe: Folge von Token (Wortsymbole).
- Leerplatz ("white space") und Kommentare werden entfernt.

  Leerplatz: Leerzeichen, Zeilenumbrüche, etc. Siehe unten.

- Zwischen manchen Token ist Leerplatz nötig, um sie zu trennen (z.B. `else x`), zwischen manchen ist er optional (z.B. `x=1`).

Lexikalische Analyse (2)

- Eingabe:

```
if(total_amount >= 100)
        // Keine Versandkosten
        shipping = 0;
else
        shipping = 5.95;
```

- Ausgabe:

  `if` `(` `Identifier` `Bin-Op` `Integer` `)`

  `Identifier` `Bin-Op` `Integer` `;`

  `else`

  `Identifier` `Bin-Op` `Real` `;`

Lexikalische Analyse (3)

- Manche Token haben außer ihrem Typ noch einen Wert, den der Scanner an den Parser übergibt:
  - ◇ Datentyp-Konstanten / Literale (z.B. `Integer`) haben einen Wert (z.B. 100).

    Eventuell führt der Scanner auch schon die Umwandlung von einer Ziffernfolge in die interne binäre Repräsentation durch.
  - ◇ Bezeichner (`Identifier`) haben einen Namen.

    Bezeichner werden in eine Symboltabelle eingetragen. Dort können weitere Daten hinterlegt werden, wie z.B. der Typ der Variablen.
  - ◇ Gibt es nur einen gemeinsamen Tokentyp für alle Operatoren, so muß der genaue Operator durch einen zusätzlichen Wert identifiziert werden.

Inhalt

Leerplatz, Kommentare (1)

- Zwischen zwei Token ist eine beliebige Folge von Leerzeichen, Tabulatorzeichen, Zeilenumbrüchen (Carriage Return, Linefeed), Formfeed (neue Seite) sowie Kommentaren erlaubt.

  Nach manchen Quellen auch ein "Vertical Tabulator" (wird sehr selten verwendet).
- Zeilenumbrüche, Einrückungen, und Kommentare können verwendet werden, um Programme lesbarer zu gestalten.
- Der Compiler ignoriert diese Dinge.

Leerplatz, Kommentare (2)

- Üblich ist folgende Einrückung:

```
anz_stellen = 1;
while(n >= 10) {
        n = n / 10;
        anz_stellen = anz_stellen + 1;
}
```

- Die abhängigen Anweisungen werden also um eine Tabulator-Position mehr eingerückt.

  Am verbreitetsten ist, alle 8 Zeichen eine Tabulator-Position zu haben. Ein einzelnes "Tab"-Zeichen wirkt dann also wie 8 Leerzeichen, genauer positioniert es auf die nächste durch 8 teilbare Spaltenposition (falls man bei 0 anfängt zu zählen). Man kann die Tabulartorbreite im Editor möglicherweise einstellen, aber ein Druckprogramm verhält sich dann eventuell anders.

Leerplatz, Kommentare (3)

- Editoren mit Syntax-Unterstützung für C++ können die Einrückung automatisch machen.

  Es gibt auch "Pretty Printer", die Programmtext nachträglich formatieren. Es kommt dabei allerdings nicht in allen Fällen das heraus, was man manuell gemacht hätte.

- Man sollte möglichst Zeilen breiter als 80 Zeichen vermeiden.

  80 Zeichen sind eine übliche Standardbreite für Editorfenster, und Druckprogramme sollten 80 Zeichen pro Zeile ausgeben können (noch sicherer: 79 Zeichen). Falls die Zeilen (z.B. durch Einrückungen) sehr lang werden, sollte man über eine Strukturierung mit Prozeduren nachdenken. Dies ist mein persönlicher Stil, Sie dürfen gerne anderer Meinung sein. Dem Compiler ist es egal.

Leerplatz, Kommentare (4)

- Zeilenstruktur, Einrückungen, u.s.w. werden schon in der lexikalischen Analyse entfernt.
- Der "eigentliche Compiler" sieht sie gar nicht mehr.
- Z.B. geschieht hier nicht, was der Programmierer erwartet:

```
if(x < 0)
        x = -x;
        cout << '-';
cout << x;
```

- Es gibt (in den meisten Compilern) nicht einmal eine Warnung.

Leerplatz, Kommentare (5)

- Kommentare haben in C++ zwei mögliche Formen:
  - ◇ Von // bis zum Ende der Zeile (neu in C++).
  - ◇ Von /* bis */ (geerbt von C).

    Man kann solche Kommentare nicht schachteln.
- Die zweite Form ist etwas gefährlich: Vergisst man, den Kommentar zu schließen, werden möglicherweise größere Teile des Programms übersprungen.

  Bis zum Ende des nächsten Kommentars. Es gibt normalerweise keine Warnung, weil Kommentare dieser Form auch verwendet werden, um bewußt einen Teil des Programmtextes "auszukommentieren" (temporär zu entfernen). Manche Compiler geben allerdings eine Warnung, wenn ein Kommentar einen Kommentarbeginn /* enthält.

Leerplatz, Kommentare (6)

- Speziell für C++ entwickelte Editoren (oder mit C++-Modus) stellen Kommentare in einer anderen Farbe als normalen Programmtext dar.
- Im allgemeinen ist es hilfreicher, größere Blöcke von Kommentar zu haben, als jede einzelne Zeile zu kommentieren.

  Klassisches schlechtes Beispiel: `i = i + 1; // Erhöhe i um 1.`
- Es gibt Programme, die speziell formatierte Kommentare auswerten, und daraus z.B. Programm-Dokumentation erstellen.

Inhalt

Bezeichner (1)

- Ein Bezeichner ("identifier") ist ein Name für eine Variable, eine Prozedur, einen Datentyp, etc.
- Ein Bezeichner ist eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt. Das Zeichen "_" zählt dabei als Buchstabe.

  Allerdings sollte man Bezeichner, die mit "_" beginnen, besser vermeiden (werden für spezielle Zwecke im Compiler verwendet).
- Beispiele:
  - ◇ Korrekt: `x`, `x2`, `X2B`, `das_ist_ein_Bezeichner`.
  - ◇ Nicht korrekt: `25m`, `KD#`, `a b`, `a/*sowas*/b`.

Bezeichner (2)

- Groß- und Kleinschreibung werden unterschieden, "`x`" und "`X`" sind zwei verschiedene Namen.

  Es ist guter Stil, leicht zu verwechselnde Namen zu vermeiden. Daher sollte man normalerweise nicht gleichzeitig beide Namen benutzen.
- In C++ können Namen beliebig lang sein, und alle Zeichen sind signifikant.

  D.h. der Compiler merkt sich immer den kompletten Namen, nicht nur einen Präfix (z.B. die ersten 31 Zeichen).
- Dies gilt aber möglicherweise nicht für den Linker (siehe nächste Folie).

Bezeichner (3)

- Das Format der Objektdateien und der Linker sind im allgemeinen nicht speziell für C++ gemacht.
- Es ist möglich, daß hier nur die ersten 8 Zeichen eines Namens abgespeichert werden.
- Z.B. meldet der Linker ggf. bei zwei Prozeduren
  - ◇ `abcdefgh1`
  - ◇ `abcdefgh2`

  das doppelt definierte Symbol "`abcdefgh`".

  Man sagt dann, daß nur 8 Zeichen des Namens signifikant sind, der Rest wird wie ein Kommentar behandelt (ignoriert).

Bezeichner (4)

- Namen sollten möglichst einheitlich nach bestimmten Konventionen gewählt werden, dann kann man sie sich leichter merken.

  Auch wenn mehrere Personen zusammen an einem Projekt arbeiten, sollte ein einheitlicher Programmierstil verwendet werden.

- Es ist in C/C++ üblich (aber keine Vorschrift), für Namen von Variablen und Prozeduren Kleinbuchstaben zu verwenden, und ggf. mehrere Teile durch "_" zu trennen.

  Bei einbuchstabigen Bezeichnern sind `i`, `j`, `k`, `n`, `m` üblicherweise ganze Zahlen, `c` ist ein Zeichen, `p` ein Pointer, und `x`, `y`, `z` sind Gleitkommazahlen.

Schlüsselworte (1)

- Manche Buchstabenfolgen haben eine spezielle Bedeutung in C++, z.B. `if`, `while` ("Schlüsselworte").
- Diese Buchstabenfolgen sind Ausnahmen zu der Regel, daß man beliebige Folgen von Buchstaben als Bezeichner für Variablen etc. verwenden darf.
- Sie sind von der Sprache C++ reserviert und heißen daher auch "reservierte Worte".
- Z.B. kann man keine Variable mit Namen "`if`" deklarieren:

```
int if; // Syntaxfehler!
```


Schlüsselworte (2)

- Solche Fehler geben oft ziemlich merkwürdige Fehlermeldungen des Compilers:
  - ◇ warning: '': ignored on left of 'int' when no variable is declared.
  - ◇ syntax error: missing ';' before 'if'
  - ◇ syntax error: ';'
- Bei Editoren, die etwas C++-Syntax kennen, werden reservierte Worte/Schlüsselworte in einer anderen Farbe als normale Bezeichner angezeigt.

Schlüsselworte (3)

- Ansonsten wird vom Programmierer erwartet, daß er die Schlüsselworte der Sprache auswendig kennt und als Bezeichner vermeidet.
- Das ist naturgemäß bei Sprachen einfacher, die wenig reservierte Worte haben:
  - ◇ C hat 32
  - ◇ C++ hat 74
  - ◇ SQL hat (je nach Dialekt) ca. 300.
- Es gibt auch Sprachen ganz ohne reservierte Worte.

Schlüsselworte (4)

`and`	`compl`	`export`	`namespace`
`and_eq`	`const`	`extern`	`new`
`asm`	`const_cast`	`false`	`not`
`auto`	`continue`	`float`	`not_eq`
`bitand`	`default`	`for`	`operator`
`bitor`	`delete`	`friend`	`or`
`bool`	`do`	`goto`	`or_eq`
`break`	`double`	`if`	`private`
`case`	`dynamic_cast`	`inline`	`protected`
`catch`	`else`	`int`	`public`
`char`	`enum`	`long`	`register`
`class`	`explicit`	`mutable`	`reinterpret_cast`

Schlüsselworte (5)

`return`	`try`	`xor`
`short`	`typedef`	`xor_eq`
`signed`	`typeid`	
`sizeof`	`typename`	
`static`	`union`	
`static_cast`	`unsigned`	
`struct`	`using`	
`switch`	`virtual`	
`template`	`void`	
`this`	`volatile`	
`throw`	`wchar_t`	
`true`	`while`	

Schlüsselworte (6)

- Zum Vergleich: Schlüsselworte in C:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

- Wenn man die Schlüsselworte einer Sprache alle erklären kann, hat man schon einen großen Teil der Sprache verstanden.

Inhalt

Konstanten/Literale

- Datenwerte kann man in Programmen als Konstanten aufschreiben:
  - `true` und `false` für den Datentyp `bool`.
  - z.B. `123` für den Datentyp `int`.
  - z.B. `1.23` oder `1.23E-4` für den Datentyp `double`.
  - z.B. `'a'` für den Datentyp `char`.
  - z.B. `"abc"` für Strings (Arrays von `char`).
- Manchmal spricht man auch von Literalen, um den Unterschied zu symbolischen Konstanten wie `pi` zu betonen.

Integer Konstanten (1)

- Eine ganze Zahl wird normalerweise dezimal als Folge von Ziffern `0` bis `9` dargestellt.
- Wenn die Zahl aber mit `0` beginnt, wird sie als Angabe im Oktalsystem (zur Basis 8) verstanden.

  Die Ziffern `8` und `9` sind dann natürlich verboten.
  Z.B. $123 = 1 * 10^2 + 2 * 10 + 3$ und $0123 = 1 * 8^2 + 2 * 8 + 3 * 1 = 83$.
- Man kann Zahlen auch hexadezimal aufschreiben, dazu muß die Konstante mit `0x` oder `0X` beginnen.

  Hexadezimal: Zur Basis 16. Zusätzliche Ziffern: `a/A` (10), `b/B` (11), `c/C` (12), `d/D` (13), `e/E` (14), `f/F` (15). Z.B. `0xFF`$= 15 * 16 + 15 = 255$.

Integer Konstanten (2)

- Angaben im Oktal- oder Hexadezimalsystem sind nützlich, wenn man die Zahlen eigentlich als Bitfolgen gebrauchen will.

  Oktal entspricht jede Ziffer 3 Bits, hexadezimal 4 Bits.
- Mit dem Suffix `L` (oder `l`) kann man eine Zahl explizit als "`long`" markieren, mit dem Suffix `U` (oder `u`) als "`unsigned`".

  Wenn eine Dezimalzahl-Konstante zu groß für `int` ist, wird sie automatisch als `long` aufgefasst. Bei Oktal-/Hexadezimal-Schreibweise wird der erste der folgenden Typen genommen, in den sie passt: `int`, `unsigned int`, `long`, `unsigned long`. Für `unsigned long` kann man die beiden Suffixe in beliebiger Reihenfolge konkatenieren, z.B. `0x123UL`.

Typumwandlungen (1)

- C ist sehr großzügig mit automatischen Typumwandlungen, deswegen ist der Typ von Konstanten oft nicht sehr wichtig.

  Das ist gefährlich: Die automatische Umwandlung ist nicht immer, was man beabsichtigt hat. Manchmal wäre eine Fehlermeldung besser.
- Man kann eine `long`-Konstante z.B. einem `char` ("very short integer") zuweisen:

```
char c = 48L;
```
- Wenn die Zahl zu groß ist, und der Compiler merkt es, wird man eine Warnung bekommen:

```
char c = 1000L;
```

Typumwandlungen (2)

- Warnungen unterscheiden sich von Fehlern dadurch, daß
  - ◇ ausführbarer Code erzeugt wird, und
  - ◇ man sie (normalerweise) abschalten kann.
- Schon hier merkt ein einfacher Compiler nicht mehr, daß ein paar Bits verloren gehen:

```
int n;
char c;

n = 1000;
c = n;
```


Gleitkomma-Konstanten (1)

- Eine Gleitkomma-Konstante (z.B. `12.34E-56`) besteht aus
  - ◇ einem ganzzahligem Anteil

    Dies ist eine Folge von Dezimalziffern.
  - ◇ einem Dezimalpunkt “.”,
  - ◇ einem gebrochenen Anteil

    Dies ist eine Folge von Dezimalziffern.
  - ◇ ein `e` or `E`,
  - ◇ einem Exponenten (zur Basis 10)

    Folge von Dezimalziffern, mit optional einem Vorzeichen.
  - ◇ Optional einen Typ-Suffix: `f`, `F`, `l`, `L`.

Gleitkomma-Konstanten (2)

- Der ganzzahlige Anteil oder der gebrochene Anteil können fehlen (aber nicht beide).
- Der Dezimalpunkt oder der Exponent (mit `e`/`E`) können fehlen (aber nicht beide).
- Z.B. sind legal: `12.3`, `12.`, `.34`, `1E0`, `1.E-2`, `.2E+5`.
- Gleitkomma-Konstanten haben den Typ `double`, nur der Suffix
  - ◇ `f`/`F` macht es zu `float`, bzw.
  - ◇ `l`/`L` zu `long double`.

Zeichenkonstanten (1)

- Zeichenkonstanten bestehen aus einem Zeichen in einfachen Anführungszeichen (Apostroph), z.B. `'a'`.

  Das Zeichen kann nicht ein einfaches Anführungszeichen selbst sein, auch kein Zeilenumbruch (Newline) oder Rückwärtsschrägstrich \.
- Anstelle eines Zeichens kann man auch eine der Escape-Sequenzen verwenden, die auf der nächsten Folie aufgelistet sind.
- Um ein Zeichen über den Oktalcode anzugeben, kann man ein, zwei, oder drei Oktalziffern verwenden, z.B. `'\0'`.

Zeichenkonstanten (2)

`\n`	Newline/Linefeed (LF)
`\t`	Horizontal tab (TAB, HT)
`\v`	Vertical tab (VT)
`\b`	Backspace (BS)
`\r`	Carriage Return (CR)
`\f`	Formfeed (FF)
`\a`	Audible alert (BEL)
`\\`	Backslash (\)
`\?`	Fragezeichen (?)
`\'`	Einfaches Anführungszeichen/Apostroph (')
`\"`	Doppeltes Anführungszeichen (")
`\`*ooo*	Character with code *ooo* (in octal)
`\x`*hh*	Character with code *hh* (in hexadecimal)

Zeichenkonstanten (3)

- Beachte: Falls man ASCII verwendet, steht `'0'` für die Zahl 48 und nicht die Zahl 0!

  Allgemein hängt die Umwandlung von Zeichen in Zahlen von der zugrundeliegenden Zeichencodierung ab.

- Konstanten vom Typ `wchar_t` werden durch ein vorangestelltes `L` gekennzeichnet, z.B. `L'ab'`.

  Im Innern der Anführungszeichen stehen dann meist mehrere Zeichen, die zusammen ein Zeichen des erweiterten Zeichensatzes beschreiben.

String Konstanten (1)

- Eine Zeichenketten-Konstante (String) ist eine Folge von Zeichen in (doppelten) Anführungszeichen z.B. `"abc"`.
- Die oben aufgelisteten Escape-Sequenzen können auch in Zeichenketten-Konstanten verwendet werden, z.B. `"eine Zeile\n"`.

  Bei der Oktal-Angabe des Zeichencodes gibt man am besten immer drei Ziffern an, sonst kann eine zufällig folgende Ziffer hinzugerechnet werden. Bei der Hexadezimalangabe muß man immer darauf achten, daß das folgende Zeichen keine Hexadezimalziffer ist, da es hier keine maximale Anzahl von Ziffern gibt.

String Konstanten (2)

- Zeichenketten-Konstanten, die direkt aufeinander folgen (also nur durch Leerplatz getrennt sind), werden in eine Zeichenkette verschmolzen.

  Auf diese Art kann man auch String-Konstanten aufschreiben, die länger als eine Zeile sind.

- Intern werden Strings als Folge der Zeichencodes, gefolgt von einem Null-Byte als Endmarkierung repräsentiert.
- `"abc"` hat z.B. den Typ `const char[4]` und besteht aus dem Zeichen `'a'`, `'b'`, `'c'`, `'\0'` in aufeinander folgenden Speicherzellen.

Inhalt

1. Compiler: Phaseneinteilung, Lexikalische Analyse
2. Leerplatz, Kommentare
3. Bezeichner, Schlüsselworte
4. Datentyp-Konstanten (Literale)
5. Operatoren, weitere Token

Operatoren (1)

- `+` (Addition), `-` (Subtraktion), `/` (Division),
  `*` (Multiplikation, Dereferenzierung),
  `%` (Divisionsrest/Modulo).
- `==` (gleich), `!=` (ungleich), `<` (kleiner), `>` (größer),
  `<=` (kleinergleich), `>=` (größergleich).
- `&&` (logisches und), `||` (log. oder), `!` (log. nicht).
- `&` (Bit-und, Referenz), `|` (Bit-oder), `^` (Bit-XOR),
  `~` (Bit-Komplement), `<<` (Linksshift, Ausgabe),
  `>>` (Rechtsshift, Eingabe).

Operatoren (2)

- `=` Zuweisung.
- `++` (Inkrement $+1$), `--` (Dekrement -1).
- `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
  (Abkürzungen für Zuweisungen).
- `,` (Sequenz).
- `.` (Selektion einer Struktur/Klassen-Komponente),
  `->` (Dereferenzierung+Komponenten-Selektion).
- Mit Schlüsselworten und den Zeichen auf der folgenden Folie können noch weitere Operatoren gebildet werden.

Weitere Token

- `;` (Ende einer Anweisung).
- `{`, `}` (begin, end).
- `(`, `)` (Klammern).
- `[`, `]` (Array Klammern).
- `<`, `>` (für Typecasts, Templates).
- `?`, `:` (Bedingte Ausdrücke).

  `:` wird auch im `switch`-Statement verwendet.
- `::` (Gültigkeitsbereich).

Trigraph-Sequenzen

- Nicht alle Programmierer verwenden einen Zeichensatz, der Zeichen wie { und } enthält.
- Deswegen sind die folgenden Ersetzungen möglich:

{	??<
}	??>
[	??(
]	??)

#	??=
\	??/
^	??'
|	??!
~	??-

- Die automatische Ersetzung gilt auch innerhalb von Strings, deshalb muß man ?? vermeiden (→ \?\?).

Operator-Schlüsselworte

- Aus dem gleichen Grund wurden Schlüsselworte als Ersatz für Operatoren eingeführt:

&&	and
&=	and_eq
&	bitand
||	or
|=	or_eq
|	bitor
^	xor
^=	xor_eq
!	not
!=	not_eq
~	compl

Längste Präfixe

- Die lexikalische Analyse liefert als nächstes Token immer den längsten Präfix vom Rest der Eingabe, der noch ein gültiges Token ist.

  D.h. die lexikalische Analyse liest so lange weitere Zeichen ein, wie das aktuelle Token sich noch verlängern läßt. Erst wenn das aktuelle Token zusammen mit dem nächsten Zeichen kein gültiges Token mehr wäre, wird das aktuelle Token für beendet erklärt (und an den Parser ausgeliefert). Das nächste Zeichen gehört dann schon zum nächsten Token (oder ist Leerplatz, der übersprungen wird).
- Z.B. würde bei der Eingabe +++ zuerst der Operator ++ geliefert, und danach der Operator +.
- Man sollte solchen kryptischen Code vermeiden.