⟨program⟩ ::= {⟨statement⟩*}
⟨statement⟩ ::= ⟨assignment⟩ | ⟨conditional⟩ | ⟨loop⟩
⟨assignment⟩ ::= ⟨identifier⟩ = ⟨expr⟩;
⟨conditional⟩ ::= **if** ⟨expr⟩ {⟨statement⟩+} |
    if ⟨expr⟩ {⟨statement⟩+} **else** {⟨statement⟩+}
⟨loop⟩ ::= **while** ⟨expr⟩ {⟨statement⟩+}
⟨expr⟩ ::= ⟨identifier⟩ | ⟨number⟩ | (⟨expr⟩) |
    ⟨expr⟩ ⟨operator⟩ ⟨expr⟩

⟨operator⟩ ::= $+ \mid - \mid * \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq$
⟨identifier⟩ ::= ⟨letter⟩ ⟨ld⟩*
⟨ld⟩ ::= ⟨letter⟩ | ⟨digit⟩
⟨number⟩ ::= ⟨digit⟩+
⟨letter⟩ ::= a | b | c | . . . | z
⟨digit⟩ ::= 0 | 1 | 2 | . . . | 9

Abstrakte/konkrete Syntax, Pragmatik

```
while (x == y)
   x = ...;
```

```
while (x == y)
   {
      ...
   }
```

```
while x = y do
   begin
      ...
   end
```

```
if x = y then
   ...
end
```

```
if x = y then
   ...
else
   ...
end
```

```
while x = y do
   ...
end
```


Statische und dynamische Semantik

Syntax definiert wohlgeformte Programme.
Semantik definiert die Bedeutung wohlgeformter Programme.

Nicht alle wohlgeformten Programme haben eine Bedeutung.
Beispiel: **while** (1.75) { . . . }

Wohlgeformte Programme mit Bedeutung werden *vor* der Ausführung von solchen ohne Bedeutung unterschieden.

Statische Semantik definiert, welche wohlgeformten Programme Bedeutung haben.

Dynamische Semantik definiert die Effekte der Ausführung wohlgeformter Programme mit Bedeutung.

Formale Semantik

Semantik $\neq$ Implementierung

Meta-Sprache zur formalen Spezifikation der Semantik basiert auf mathematischen Methoden.

Formale Semantik gut als Referenz, aber meist schwer lesbar.

Operationale Semantik: Pseudo-Implementierung auf virtueller Maschine.

Axiomatische Semantik: Zustandsübergänge mittels Logik beschrieben.

Denotationale Semantik: Zustandsübergänge und aktueller Speicherzustand durch Funktionen beschrieben.

Axiomatische Semantik

$\{Precondition\}$ *Statement* $\{Postcondition\}$
Beispiel: $\{y \geq 0\}$ $x = y + 1;$ $\{x > 0\}$

{wahr}
if $x \geq y$ **then** max = x; **else** max = y;
$\{(max = x \text{ und } x \geq y) \text{ oder } (max = y \text{ und } y > x)\}$

{P} S1; S2; {Q} wenn {P} S1; {R} und {R} S2; {Q}

{P} **while** B **do** L {P und nicht B} wenn {P} L {P}

Denotationale Semantik

dsem(S1; S2;, mem) = mem2 wenn dsem(S1;, mem) = mem1
und dsem(S2;, mem1) = mem2

dsem(**if** B **then** L1 **else** L2, mem) = mem1 wobei
mem1 = dsem(L1, mem) wenn mem(B) = wahr;
mem1 = dsem(L2, mem) wenn mem(B) = falsch

dsem(**while** B **do** L, mem) =
mem wenn mem(B) = falsch;
dsem(**while** B **do** L, dsem(L, mem)) wenn mem(B) = wahr

Interpretation und Übersetzung

Sowohl Interpreter als auch traditionelle Computer führen folgende Schritte in einer Schleife aus:

1. Lade den nächsten Befehl (oder das nächste Statement)
2. Bestimme, was zu tun ist
3. Führe die Aktionen aus

Programme in einer high-level Sprache werden mittels Compiler, Linker und Loader in äquivalente Programme in einer lower-level Sprache übersetzt; diese werden dann interpretiert.

Interpretation und Übersetzung (2)

Reine Übersetzung: übersetzte Programme ("native code") von der Hardware ausgeführt (hohe Ausführungsgeschwindigkeit, statische Typüberprüfungen möglich)

Reine Interpretation: Programmtext direkt vom Interpreter ausgeführt (portabel, interaktive Softwareentwicklung unterstützt, oft Speichereffizient)

Mischformen: Programme in Zwischencode übersetzt; dieser von einem Interpreter ausgeführt (Kompromiss)

Mehrere Übersetzungsschritte (Macros) können die Ausdruckskraft einer Sprache erhöhen.

Binden

Das Zuweisen eines Wertes an ein Attribut (Name, Typ, ...) eines Sprachelements (Variable, Statement, ...) heisst *Binden*.

Eigenschaften von Bindungen:

- Zeipunkt des Bindens (Zeitpunkt der Sprachdefinition, Implementierung, Übersetzung oder Ausführung)
- Stabilität der Bindung (fix oder änderbar)

Eine fixe Bindung vor der Ausführung ist *statisch*,
eine änderbare Bindung während der Ausführung *dynamisch*.
Fixe Bindungen während der Ausführung sind auch möglich.

Variablen

Variable ist Tupel ⟨Name, Scope, Typ, l-Wert, r-Wert⟩

Name: Zeichenfolge zur Bezeichnung der Variablen in einem Programm (oder Programmabschnitt)

Scope (oder Gültigkeitsbereich): Der Programmabschnitt in dem der Name bekannt (und gültig) ist

Typ: Angaben zum Wertebereich der Variablen

l-Wert: Die mit der Variable assoziierte Speicheradresse

r-Wert: Der kodierte Wert, der an der durch den l-Wert bestimmten Adresse gespeichert ist

Namen und Gültigkeitsbereiche

Ein Variablenname wird oft durch eine *Deklaration* eingeführt. Der Gültigkeitsbereich erstreckt sich von der Deklaration bis zu einem durch die Sprachdefinition bestimmten Ende.

Dynamic scoping: Gültigkeitsbereich erstreckt sich bis zur erneuten Ausführung einer Deklaration desselben Namens (APL, LISP, SNOBOL4, ...)

Static (lexical) scoping: Gültigkeitsbereiche werden durch die lexikalische Programmstruktur bestimmt (C, Ada, ...)

In blockstrukturierten Sprachen endet der Gültigkeitsbereich mit dem Block, in dem die Variable deklariert wurde.
Innere Blöcke können Namen in äusseren Blöcken verdecken.

Dynamic scoping

```
{  /* Block A */                          {  /* Block B */
   int x;                                    int x;
   ...                                       ...
}                                         }
...                                       ...

                  {  /* Block C */
                     ...
                     x = ...;
                     ...
                  }
```

Static scoping

```
#include <stdio.h>
main()
{  int x, y, z;
   scanf("%d %d %d", &x, &y, &z);
      /* drei Dezimalzahlen werden gelesen und
         in den l-Werten von x, y und z gespeichert */
   {  /* Block zum Vertauschen von x und y */
      int z;
      z = x;
      x = y;
      y = z;
   }
   printf("%d %d %d", x, y, z);
}
```


Typen

Der Typ einer Variablen beschreibt die *Menge der Werte*, die im l-Wert der Variable gespeichert werden können, und die *Operationen*, die angewendet werden können, um solche Werte zu erzeugen, zu modifizieren und darauf zuzugreifen.

Einige Typnamen werden zum Zeitpunkt der Sprachdefinition an Wertemengen und Operationenmengen gebunden.
Werte und Operationen werden zum Zeitpunkt der Sprachimplementation an Maschinen-Repräsentationen gebunden.

Durch Typdeklarationen werden neue Typen zur Übersetzungszeit eingeführt.

Z.B. Pascal: **type** vector = **array**[1..10] **of** integer

Benutzerdefinierter Typ in C++

```
class stack_of_char {
public:
   stack_of_char(int sz) {
      top = s = new char[size = sz];
   }
   ~stack_of_char() {delete[] s;}
   void push(char c) {*top++ = c;}
   char pop() {return *--top;}
   int length() {return top - s;}
private:
   int size;
   char* top;
   char* s;
};
```


Statische Typisierung

Eine Sprache ist *statisch typisiert* wenn Typen statisch an Variablen gebunden werden (C, Modula-2, Ada, Java, . . .).

Z.B. Pascal: **var** x, y: integer;
c: character;

Statische Typisierung ermöglicht statische Typüberprüfungen.

Z.B.: x := c; ist illegal

Variablentypen müssen nicht immer explizit spezifiziert sein.

Z.B.: in FORTRAN bestimmt der Variablenname den Typ;
in ML wird der Typ durch *Typinferenz* ermittelt.

Dynamische Typisierung

Eine Sprache ist *dynamisch typisiert* wenn Typen erst zur Laufzeit an Variablen gebunden werden (LISP, Smalltalk, . . .).
Solche Variablen sind *polymorph*.

Ein Typ bleibt an eine Variable gebunden, bis ein neuer Wert an die Variable zugewiesen wird. Dynamische Typinformation muss zur Laufzeit mitgeführt werden.

Dynamische Typüberprüfungen erfolgen zur Laufzeit.
Z.B.: x := y + 3; ist illegal wenn y vom Typ "String" ist.
y kann vom Typ "Integer" oder "Real" sein.

Eine Sprache, in der keine Typüberprüfung erfolgt, ist *untypisiert* (Assembler).

l-Werte

Der Speicherbereich wird während der Lebenszeit eines Daten-Objekts benötigt.
Das Binden eines Speicherbereichs (l-Wertes) an eine Variable heißt *Speicher-Allokation*, das Aufheben der Bindung am Ende der Lebenszeit *Speicher-Deallokation*.

Man spricht von *statischer Allokation* wenn die Allokation vor der Programmausführung und die Deallokation nach Programmbeendigung erfolgt.
Sonst spricht man von *dynamischer Allokation*.

Dynamische Allokation (bzw. Deallokation) erfolgt entweder *explizit*, oder *implizit* bei Erreichen einer Variablendeklaration (bzw. am Ende des Gültigkeitsbereichs einer Variablen).

r-Werte

Der r-Wert einer Variablen ist in imperativen Sprachen meist dynamisch änderbar; nicht aber in funktionalen und logischen Sprachen.

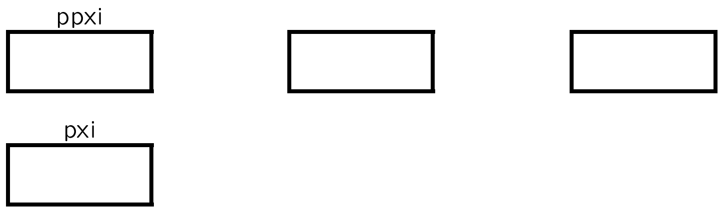
In den meisten Sprachen kann man Konstanten definieren; das sind Namen für r-Werte.
Z.B. in Pascal erfolgt die Bindung zwischen Name und r-Wert vor der Ausführung, in C oder Ada zur Laufzeit.

Einige Sprachen (wie ML) verlangen, dass jede Variable mit einem r-Wert initialisiert wird; andere Sprachen unterstützen die Initialisierung bei der Variablendeklaration.

Meist ist nicht definiert, welchen r-Wert eine uninitialisierte Variable hat (Fehlermeldung, implizite Initialisierung, . . .).

Zeiger

Ein Zeiger ist ein r-Wert, der als l-Wert verwendbar ist.



type PI = ^integer;	**type** PPI = ^PI;	int x = 0;
var pxi: PI;	**var** ppxi: PPI;	int* px = &x;
. . .	. . .	int** ppx = &px;
new(pxi);	**new**(ppxi);	int y = *px;
pxi^ = 0;	ppxi^ = pxi;	**ppx = 5;

Routinen

Routine hat Namen, Gültigkeitsbereich, Typ, l-Wert, r-Wert

Kopf: Name + Typen = *Signatur* (oft auch Parameternamen)
Z.B.: fun: $T_1 \times T_2 \times \cdots \times T_n \to R$

Rumpf: *lokale* Deklarationen + ausführbare Anweisungen
r-Wert entspricht Rumpf, l-Wert dessen Speicheradresse

Eine Sprache behandelt Routinen als *first class Objekte* wenn sie Routine-Variablen (bzw. Zeiger auf Routinen) unterstützt.

Aufruf: spezifiziert aktuelle Parameter; z.B.: i = sum(3);

C-Funktion und Zeiger auf Funktion

```
int sum(int n)
{
   int i, s;
   s = 0;
   for (i = 1; i <= n; i++)
      s = s + 1;
   return s;
}
```

```
int i;
int (*ps)(int);
...
ps = &sum;
i = (*ps)(5);
   /* i = sum(5); */
```


Routinen: Deklaration vs. Definition

```
int A(int x, int y);  /* Deklaration von A */
float B(int z)        /* Definition von B */
{  int w, u;
   ...
   w = A(z, u);       /* A ist hier sichtbar */
   ...
};
int A(int x, int y)   /* Definition von A */
{  float t;
   ...
   t = B(x);          /* B ist hier sichtbar */
   ...
};
```


Aufruf von Routinen

Instanz einer Routine: Code-Segment + *activation record* (*stack frame*)

Umgebung: lokale Umgebung = Objekte im activation record
nichtlokale Umgebung = alle anderen sichtbaren Objekte

Rekursion: ein neuer activation record kann angelegt werden bevor die Ausführung der Routine beendet ist. Bindung zwischen Code-Segment und activation record dynamisch.

Formale Parameter: Parameter im Kopf einer Routine

Aktuelle Parameter: Parameter (Argumente) im Aufruf

Bindung von Parametern

```
procedure Example(A: T1; B: T2 := W; C: T3);
```
-- Ada-Prozedur mit Default-Wert für Parameter `B`

```
Example(X, Y, Z);
```
-- Aufruf; Parameter über Position assoziiert

```
Example(C => Z, A => X, B => Y);
```
-- Parameter über Namen assoziiert

```
Example(X, C => Z);
```
-- Mischform; Default-Wert wird verwendet

Generische Routinen

```
template <class T> void swap(T& a, T& b)
\* generische Routine zum Vertauschen von a und b */
{
  T temp = a;
  a = b;
  b = temp;
}
```

```
int i, j;
...
swap(i, j);
```

```
float f, g;
...
swap(f, g);
```


Überladen und Aliasing

Ein Name mit mehr als nur einer Bedeutung ist *überladen*.

Zwei Namen, die dasselbe Objekt bezeichnen, sind *aliases*.

```
int i, j, k;
float a, b, c;
...
i = j + k;
a = b + c;
a = b + c + b();
a = b() + c + b(i);
```

```
int x = 0;
int *i = &x;
int *j = &x;
...
*i = 10;
```