1. Überblick

Kurze Geschichte des Übersetzerbaus

Früher Geheimwissenschaft, heute eines der am besten erforschten Informatikgebiete

1957	Fortran	Erste Compiler (arithmetische Ausdrücke, Anweisungen, Prozeduren)
1960	Algol	Erste saubere Sprachdefinition (Grammatiken in Backus-Naur-Form, Blockstruktur, Rekursion)
1970	Pascal	Benutzerdefinierte Typen, virtuelle Maschinen (P-Code)
1985	C++	Objektorientierung, Exceptions, Templates
1995	Java	Just-in-time-Compilation

Wir betrachten hier nur den imperativen Übersetzerbau

Für funktionale Sprachen (z.B. Lisp) oder logische Sprachen (z.B. Prolog) sind andere Techniken nötig.

Wozu lernen wir Übersetzerbau?

Gehört zur Allgemeinbildung jedes Informatikers

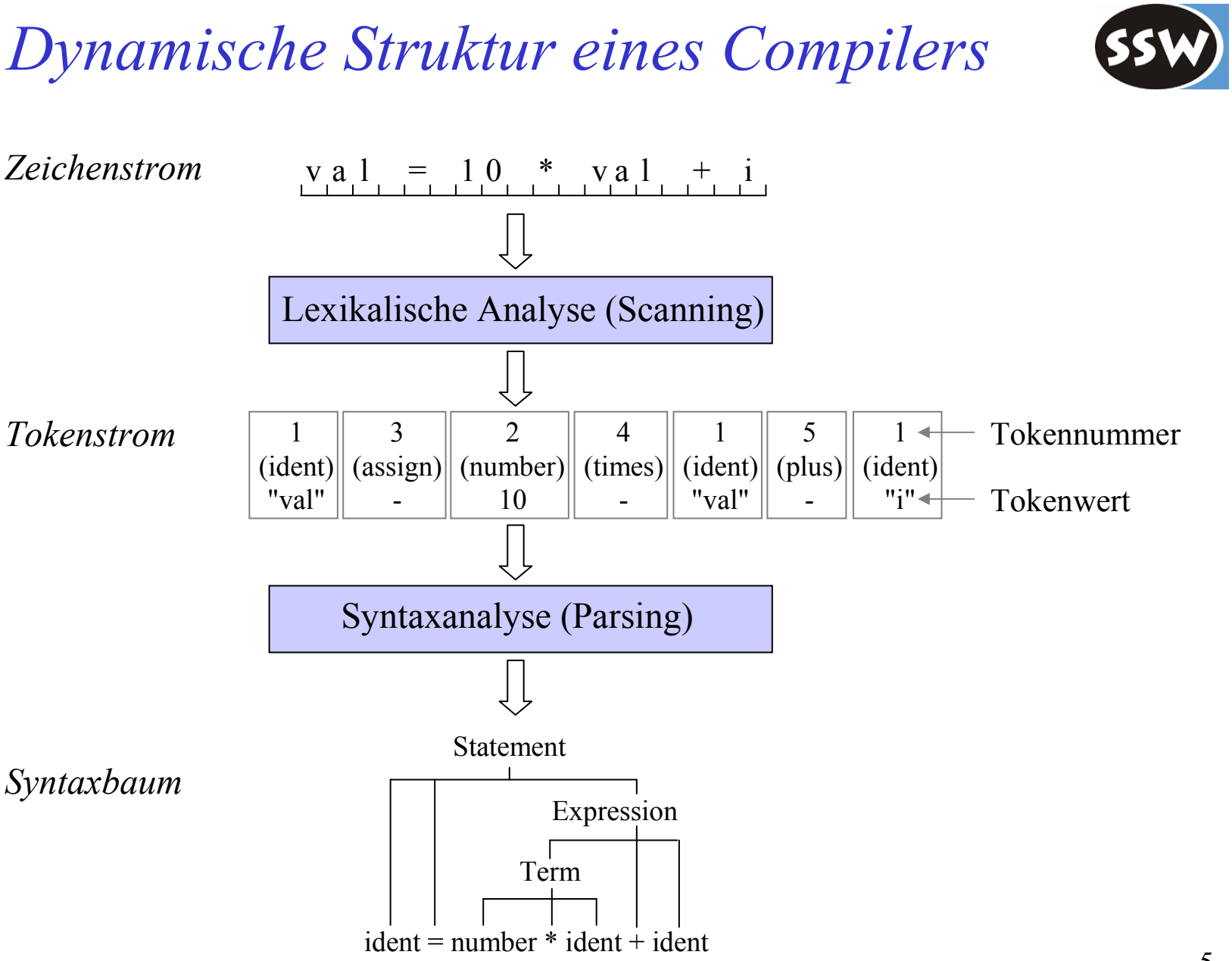
- Wie funktioniert ein Compiler?
- Wie funktioniert ein Computer auf Maschinenebene?
  (Instruktionssatz, Register, Adressierungsarten, Laufzeitdatenstrukturen, ...)
- In welchen Code werden Sprachkonstrukte übersetzt? (Gefühl für Effizienz)
- Gefühl für gutes Sprachdesign; Umgang mit Grammatiken
- Gibt Gelegenheit für ein nichttriviales Programmierprojekt im Studium

Übersetzerbau-Kenntnisse sind auch im Software-Engineering nützlich

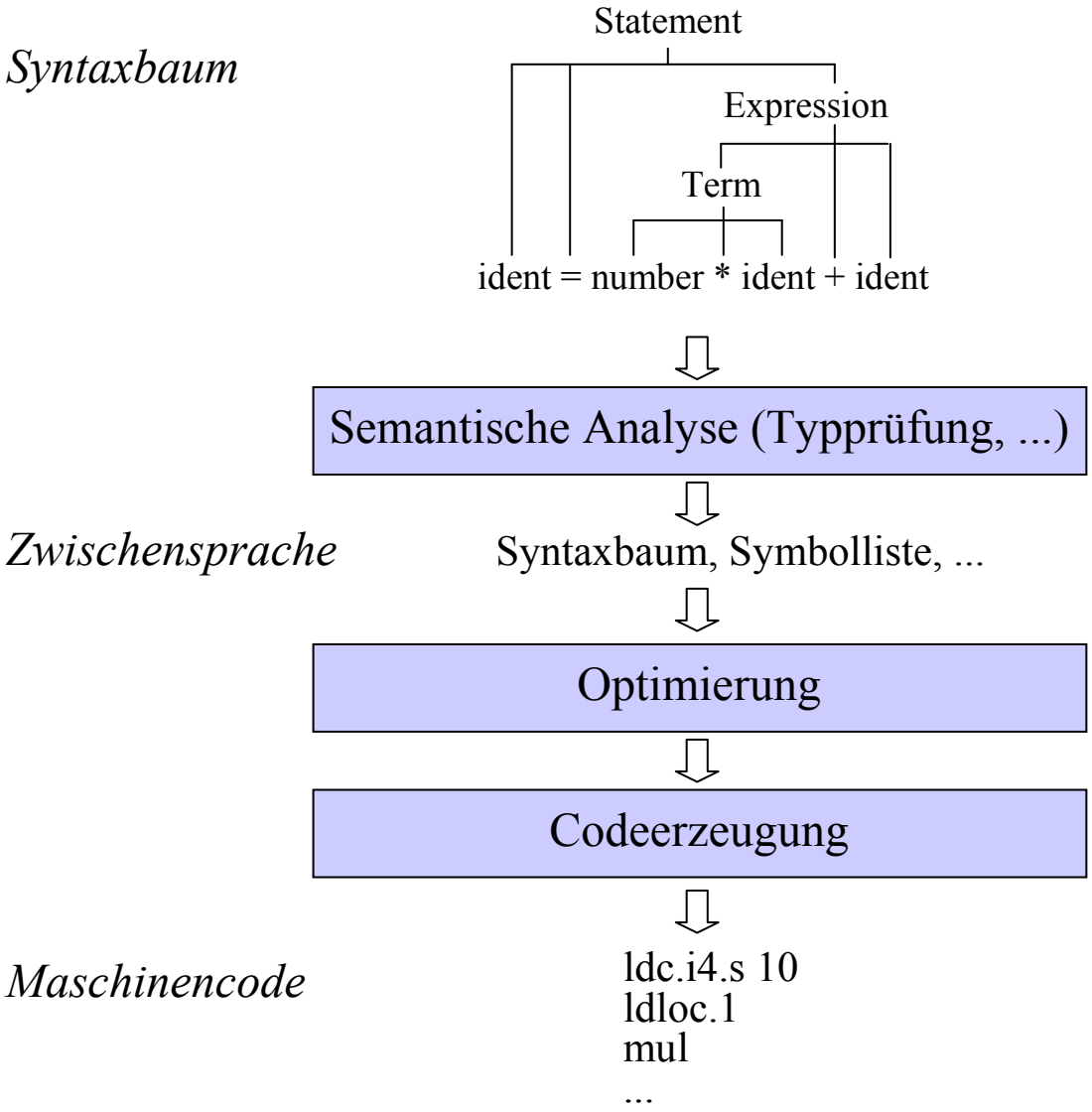
- Lesen syntaktisch strukturierter Kommandoparameter
- Lesen strukturierter Dateien (z.B. XML-Dateien, Stücklisten, Bild-Dateien, ...)
- Suchen in hierarchischen Namensräumen
- Interpretative Abarbeitung von Codes
- ...

1. Überblick

Dynamische Struktur eines Compilers

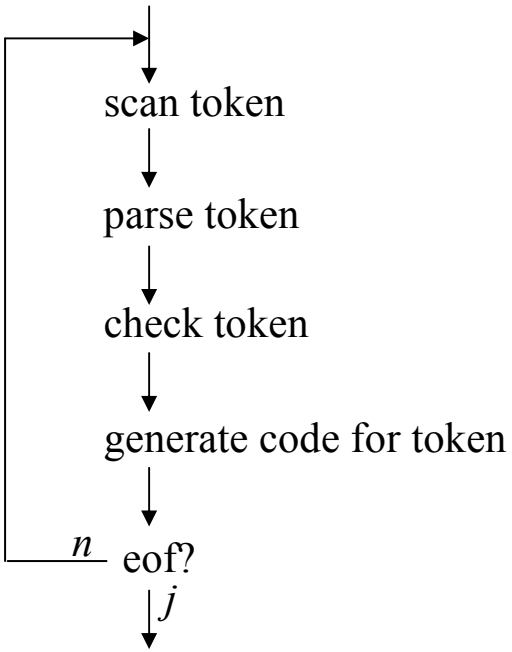


Dynamische Struktur eines Compilers



Einpass-Compiler

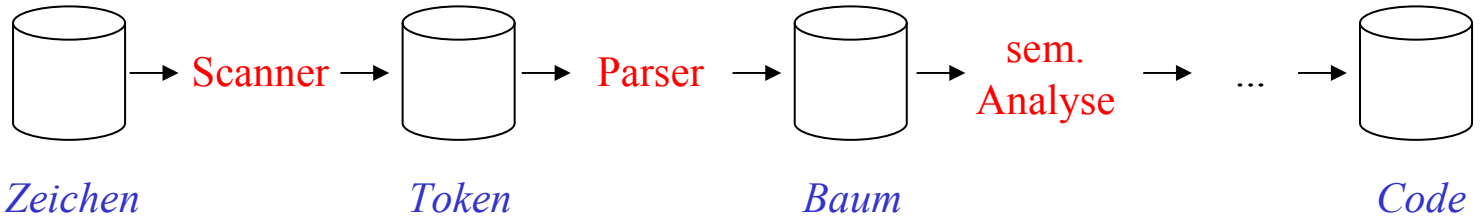
Die einzelnen Phasen arbeiten verzahnt



Während das Quellprogramm gelesen wird, wird bereits das Zielprogramm erzeugt.

Mehrpass-Compiler

Phasen sind eigene Programme, die nacheinander ablaufen

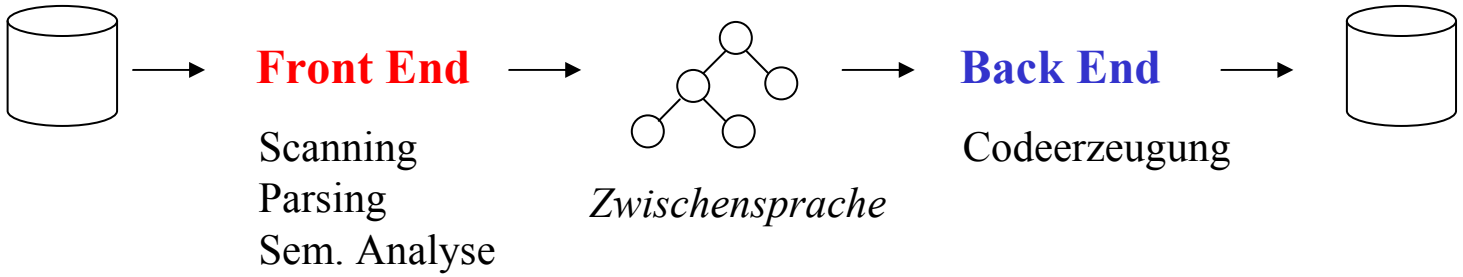


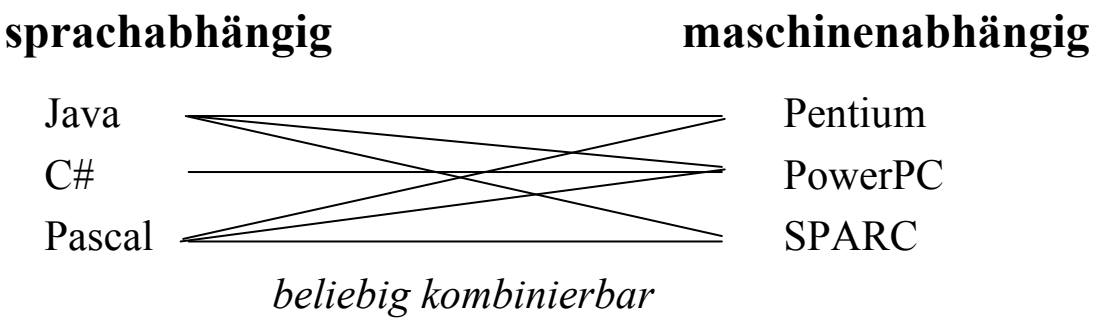
Jede Phase liest von einer Datei und schreibt ihre Ausgabe auf eine neue Datei

Wann ist das notwendig?

- wenn der Hauptspeicher zu klein ist (heute irrelevant)
- wenn die Sprache sehr komplex ist
- wenn einfache Portierbarkeit gewünscht ist

Heute oft Zweipass-Compiler





Vorteile

- bessere Portierbarkeit
- Kombination beliebiger Front Ends mit beliebigen Back Ends möglich
- Zwischensprache ist einfacher optimierbar als Quellsprache

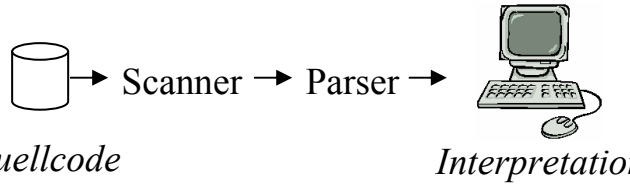
Nachteile

- langsamer
- mehr Speicherverbrauch

Compiler versus Interpreter

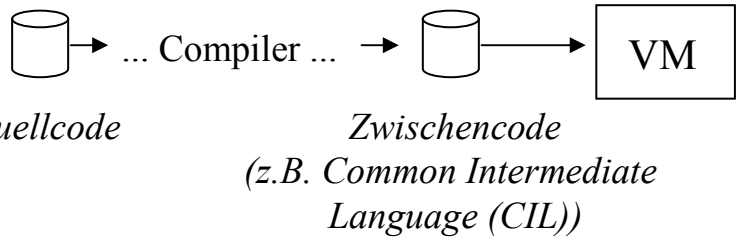
Compiler übersetzt in Maschinencode

Interpreter führt Quellprogramm "direkt" aus



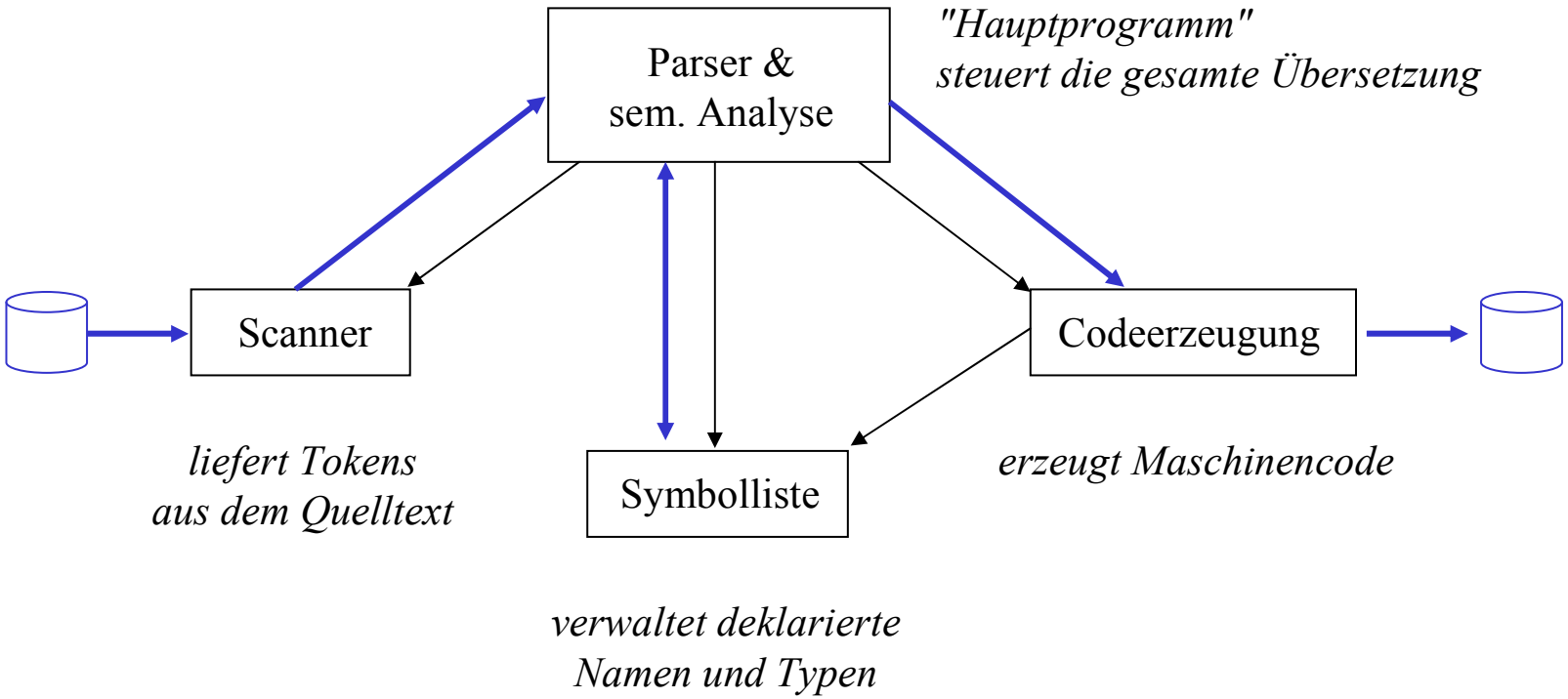
- Anweisungen in einer Schleife laufen jedesmal erneut durch Scanner und Parser

Auch Interpretation von Zwischencode möglich



- Quellcode wird in den Code einer *virtuellen Maschine* (VM) übersetzt
- VM interpretiert den Code; simuliert physische Maschine

Statische Struktur eines Compilers



Aufrufbeziehung

Datenfluss

1. Überblick

Woraus besteht eine Grammatik?

Beispiel

```
Statement = "if" "(" Condition ")" Statement [ "else" Statement ].
```

Vier Bestandteile

Terminalsymbole	werden nicht mehr weiter zerlegt	"if", ">=", ident, number, ...
Nonterminalsymbole	werden weiter zerlegt	Statement, Expr, Type, ...
Produktionen	Ableitungsregeln	Statement = Designator "=" Expr ";". Designator = ident ["." ident]. ...
Startsymbol	oberstes Nonterminal-symbol	CSharp

EBNF-Schreibweise

Extended Backus-Naur Form

John Backus: entwickelte ersten Fortran-Compiler
Peter Naur: gab den Algol60-Report heraus
Niklaus Wirth: entwickelte Pascal

Symbol	*Bedeutung*	*Beispiele*
String	bedeutet sich selbst	"=", "while"
Name	bezeichnet T- oder NT-Symbol	ident, Statement
=	trennt Regelseiten	A = b c d .
.	schließt eine Regel ab	
|	trennt Alternativen	a | b | c ≡ a oder b oder c
(...)	fasst Alternativen zusammen	a (b | c) ≡ ab | ac
[...]	Option	[a] b ≡ ab | b
{...}	Wiederholung	{ a } b ≡ b | ab | aab | aaab | ...

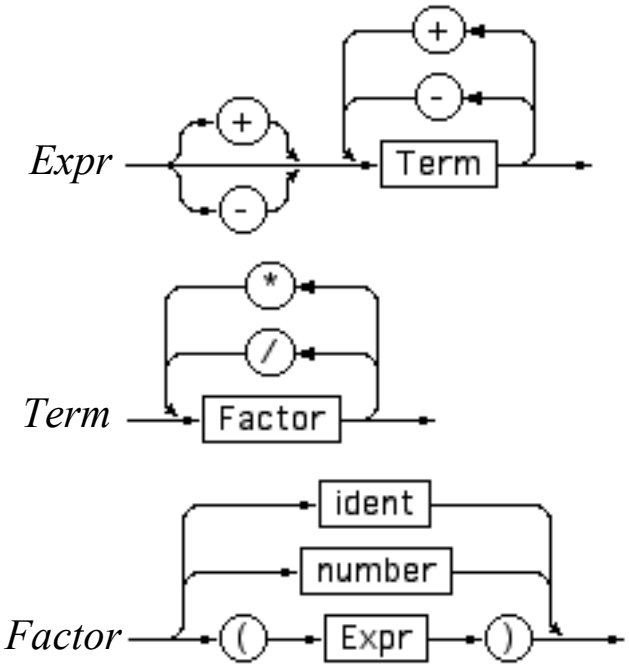
Konvention

- Terminalsymbole beginnen mit Kleinbuchstaben (z.B. ident)
- Nonterminalsymbole beginnen mit Großbuchstaben (z.B. Statement)

Beispiel: Grammatik der arithmetischen Ausdrücke

Produktionen

```
Expr    = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term    = Factor { ( "*" | "/" ) Factor }.
Factor  = ident | number | "(" Expr ")".
```



Terminalsymbole

Einfache TS: "+", "-", "*", "/", "(", ")"
(nur 1 Ausprägung)

Terminalklassen: ident, number
(mehrere Ausprägungen)

Nonterminalsymbole

Expr, Term, Factor

Startsymbol

Expr

Vorrangregeln

Mit Grammatiken lassen sich auch Vorrangregeln von Operatoren ausdrücken

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ( "*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```

Eingabe: - a * 3 + b / 4 - c

⇒ - ident * number + ident / number - ident

⇒ - Factor * Factor + Factor / Factor - Factor

⇒ - Term + Term - Term

⇒ Expr

* und / binden stärker als + und -

- bezieht sich nicht auf *a*, sondern auf *a**3

Wie müsste man die Grammatik umformen, so dass sich - auf *a* bezieht?

Terminale Anfänge von NTS

Mit welchen TS kann ein NTS beginnen?

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ( "*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```

First(Factor) = ident, number, "("

First(Term) = First(Factor)
= ident, number, "("

First(Expr) = "+", "-", First(Term)
= "+", "-", ident, number, "("

Terminale Nachfolger von NTS

Welche TS können auf ein NTS folgen?

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ("*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```

Follow(Expr) = ")", eof

Follow(Term) = "+", "-", Follow(Expr)
= "+", "-", ")", eof

Follow(Factor) = "*", "/", Follow(Term)
= "*", "/", "+", "-", ")", eof

Wo kommt *Expr* auf der rechten Seite einer Produktion vor? Welche TS können dort folgen?

Begriffe der Formalen Sprachen

Alphabet

Die Menge der Terminal- und Nonterminalsymbole einer Grammatik

Kette

Eine endliche Folge von Symbolen aus einem Alphabet

Ketten werden mit griechischen Symbolen bezeichnet (α, β, γ, ...)
z.B: α = ident + number
β = - Term + Factor * number

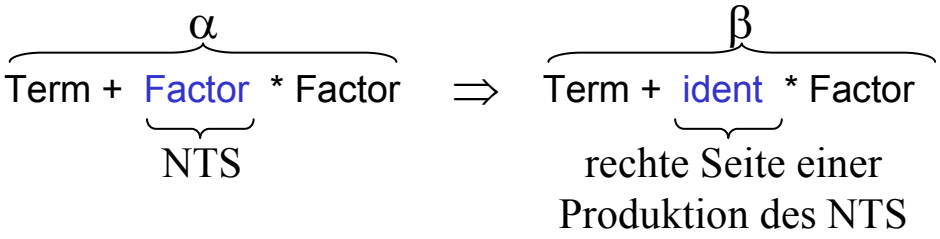
Leere Kette

Die Kette, die kein Symbol enthält.
Wird mit ε bezeichnet.

Ableitungen und Reduktionen

Ableitung

$\alpha \Rightarrow \beta$ (direkte Ableitung)



$\alpha \Rightarrow^* \beta$ (indirekte Ableitung) $\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow ... \Rightarrow \gamma_n \Rightarrow \beta$

$\alpha \Rightarrow^L \beta$ (linkskanonische Ableitung) das linkeste NTS in α wird durch seine rechte Seite ersetzt

$\alpha \Rightarrow^R \beta$ (rechtskanonische Ableitung) das rechteste NTS in α wird durch seine rechte Seite ersetzt

Reduktion

Gegenteil einer Ableitung:
Die rechte Seite einer Produktion in β wird durch das entsprechende NTS ersetzt

Löschbarkeit

Eine Kette α heißt löschbar, wenn sie in die leere Kette abgeleitet werden kann.
$\alpha \Rightarrow^* \varepsilon$

Beispiel

```
A = B C.
B = [ b ].
C = c | d | .
```

B ist löschbar: $B \Rightarrow \varepsilon$

C ist löschbar: $C \Rightarrow \varepsilon$

A ist löschbar: $A \Rightarrow B\ C \Rightarrow C \Rightarrow \varepsilon$

Weitere Begriffe

Phrase

Eine aus einem Nonterminalsymbol ableitbare Kette.

Term-Phrasen sind z.B.: Factor
Factor * Factor
ident * Factor
...

Satzform

Eine aus dem Startsymbol ableitbare Kette.

Z.B.: Expr
Term + Term + Term
Term + Factor * ident + Term
...

Satz

Eine Satzform, die nur aus Terminalsymbolen besteht.

Z.B.: ident * number + ident

Sprache (Formale Sprache)

Die Menge aller Sätze einer Grammatik (meist unendlich groß).
Z.B.: Die Sprache C# ist die Menge aller gültigen C#-Programme

Rekursion

Eine Produktion ist rekursiv, wenn $A \Rightarrow^* \omega_1 A \omega_2$

Verwendet zur Darstellung von Wiederholungen und Schachtelungen

Direkte Rekursion $A \Rightarrow \omega_1 A \omega_2$

Linksrekursion	A = b | A a.	A ⇒ A a ⇒ A a a ⇒ A a a a ⇒ b a a a a a ...
Rechtsrekursion	A = b | a A.	A ⇒ a A ⇒ a a A ⇒ a a a A ⇒ ... a a a a a b
Zentralrekursion	A = b | "(" A ")".	A ⇒ (A) ⇒ ((A)) ⇒ (((A))) ⇒ (((... (b)...)))

Indirekte Rekursion $A \Rightarrow^* \omega_1 A \omega_2$

Beispiel

```
Expr   = Term { "+" Term }.
Term   = Factor { "*" Factor }.
Factor = id | "(" Expr ")".
```

Expr ⇒ Term ⇒ Factor ⇒ "(" Expr ")"

Beseitigung von Linksrekursion

Linksrekursion stört bei der Topdown-Syntaxanalyse

```
A = b | A a.
```

Beide Alternativen fangen mit *b* an.
Der Parser kann sich nicht entscheiden, welche er wählen soll.

Umwandlung von Rekursion in Iteration

```
E = T | E "+" T.
```

Überlegen, welche Phrasen erzeugt werden können

```
T
T + T
T + T + T
...
```

Daraus sieht man, wie die iterative EBNF-Regel auszusehen hat.

```
E = T { "+" T }.
```

1. Überblick

Einfache BNF-Schreibweise

Terminalsymbole	ohne Hochkommas (ident, +, -)
Nonterminalsymbole	in spitzen Klammern (<Expr>, <Term>)
Regelseiten	durch ::= getrennt

BNF-Grammatik der arithmentischen Ausdrücke

```
<Expr>    ::=  <Sign> <Term>
<Expr>    ::=  <Expr> <Addop> <Term>

<Sign>    ::=  +
<Sign>    ::=  -
<Sign>    ::=

<Addop>   ::=  +
<Addop>   ::=  -

<Term>    ::=  <Factor>
<Term>    ::=  <Term> <Mulop> <Factor>

<Mulop>   ::=  *
<Mulop>   ::=  /

<Factor>  ::=  ident
<Factor>  ::=  number
<Factor>  ::=  ( <Expr> )
```

- Alternativen werden zu mehreren Regeln
- Wiederholung muss durch Rekursion ausgedrückt werden

Vorteile

- weniger Metasymbole (kein |, (), [], {})
- Syntaxbaum lässt sich einfacher konstruieren

Nachteile

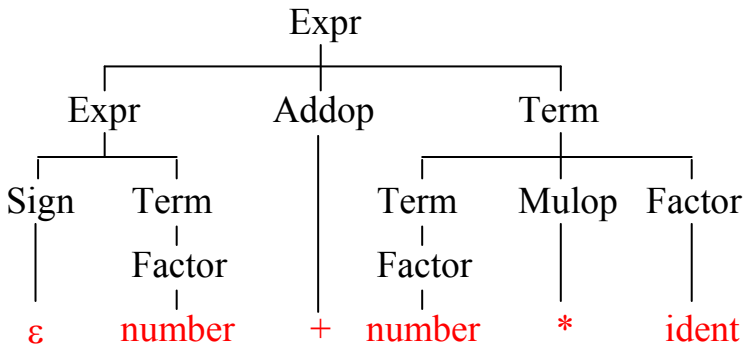
- länger
- schwerer lesbar

Syntaxbaum

Zeigt die Ableitungsstruktur für einen Satz einer Grammatik

z.B. für 10 + 3 * i

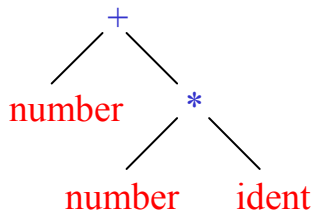
Konkreter Syntaxbaum (Parse Tree)



wäre mit EBNF nicht so einfach
wegen [...] und {...}, z.B.:
Expr = [Sign] Term { Addop Term }.

Vorrangregeln beachtet:
NTS weiter unten im Baum haben
Vorrang vor NTS weiter oben im Baum.

Abstrakter Syntaxbaum (Blätter = Operanden, innere Knoten = Operatoren)



oft als interne Programmdarstellung
für Optimierungen verwendet

Mehrdeutigkeit

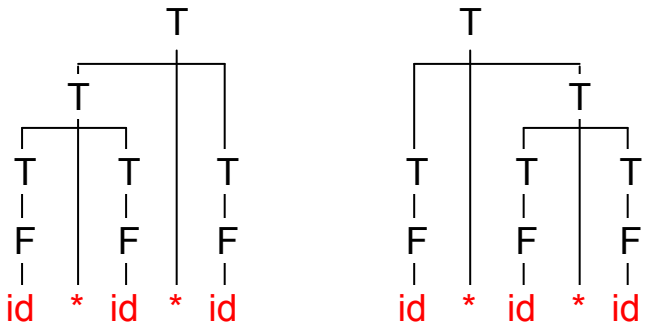
Eine Grammatik ist mehrdeutig, wenn man für einen Satz mehrere Syntaxbäume angeben kann.

Beispiel

```
T = F | T "*" T.
F = id.
```

Beispielsatz: id * id * id

Über diesen Satz können zwei verschiedene Syntaxbäume gebaut werden



Mehrdeutige Grammatiken sind zur Syntaxanalyse ungeeignet!

Beseitigung von Mehrdeutigkeit

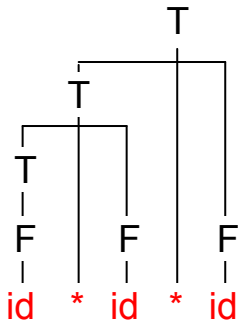
Im Beispiel

```
T = F | T "*" T.
F = id.
```

ist nicht die Sprache mehrdeutig, sondern nur die Grammatik.

Die Grammatik kann umgeformt werden zu

```
T = F | T "*" F.
F = id.
```



d.h. T hat Priorität vor F

nur dieser Syntaxbaum ist möglich

Noch besser: Umformung in EBNF

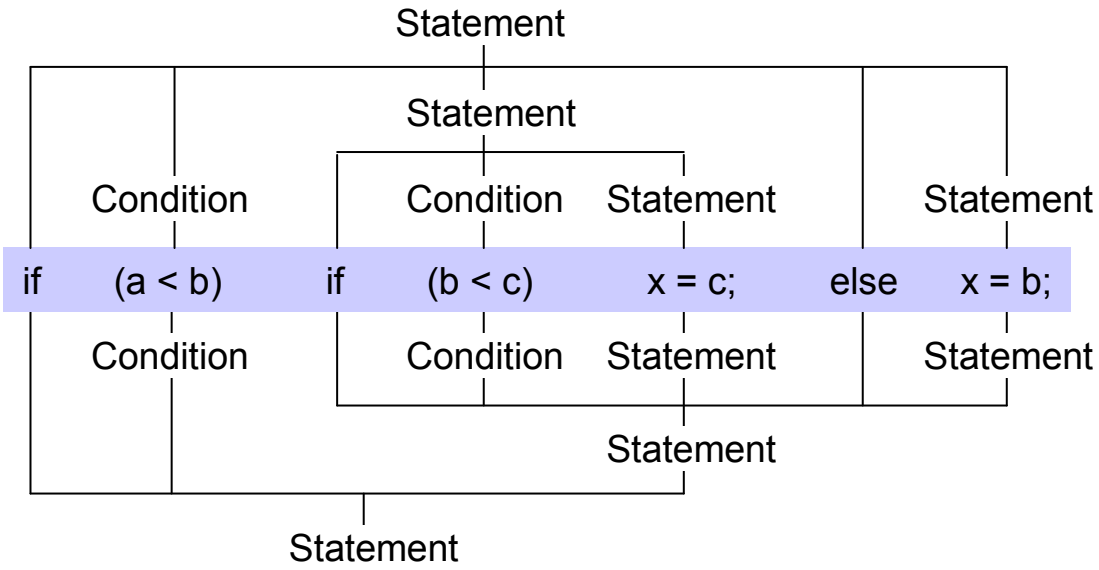
```
T = F { "*" F }.
F = id.
```

Inhärente Mehrdeutigkeit

Es gibt Sprachen, die inhärent mehrdeutig sind

Beispiel: Dangling Else

```
Statement =  Assignment
           |  "if" Condition Statement
           |  "if" Condition Statement "else" Statement
           |  ... .
```



Es lässt sich keine eindeutige Grammatik finden!

Ausweg in C#
Man erkennt die längst-mögliche rechte Seite der Statement-Produktion
⇒ führt zum unteren Syntaxbaum

1. Überblick

Grammatikklassen

Hierarchie von Noam Chomsky (1956)

Grammatiken sind prinzipiell Mengen von Produktionen der Form $\alpha = \beta$.

Klasse 0 **Unbeschränkte Grammatiken** (α und β beliebig)

Beispiel: A = a A b | B c B.
aBc = d.
dB = bb.

$A \Rightarrow aAb \Rightarrow aBcBb \Rightarrow dBb \Rightarrow bbb$

Erkennbar durch Turingmaschinen

Klasse 1 **Kontextsensitive Grammatiken** ($|\alpha| \leq |\beta|$)

Beispiel: a A = a b c.

Erkennbar durch linear beschränkte Automaten

Klasse 2 **Kontextfreie Grammatiken** ($\alpha = NT, \beta \neq \varepsilon$)

Beispiel: A = a b c.

Erkennbar durch Kellerautomaten

Klasse 3 **Reguläre Grammatiken** ($\alpha = NT, \beta = T \mid T\ NT$)

Beispiel: A = b | b B.

Erkennbar durch endliche Automaten

Im Übersetzerbau sind nur diese beiden Grammatikklassen relevant

1. Überblick

Beispiel-Programm


```
{this funtion prints "Hello"}
function int hello
  call put_c <- 'H' end
  call put_c <- 'e' end
  call put_c <- 'l' end
  call put_c <- 'l' end
  call put_c <- 'o' end
end

{absolute value of x}
function int abs <- int x
  if x < 0 abs = 0-x
  else abs = x end
end

{entry point {to the program}}
function int main
  call hello
  int nr
  nr = call abs <- call get_i end
  call put_i <- nr end
end
```

Programm: Sammlung von Funktionen in einer Datei.

Typen:
int … Ganzzahlen
char … Zeichen

Variablen: lokal, an beliebigen Stellen zu deklarieren.

Funktionen: beliebige Eingabeparameter, genau ein Rückgabewert (Variable mit Namen der Funktion).

Einsprungspunkt: Funktion mit dem Namen "main".

Kommentare: Blockkommentare, können geschachtelt auftreten.

Arithmetische Operatoren: +, -

Vergleichsoperatoren: <, >, =, # (ungleich)

Vordefinierte Funktionen: put_c, put_i, get_c, get_i

Lexikalische Struktur von Micro

Namen	ident = idChar { idChar | digit }.	idChar = "a..z" und "A..Z" und "_" digit = "0..9"
Zahlen	number = digit { digit }.	alle Zahlen sind vom Typ *int*
Zeichenkonstanten	charConst = '\'' char '\''.	alle Zeichenkonstanten sind vom Typ *char* (dürfen jeden char auch zB: \n enthalten)
keine Strings		
Schlüsselwörter	if else end call function	
Operatoren	+ -	
	= # > <	
	, <-	
Kommentare	{ ... }	dürfen auch geschachtelt auftreten
Typen	int char	

Syntaktische Struktur von Micro


```
Program = { Function }.

Function =
"function" ident ident [ "<-" FormPar { "," FormPar } ] Block "end".

FormPar = ident ident.

Block = { Statement }.

Statement =
  ( ident ( ident | "=" Expr | "<-" ( numberLit | charLit ) )
  | "if" Cond Block [ "else" Block ] "end"
  | Call
  )
.

Cond = Expr ( "<" | "=" | "#" | ">" ) Expr.

Expr = Term { ( "+" | "-" ) Term }.

Term = ( charLit | numberLit | ident | Call ).

Call = "call" ident [ "<-" Expr { "," Expr } "end" ].
```

2. Lexikalische Analyse

Aufgaben der Lexikalischen Analyse

1. **Liefert Terminalsymbole** (Tokens)

Zeichenfolge

Tokenfolge
(muss mit EOF enden)

2. **Überliest bedeutungslose Zeichen**
   - Leerzeichen
   - Tabulatoren
   - Zeilenenden (CR, LF)
   - Kommentare

Token haben eine syntaktische Struktur, z.B.

```
ident =     letter { letter | digit }.
number =    digit { digit }.
if =        "i" "f".
eq =        "=" "=".
...
```

Warum ist die Erkennung der Token nicht Teil der Syntaxanalyse?

Warum nicht Teil der Syntaxanalyse?

Würde die Syntaxanalyse verkomplizieren
(z.B. schwierige Unterscheidung zwischen Schlüsselwörtern und Namen)

```
Statement =  ident "=" Expr ";"
           |  "if" "(" Expr ")" ... .
```

müsste geschrieben werden als

```
Statement =  "i" (   "f" "(" Expr ")" ...
                   |   not_f { letter | digit } "=" Expr ";"
                   )
           |  not_i { letter | digit } "=" Expr ";".
```

Scanner muss auch Leerzeichen, Tabs, Zeilenenden, Kommentare entfernen
(können überall vorkommen => würde komplizierte Grammatik ergeben)

```
Statement =  "if" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} ... .
Blank = " " | "\r" | "\n" | "\t" | Comment.
```

Für Token reichen reguläre Grammatiken
(einfacher und schneller analysierbar als kontextfreie Grammatiken)

2. Lexikalische Analyse

Reguläre Grammatiken

Definition

Eine Grammatik heißt regulär, wenn sie sich durch Regeln der folgenden Art ausdrücken lässt:

```
A = a.
A = b B.
```

a, b ∈ TS
A, B ∈ NTS

Beispiel Grammatik für Namen

```
Ident  =  letter
       |  letter Rest.
Rest   =  letter
       |  digit
       |  letter Rest
       |  digit Rest.
```

z.B. Ableitung des Namens xy3

Ident ⇒ letter Rest ⇒ letter letter Rest ⇒ letter letter digit

Andere Definition

Eine Grammatik heißt regulär, wenn sie sich durch eine einzige EBNF-Regel ohne Rekursion ausdrücken lässt.

Beispiel Grammatik für Namen

```
Ident  =  letter { letter | digit }.
```

Beispiele

Lässt sich folgende Grammatik in eine reguläre Grammatik umformen?

```
E = T { "+" T }.
T = F { "*" F }.
F = id.
```

Lässt sich folgende Grammatik in eine reguläre Grammatik umformen?

```
E = F { "*" F }.
F = id | "(" E ")".
```

Beschränkungen regulärer Grammatiken

Reguläre Grammatiken können *keine Klammerstrukturen* ausdrücken.
D.h. keine Zentralrekursion möglich!

Zentralrekursion wird aber in Programmiersprache oft gebraucht.

- geschachtelte Ausdrücke — Expr ⇒ ... "(" Expr ")" ...
- geschachtelte Anweisungen — Statement ⇒ "do" Statement "while" "(" Expr ")"
- innere Klassen — Class ⇒ "class" "{" ... Class ... "}"

Daher benötigt man für die Syntaxanalyse solcher Sprachen kontextfreie Grammatiken.

Lexikalische Strukturen sind meist regulär

Namen	letter { letter | digit }
Zahlen	digit { digit }
Zeichenketten	"\"" { noQuote } "\""
Schlüsselwörter	letter { letter }
Sonderzeichen	">" "="

Ausnahme: geschachtelte Kommentare

/* ..... /* ... */ ..... */

Müssen im Scanner sonderbehandelt werden

Reguläre Ausdrücke

Andere Schreibweise für reguläre Grammatiken

Definition

1. ε (leere Kette) ist ein regulärer Ausdruck
2. Ein Terminalsymbol ist ein regulärer Ausdruck
3. Wenn α und β reguläre Ausdrücke sind, sind auch folgende Ausdrücke regulär

α β	
(α | β)	
(α)?	ε | α
(α)*	ε | α | αα | ααα | ...
(α)+	α | αα | ααα | ...

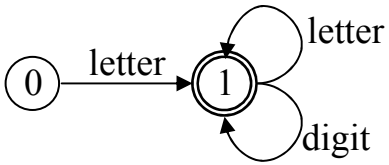
Beispiele

"w" "h" "i" "l" "e"	while
letter (letter | digit)*	Namen
digit+	Zahlen

Endlicher Automat

Automat zur Erkennung einer regulären Sprache
(engl. DFA = deterministic finite automaton)

Beispiel




Endzustand

Startzustand per Konvention immer 0

Zustandsübergangsfunktion als Tabelle

δ	letter	digit
z0	z1	error
z1	z1	z1

"endlich", weil δ explizit angeschrieben werden kann

Definition

Ein deterministischer endlicher Automat ist ein 5-Tupel (Z, S, δ, z0, F)

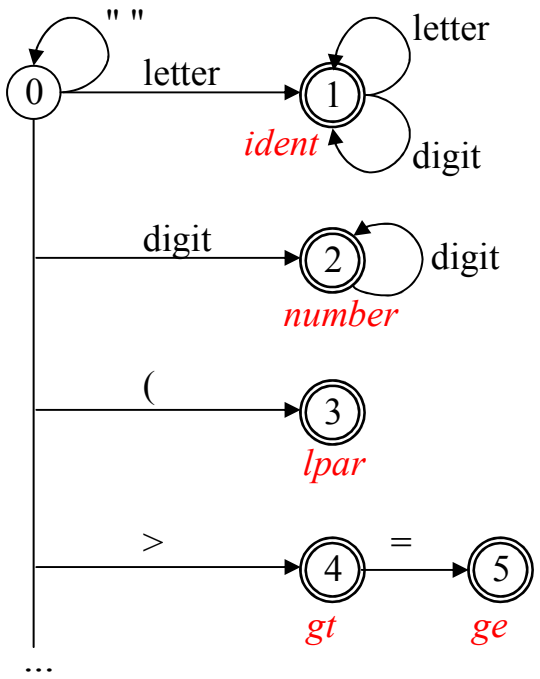
- Z — Menge von Zuständen
- S — Menge von Eingabesymbolen
- $\delta: Z \times S \rightarrow Z$ — Zustandsübergangsfunktion
- z0 — Anfangszustand
- F — Menge von Endzuständen

Die durch einen DFA erkannte **Sprache** ist die Menge aller Symbolfolgen, die vom Startzustand in einen Endzustand führen

Ein DFA hat einen Satz erkannt
- wenn sich der DFA in einem Endzustand befindet
- und wenn die Eingabe zu Ende ist oder kein Übergang mit dem nächsten Symbol möglich ist

Scanner als DFA

Man kann sich den Scanner als großen DFA vorstellen



Beispiel

Eingabe: max >= 30

z0 —m a x→ z1

- kein Übergang mehr mit " " in z1
- *ident* erkannt

z0 —> =→ z5

- überliest Blanks am Anfang
- bleibt nicht in z4 stehen
- kein Übergang mehr mit " " in z5
- *ge* erkannt

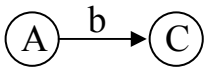
z0 —3 0→ z2

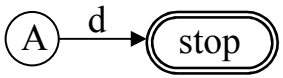
- überliest Blanks am Anfang
- kein Übergang mehr mit " " in z2
- *number* erkannt

Scanner beginnt nach jedem erkannten Symbol wieder in z0.

Umwandlung reg. Grammatik ↔ DFA

Kann nach folgendem Schema erfolgen

A = b C. ⇔ 

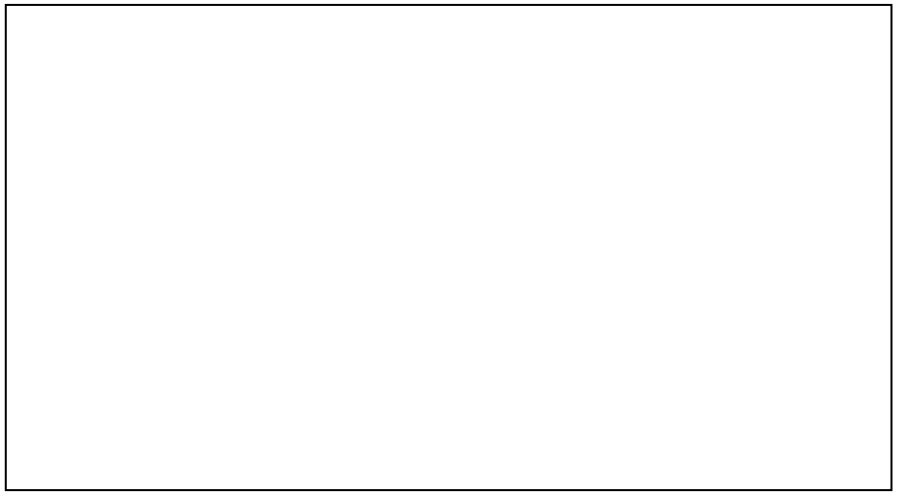
A = d. ⇔ 

Beispiel

Grammatik

A = a B | b C | c.
B = b B | c.
C = a C | c.

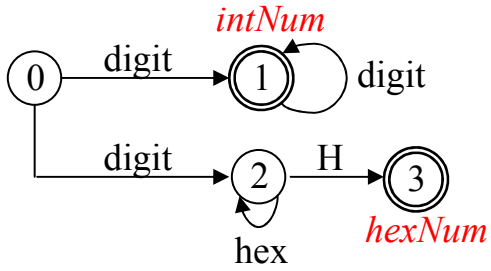
Automat



Nichtdeterministischer DFA (NDFA)

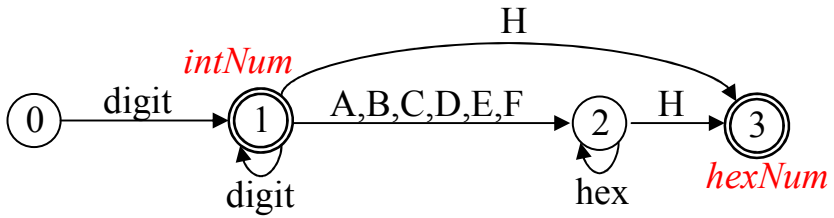
Beispiel

```
intNum =   digit { digit }.
hexNum = digit { hex } "H".
digit =      "0" | "1" | ... | "9".
hex =        digit | "A" | ... | "F".
```



nicht deterministisch,
weil 2 digit-Übergänge
in z0 möglich sind

Jeder NDFA kann in einen äquivalenten DFA umgewandelt werden
(Algorithmus siehe z.B. Aho, Sethi, Ullman: Compilerbau)



Implementierung eines DFA (Variante 1)

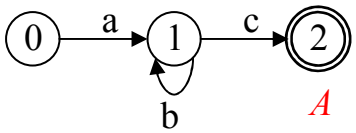
Speicherung von δ als Matrix

```
int[,] delta = new int[maxStates, maxSymbols];
int lastState, state = 0;  // DFA starts in state 0
do {
   int sym = next symbol;
   lastState = state;
   state = delta[state][sym];
} while (state != undefined);
assert(lastState ∈ F);  // F is set of final states
return recognizedToken[lastState];
```

Das ist ein Beispiel für einen universellen *tabellengesteuerten Algorithmus*

Beispiel für δ

A = a { b } c.

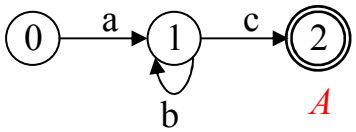


δ	a	b	c	
0	1	-	-	
1	-	1	2	
2	-	-	-	E

```
int[][] delta = {{1,-1,-1}, {-1,1,2}, {-1,-1,-1}};
```

Diese Implementierung ist allerdings nicht besonders effizient.

Implementierung eines DFA (Variante 2)



Auscodieren der Zustände

```
char ch = read();
s0:  if (ch == 'a') { ch = read(); goto s1; }
     else goto err;
s1:  if (ch == 'b') { ch = read(); goto s1; }
     else if (ch == 'c') { ch = read(); goto s2; }
     else goto err;
s2:  return A;
err: return errorToken;
```

In Java ist das etwas umständlicher:

```
int state = 0;
loop:
   for (;;) {
      char ch = read();
      switch (state) {
         case 0:  if (ch == 'a') { state = 1; break; }
                  else break loop;
         case 1:  if (ch == 'b') { state = 1; break; }
                  else if (ch == 'c') { state = 2; break; }
                  else break loop;
         case 2: return A;
      }
   }
return errorToken;
```

2. Lexikalische Analyse

Schnittstelle des Scanners


```
class Scanner {
   Token Next () {...}
}
```

Lesen des Tokenstroms

```
Token t;
for (;;) {
   t = Scanner.Next();
   ...
}
```

Tokens


```
class Token {
   TokenKind kind;     // token code
   int line;           // token line (for error messages)
   int col;            // token column (for error messages)
   int val;            // token value (for number and charCon)
   string str;         // token string (for numbers and identifiers)
}
```

Token-Codes für Micro

Fehlertoken	Tokenklassen	Operatoren und Sonderzeichen	Schlüsselwörter	End of file
none	Ident /* max, x, y */ numberLit /* 1, 25, 3 */ charLit /* 'a', 'x' */	plus /* + */ minus /* - */ eql /* = */ neq /* # */ lss /* < */ gtr /* > */ comma /* , */ methPar /* <- */	ifKW /* if */ elseKW /* else */ endKW /* end */ callKW /* call */ functionKW /* function */	eof

Scanner-Implementierung

Zustand des Scanners

```
private Stream source;        // Eingabestrom
private int line;             // Zeile des Zeichens ch
private int col;              // Spalte des Zeichens ch
private char ch;              // nächstes noch unverarbeitetes Zeichen
private IError error;         // Fehlerlog
```

Konstruktor()

```
public Scanner(Stream source, IError err) {
  error = err;  this.source = source;  line = 1;  col = 0; NextCh();
}
```

Methode NextCh()

```
private void NextCh() {
  try {
    ch = (char) source.ReadByte();
    switch (ch) {
      case cCr: ch = (char) source.ReadByte(); // skip CR
        // "FALL-THROUGH" end of line can either be CRLF or just LF
        goto case cLf;
      case cLf: line++; col = 0; break;
      default: col++;  break;
    }
  } catch (IOException) { ch = cEof; }
}
```

- *ch* = nächstes Eingabezeichen
- liefert bei Dateiende *EOF*
- führt *line* und *col* mit

Methode Next()


```
public Token Next () {
   while (ch <= ' ') NextCh();  // skip blanks, tabs, eols
   Token t = new Token(TokenKind.none, line, col);
   switch (ch) {
      case 'a': ... case 'z': case 'A': ... case 'Z': ReadName(t); break;
      case '0': case '1': ... case '9': ReadNumber(t); break;
      case '+': NextCh(); t.Kind = TokenKind.plus; break;
      case EOF: t.Kind = TokenKind.eof; break; // no NextCh() any more
      ...
      case '<':     NextCh();
                    if (ch == '-') { NextCh(); t.Kind = TokenKind.methPar; }
                    else t.Kind = TokenKind.lss;
                    break;
      ...

      case '{':     SkipComment(t); t = Next();
                    break;

      default:      NextCh();
                    error.Error(t, "Unexpected input character: " + ch …);
                    break;
   }
   return t;
} // ch holds the next character that is still unprocessed
```

- Namen, Schlüsselwörter
- Zahlen
- einfache Tokens
- zusammengesetzte Tokens
- Kommentare
- fehlerhaftes Zeichen

Weitere Methoden

private void ReadName (Token t)

- *ch* enthält zu Beginn den ersten Buchstaben des Names
- *ReadName* liest weitere Buchstaben, Ziffern und '_' und speichert sie in *t.Str*
- sucht den Namen in einer Schlüsselworttabelle (Hashing oder binäres Suchen)
  wenn gefunden: t.Kind = *Schlüsselwortcode*;
  sonst: t.Kind = TokenKind.ident;
- *ch* enthält am Ende das erste Zeichen nach dem Namen

private void ReadNumber (Token t)

- *ch* enthält zu Beginn die erste Ziffer der Zahl
- *ReadNumber* liest weitere Ziffern und speichert sie in *t.Str*;
  konvertiert am Ende die Ziffernkette in eine Zahl und speichert diese in *t.Val.*
  wenn Überlauf: Fehler melden
- t.Kind = TokenKind.numberLit;
- *ch* enthält am Ende das erste Zeichen nach der Zahl

Effizienzüberlegungen

Typische Programmgröße

ca. 1000 Anweisungen
⇒ ca. 6000 Symbole
⇒ ca. 60000 Zeichen

Lexikalische Analyse ist die zeitaufwendigste Phase in einem Compiler
(ca. 20-30% der Übersetzungszeit)

Jedes Zeichen so selten wie möglich umspeichern

deshalb ist *ch* global und kein Parameter von *NextCh()*

Eventuell gepuffert lesen

```
Stream file = new FileStream("MyProg.mic");
Stream buf = new BufferedStream(file);
```

ist aber nur bei großen Eingabedateien spürbar

3. Syntaxanalyse

Kontextfreie Grammatiken

Problem

Reguläre Grammatiken können keine Zentralrekursion ausdrücken

```
E = x | "(" E ")".
```

Dafür braucht man zumindest kontextfreie Grammatiken

Definition

Eine Grammatik heißt ***kontextfrei*** (KFG), wenn alle Produktionen folgende Form haben:

A = α.

A ∈ NTS, α nichtleere Folge von TS und NTS
In EBNF kann α auch |, (), [] und {} enthalten

Beispiel

```
Expr = Term { ("+" | "-") Term }.
Term = Factor { ("*" | "/") Factor }.
Factor = id | "(" Expr ")".
```

indirekte Zentralrekursion

Kontextfreie Sprachen werden durch ***Kellerautomaten*** erkannt

Kellerautomat

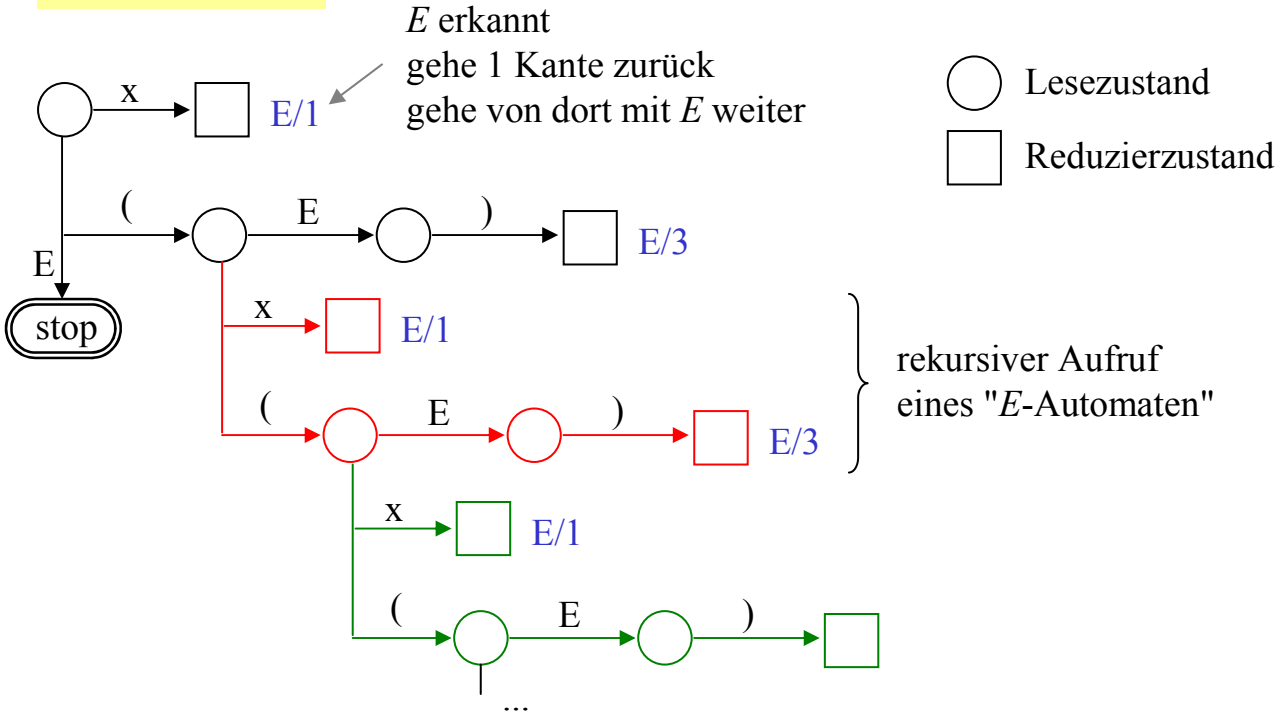
englisch: Push Down Automaton (PDA)

Eigenschaften

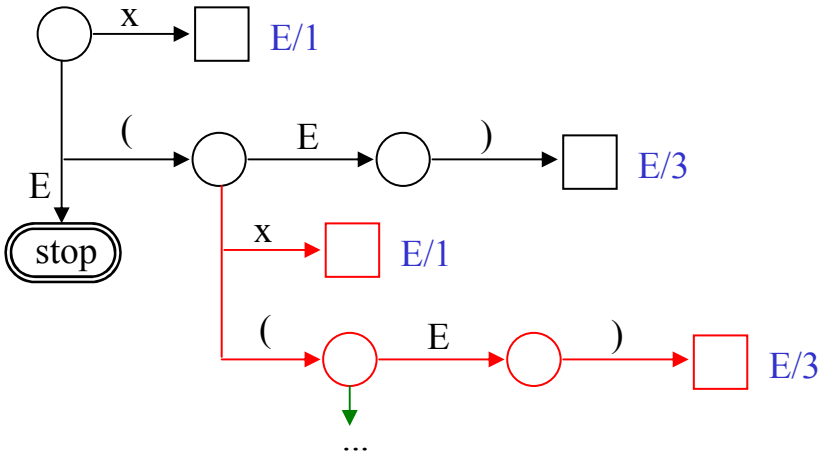
- erlaubt Zustandsübergänge mit Terminal- und Nonterminalsymbolen
- merkt sich Zustandsübergänge in einem Keller

Beispiel

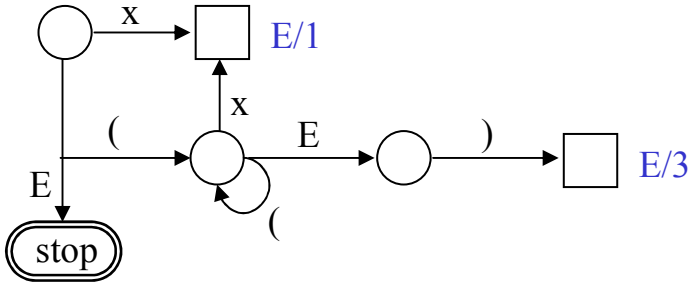
E = x | "(" E ")".



Kellerautomat (Fortsetzung)



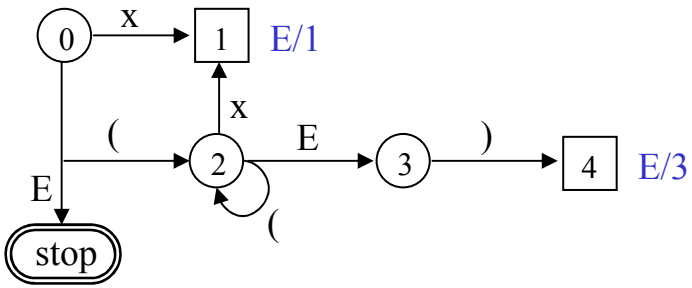
Kann vereinfacht werden zu



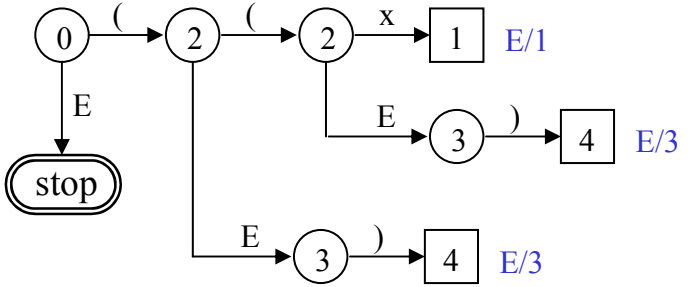
Erfordert Keller, um den Rückweg durch alle bisher durchlaufenen Zustände zu finden

Arbeitsweise eines PDA

Beispiel: Erkennung von ((x))



Durchlaufene Zustände werden in einem Keller gespeichert



Einschränkung kontextfreier Grammatiken

KFGs können keine *Kontextbedingungen* ausdrücken

Zum Beispiel:

- ***Jeder verwendete Name muss deklariert worden sein***
  Deklaration gehört zum Kontext der Verwendung
  ```
  x = 3;
  ```
  kann je nach Kontext richtig oder falsch sein

- ***In Ausdrücken müssen die Typen der Operanden übereinstimmen***
  Typen werden in der Deklaration festgelegt ⇒ gehört zum Kontext der Verwendung

Lösungsmöglichkeiten

- ***Kontextsensitive Grammatiken***
  zu kompliziert

- ***Überprüfung der Kontextbedingungen in der semantischen Analyse***
  d.h. die Grammatik erlaubt Konstrukte, die laut Kontextbedingungen verboten sind, z.B.:
  ```
  int x = "drei";
  ```
  syntaktisch korrekt
  semantisch falsch
  Fehler wird erst bei semantischer Analyse gemeldet.

Kontextbedingungen

Für jede Syntaxregel werden die semantischen Zusatzbedingungen angegeben

Zum Beispiel für Micro

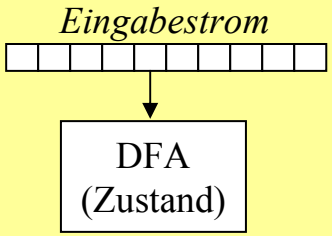
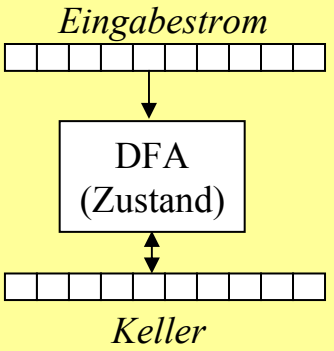
```
Statement = ident "=" Expr ";".
```

- *ident* muss eine Variable oder ein Funktionsargument sein.
- Der Typ von *Expr* muss mit dem Typ von *ident* zuweisungskompatibel sein.

```
Call = "call" ident.
```

- *ident* muss eine Funktion bezeichnen.
- Die Funktion darf keine Parameter erwarten.

Reguläre versus kontextfreie Grammatiken

	Reguläre Grammatiken	**Kontextfreie Grammatiken**
Anwendung	Lexikalische Analyse	Syntaxanalyse
Erkennung	**DFA** (kein Keller)  	**PDA** (Keller) 
Produktionen	A = a | b C.	A = α.
Probleme	Klammerkonstrukte	Kontextsensitive Konstrukte (z.B. Typprüfungen, ...)

3. Syntaxanalyse

Syntaxanalyse mit Rekursivem Abstieg

englisch: *Recursive Descent Parsing*

- Topdown-Technik
- Syntaxbaum wird von oben nach unten aufgebaut

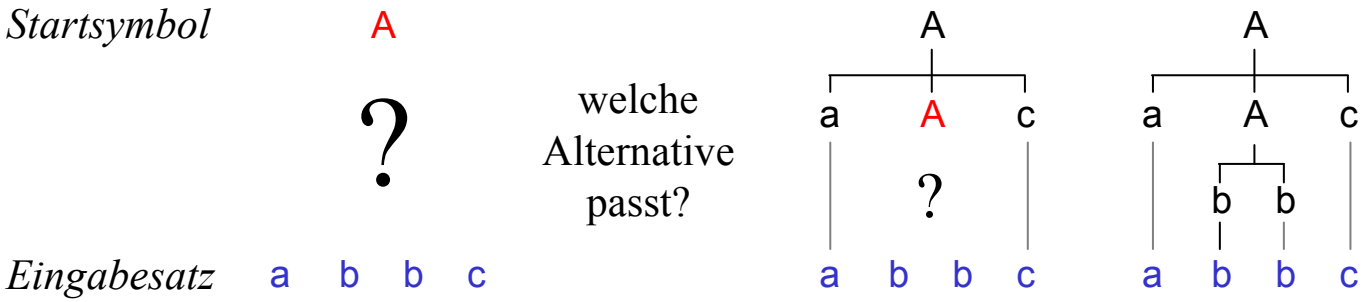
Beispiel

Grammatik

A = a A c | b b.

Eingabesatz

a b b c



Auswahl der passenden Alternative auf Grund von

- Vorgriffssymbol aus dem Eingabestrom
- Terminale Anfänge der einzelnen Alternativen

Statische Variablen des Parsers

Vorgriffssymbol

Parser schaut immer um 1 Symbol voraus (look ahead)

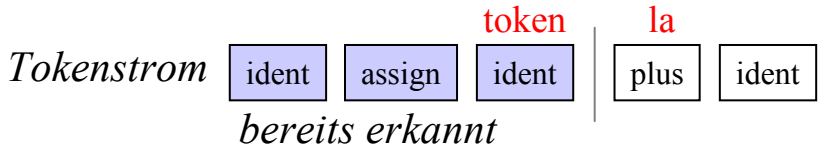
```
private Token la;          // Lookahead-Token (noch nicht erkannt)
```

Parser merkt sich das letzte erkannte Symbol (für Semantikverarbeitung)

```
private Token token;       // zuletzt erkannter Token
```

Diese Variablen werden von der Hilfsmethode *Scan()* gefüllt

```
static void Scan () {
   token = la;
   la = scanner.Next();
}
```



Scan() wird zu Beginn der Syntaxanalyse aufgerufen ⇒ erstes Symbol steht in *la*

Erkennung von Terminalsymbolen

Muster

Eingabesymbol: a
Parser-Aktion: Check(a);

Dazu nötige Hilfsmethoden

```
private void Check (TokenKind expected) {
   if (la.Kind == expected) Scan();  // erkannt, daher weiterlesen
   else Error("unexpected token, expected: " + expected + ", found: " + fLa.Kind);
}
```

```
private void Error (string msg) {
   error.Error(la, msg);
}
```

```
public enum TokenKind
{
   none, ident, numberLit, charLit, plus, minus, eql, neq,
   lss, gtr, comma, ifKW, elseKW, endKW, callKW, functionKW, methPar, eof
};
```

Erkennung von Nonterminalsymbolen

Muster

Nonterminalsymbol: A
Parser-Aktion: A(); // Aufruf der Erkennungsmethode von A

Jedes Nonterminalsymbol wird durch eine gleichnamige Methode erkannt

```
private void A () {
   ... Aktionen zur Erkennung von A ...
}
```

Initialisierung des Micro-Parsers

```
public void Parse () {
   Scan();                    // füllt token und la
   Micro();                   // ruft die Erkennungsmethode des Startsymbols auf
   Check(TokenKind.eof); // nach dem Programm muss die Eingabedatei zu Ende sein
}
```

Erkennung von Sequenzen

Muster

Grammatikregel: A = a B c.

Erkennungsmethode:

```
private void A () {
   // la enthält das erste Symbol von A
   Check(a);
   B();
   Check(c);
   // la enthält das Nachfolgesymbol von A
}
```

Simulation

A = a B c.
B = b b.

```
private void A () {          ---------- a b b c
   Check(a);                 ---------- b b c
   B();                      ---------- c
   Check(c);                 ----------
}

private void B () {          ---------- b b c
   Check(b);                 ---------- b c
   Check(b);                 ---------- c
}
```

restliche Eingabe

Erkennung von Alternativen

Grammatikmuster: $\alpha \mid \beta \mid \gamma$ α, β, γ sind beliebige EBNF-Ausdrücke

Erkennungsaktion:

```
if (la ∈ First(α)) { ... erkenne α ... }
else if (la ∈ First(β)) { ... erkenne β ... }
else if (la ∈ First(γ)) { ... erkenne γ ... }
else Error("...");  // sprechende Fehlermeldung
```

Beispiel

```
A = a B | B b.
B = c | d.
```

First(aB) = {a}
First(Bb) = First(B) = {c, d}

```
private void A () {
   if (la == a) {
      Check(a);
      B();
   } else if (la == c || la == d) {
      B();
      Check(b);
   } else Error("invalid start of A");
}
```

Erkennung von EBNF-Optionen

Grammatikmuster: [α] α ist ein beliebiger EBNF-Ausdruck

Erkennungsaktion: if (la ∈ First(α)) { ... *erkenne* α ... } // kein Fehlerzweig!

Beispiel

```
A = [ a b ] c.
```

```
private void A () {
    if (la == a) {
        Check(a);
        Check(b);
    }
    Check(c);
}
```

Erkennung der Phrasen a b c und c

Erkennung von EBNF-Iterationen

Grammatikmuster: { α } α ist ein beliebiger EBNF-Ausdruck

Erkennungsaktion: while (la ∈ First(α)) { ... *erkenne* α ... }

Beispiel

```
A = a { B } b.
B = c | d.
```

```
private void A () {
   Check(a);
   while (la == c || la == d) B();
   Check(b);
}
```

Erkennung der Phrasen a c d c b und a b

Alternativ dazu

```
private void A () {
   Check(a);
   while (la != b && la != Token.EOF) B();
   Check(b);
}
```

Ohne EOF: Gefahr einer Endlosschleife, wenn b vergessen wird.

Umgang mit großen First-Mengen

Falls Menge > 4 Elemente: Sammlung verwenden (Übersicht, Geschwindigkeit)

Beispiel: First(A) = {a, b, c, d, e}
First(B) = {f, g, h, i, j}

Mengenabfrage

```
C = A | B.
```

```
private void C () {
    if (firstA[la]) A();
    else if (firstB[la]) B();
    else Error("invalid C");
}
```

Umgang mit großen First-Mengen

Falls Menge < 4 Elemente: auscodieren (*schneller*)

z.B.: First(A) = {a, b, c}

```
if (la == a || la == b || la == c) ...
```

Falls die Menge ein Intervall bildet: Intervalltest (*schnell*, aber *fehleranfällig*)

```
if (a <= la && la <= c) ...
```

Symbolnummern können oft so vergeben werden, dass häufig abgefragte Mengen Intervalle bilden.

Beispiel

First(A) = { a, c, d }
First(B) = { a, d }
First(C) = { b, e }

```
const int
   a = 0,
   d = 1,
   c = 2,
   b = 3,
   e = 4,
```

First(B)
First(A)
First(C)

Optimierungen beim rekursiven Abstieg

Vermeiden von Mehrfachtests

```
A = a | b.
```

```
private void A () {
    if (la == a) Scan();  // kein Check(a);
    else if (la == b) Scan();
    else Error("invalid A");
}
```

```
A = { a | B d }.
B = b | c.
```

```
private void A () {
    while (la == a || la == b || la == c) {
        if (la == a) Scan();
        else {  // keine Prüfung mehr
            B(); Check(d);
        } // kein Fehlerfall
    }
}
```

Effizienteres Schema für Alternativen in einer Iteration

```
A = { a | B d }.
```

```
private void A () {
    for (;;) {
        if (la == a) Scan();
        else if (la == b || la == c) { B(); Check(d); }
        else break;
    }
}
```

Optimierungen beim rekursiven Abstieg

Häufiges Iterationskonstrukt

α { separator α }

```
for (;;) {
    ... erkenne α ...
    if (la == separator) Scan(); else break;
}
```

Konkretes Beispiel

ident { "," ident }

```
for (;;) {
    Check(TokenKind.ident);
    if (la.Kind == TokenKind.comma) Scan(); else break;
}
```

Eingabe z.B.: a , b , c

Richtige Bestimmung der terminalen Anfänge

Grammatik

```
A = B a.
B = { b } c
  | [ d ]
  | e.
```

terminale Anfänge

b und *c*

d und *a* (!)

e

```
C = D e
  | f.
D = { d }.
```

d und *e* (weil *D* löschbar!)

f

Parser-Methoden

```
private void A () {
   B(); Check(a);
}
```

```
private void B () {
   if (la == b || la == c) {
      while (la == b) Scan();
      Check(c);
   } else if (la == d || la == a) {
      if (la == d) Scan();
   } else if (la == e) {
      Scan();
   } else Error("invalid B");
}
```

```
private void C () {
   if (la == d || la == e) {
      D(); Check(e);
   } else if (la == f) {
      Scan();
   } else Error("invalid C");
}
```

```
private void D () {
   while (la == d) Scan();
}
```

Rekursiver Abstieg und Syntaxbaum

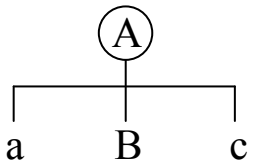
Syntaxbaum wird nur implizit aufgebaut

- durch Methoden, die gerade aktiv sind
- d.h. durch die Produktionen, an denen gerade gearbeitet wird

Beispiel A = a B c.
B = d e.

Aufruf von *A()*

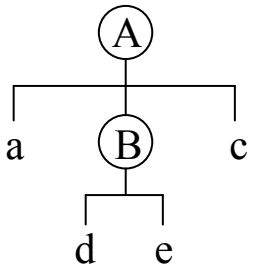
```
private void A () {
    Check(a); B(); Check(c);
}
```



A in Arbeit

Erkennung von *a*
Aufruf von *B()*

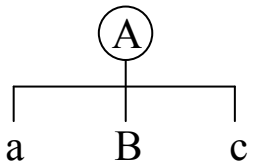
```
private void B () {
    Check(d); Check(e);
}
```



A in Arbeit
B in Arbeit

"Keller"

Erkennung von *d* und *e*
Rückkehr von *B()*



A in Arbeit

3. Syntaxanalyse

LL(1)-Eigenschaft

Vorbedingung für den rekursiven Abstieg

LL(1) ... erkennbar von **L**inks nach rechts
mit **L**inkskanonischen Ableitungen (linkstes NTS zuerst ableiten)
und **1** Vorgriffssymbol

Definition

1. Eine Grammatik ist LL(1), wenn alle ihre Produktionen LL(1) sind.
2. Eine Produktion ist LL(1), wenn für alle ihre Alternativen
   $\alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
   gilt:
   $First(\alpha_i) \cap First(\alpha_j) = \{\}$ (für beliebige *i* und *j*)

Mit anderen Worten

- Die terminalen Anfänge aller Alternativen einer Produktion müssen paarweise disjunkt sein.
- Der Parser muss sich auf Grund des Vorgriffssymbols für eine der Alternativen entscheiden können.

Beseitigung von LL(1)-Konflikten

Faktorisierung

```
IfStatement =  "if" "(" Expr ")" Statement
             |  "if" "(" Expr ")" Statement "else" Statement.
```

Gleiche Anfänge herausziehen

```
IfStatement =  "if" "(" Expr ")" Statement (
                                             | "else" Statement
                                             ).
```

... oder in EBNF

```
IfStatement =  "if" "(" Expr ")" Statement [ "else" Statement ].
```

Faktorisierung mit Auflösen von Nonterminalsymbolen

```
Statement   =  Designator "=" Expr ";"
             |  ident "(" [ ActualParameters ] ")" ";".
Designator  =  ident { "." ident }.
```

Designator in *Statement* einsetzen

```
Statement   =  ident { "." ident } "=" Expr ";"
             |  ident "(" [ ActualParameters ] ")" ";".
```

dann faktorisieren

```
Statement   =  ident  (   { "." ident } "=" Expr ";"
                      |   "(" [ ActualParameters ] ")" ";"
                      ).
```

Beseitigung von Linksrekursion

Linksrekursion ist immer ein LL(1)-Konflikt

Zum Beispiel

```
IdentList = ident | IdentList "," ident.
```

erzeugt folgende Phrasen

```
ident
ident "," ident
ident "," ident "," ident
...
```

kann immer durch Iteration ersetzt werden

```
IdentList = ident { "," ident }.
```

Versteckte LL(1)-Konflikte

EBNF-Optionen und -Iterationen sind versteckte Alternativen

A = [α] β. ⇔ A = α β | β. α und β sind beliebige EBNF-Ausdrücke

Regeln

A = [α] β.	First(α) ∩ First(β) muss {} sein
A = { α } β.	First(α) ∩ First(β) muss {} sein

A = α [β].	First(β) ∩ Follow(A) muss {} sein
A = α { β }.	First(β) ∩ Follow(A) muss {} sein

A = α | .	First(α) ∩ Follow(A) muss {} sein

Beseitigung versteckter LL(1)-Konflikte


```
Name = [ ident "." ] ident.
```

Wo steckt der Konflikt und wie kann er beseitigt werden?

```
Prog = Declarations ";" Statements.
Declarations = D { ";" D }.
```

Wo steckt der Konflikt und wie kann er beseitigt werden?

Dangling Else

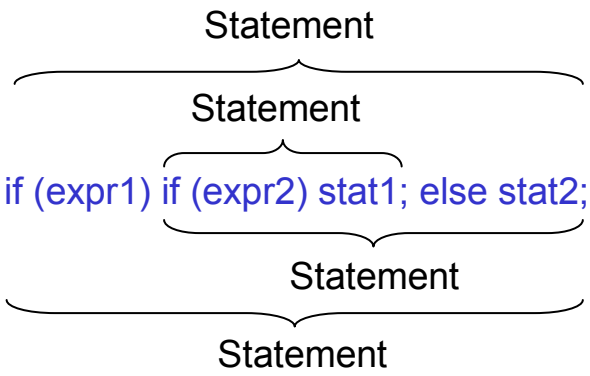
If-Anweisung in C#

```
Statement =  "if" "(" Expr ")" Statement [ "else" Statement ]
           |  ... .
```

Das ist ein LL(1)-Konflikt!

First("else" Statement) ∩ Follow(Statement) = { "else" }

Es ist sogar eine Mehrdeutigkeit, die sich nicht beheben lässt



Es gibt 2 verschiedene Syntaxbäume!

Behandlung von LL(1)-Konflikten

LL(1)-Konflikt ist nur eine Warnung

Der Parser nimmt einfach die erste passende Alternative

```
A = a b c
  | a d.
```

← mit *a* als Vorgriffssymbol nimmt der Parser diese Alternative

Beispiel: Dangling Else

```
Statement = "if" "(" Expr ")" Statement [ "else" Statement ]
          | ... .
```

Wenn das Vorgriffssymbol hier "else" ist,
betritt der Parser die Option;
d.h. das "else" gehört zum innersten "if"

```
if (expr1) if (expr2) stat1; else stat2;
                 Statement
           Statement
```

Das ist hier "zufällig" die gewünschte Interpretation.

Weitere Anforderungen an eine Grammatik

(Vorbedingungen für die Syntaxanalyse)

Vollständigkeit

Für jedes NTS muss es eine Produktion geben

```
A = a B C .
B = b b .
```

falsch!
Keine Produktion für *C*

Terminalisierbarkeit

Jedes NTS muss sich (direkt oder indirekt) in eine Kette von Terminalsymbolen ableiten lassen

```
A = a B | c .
B = b B .
```

falsch!
B kann nicht in lauter Terminalsymbole abgeleitet werden

Zirkularitätsfreiheit

Ein NTS darf nicht (direkt oder indirekt) in sich selbst ableitbar sein ($A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow ... \Rightarrow A$)

```
A = a b | B .
B = b b | A .
```

falsch!
Zirkulär, weil $A \Rightarrow B \Rightarrow A$

3. Syntaxanalyse

Ziele der Syntaxfehlerbehandlung

Anforderungen

1. Der Parser soll möglichst viele Fehler pro Übersetzung finden
2. Der Parser darf auch bei schlimmen Fehlern nicht abstürzen
3. Die Fehlerbehandlung soll die fehlerfreie Analyse nicht bremsen
4. Die Fehlerbehandlung soll den Parser-Code nicht aufblähen

Fehlerbehandlungsmethoden für den rekursiven Abstieg

- Fehlerbehandlung im "Panic Mode"
- Fehlerbehandlung mit allgemeinen Fangsymbolen
- Fehlerbehandlung mit speziellen Fangsymbolen

Panic Mode

Die Syntaxanalyse wird nach dem ersten Fehler abgebrochen

```
void Error (string msg) {
    Console.WriteLine("-- line {0}, col {1}: {2}", la.line, la.col, msg);
    throw new Exception("Panic Mode - exiting after first error");
}
```

Vorteile

- billig
- für kleine Kommandosprachen oder für Interpreter ausreichend

Nachteile

- für größere Sprachen indiskutabel

Fehlerbehandlung mit allgemeinen Fangsymbolen

Beispiel

erwartete Symbolfolge: a b c d ...
gelesene Symbolfolge: a x y d ...

Wiederaufsatz (Synchronisation der restlichen Eingabe mit der Grammatik)

1. **"Fangsymbole" bestimmen, mit denen nach dem Fehler fortgesetzt werden kann.**

   Mit welchen Symbolen kann hier fortgesetzt werden?
   mit *c* Nachfolger von *b* (das an der Fehlerstelle erwartet wurde)
   mit *d* Nachfolger von *b c*
   ...
   Fangsymbole sind an dieser Fehlerstelle {*c*, *d*, ...}

2. **Fehlerhafte Symbole überlesen, bis ein Fangsymbol auftritt.**
   x und *y* werden hier überlesen; mit *d* kann fortgesetzt werden.

3. **Parser an die Stelle in der Grammatik steuern, an der fortgesetzt werden kann.**

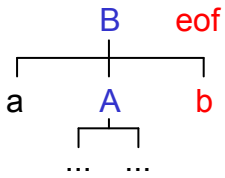
Berechnung der Fangsymbole

Jeder Erkennungsmethode eines Nonterminalsymbols *A* werden die aktuellen Nachfolger von *A* als Parameter mitgegeben

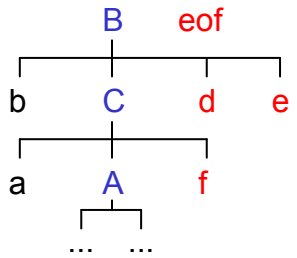
```
private void A (BitArray sux) {
   ...
}
```

sux ... Nachfolger aller NTS, an denen gerade gearbeitet wird

sux_A kann je nach Kontext eine andere Menge sein



$sux_A = \{b, eof\}$



$sux_A = \{f, d, e, eof\}$

sux enthält immer *eof* (Nachfolger des Startsymbols)

Erkennung von Terminalsymbolen

Grammatik

A = ... a s_1 s_2 ... s_n .

$s_i \in TS \cup NTS$

Erkennungsmuster

Check(a, $sux_A \cup First(s_1) \cup First(s_2) \cup ... \cup First(s_n)$);

kann statisch vorausberechnet werden

muss zur Laufzeit berechnet werden

```
private void Check (int expected, BitArray sux) {...}
```

Beispiel

A = a b c.

```
private void A (BitArray sux) {
    Check(a, Add(sux, fs1));
    Check(b, Add(sux, fs2));
    Check(c, sux);
}
```

```
BitArray fs1 = new BitArray();
fs1[b] = true; fs1[c] = true;
```

am Programmbeginn vorberechnet

```
private BitArray Add (BitArray a, BitArray b) {
    BitArray c = (BitArray) a.Clone();
    c.Or(b);
    return c;
}
```

Erkennung von Nonterminalsymbolen

Grammatik

A = ... B s_1 s_2 ... s_n .

Erkennungsmuster

B(sux_A ∪ First(s_1) ∪ First(s_2) ∪ ... ∪ First(s_n));

Beispiel

A = a B c.
B = b b.

```
private void A (BitArray sux) {
    Check(a, Add(sux, fs3));
    B(Add(sux, fs4));
    Check(c, sux);
}
```

fs3 = {b, c}
fs4 = {c}

```
private void B (BitArray sux) {
    Check(b, Add(sux, fs5));
    Check(b, sux);
}
```

fs5 = {b}

Das Startsymbol *S* wird mit *S(fs0);* aufgerufen, wobei *fs0* = {*eof*}

Überlesen fehlerhafter Symbole

Fehler werden in *Check()* bemerkt

```
private void Check (TokenKind expected, BitArray sux) {
   if (la.Kind == expected) Scan();
   else Error(expected + " expected", sux);
}
```

Nach Meldung des Fehlers wird bis zum nächsten Fangsymbol überlesen

```
public void Error (string msg, BitArray sux) {
   Console.WriteLine("-- line {0}, col {1}: {2}", la.line, la.col, msg);
   errors++;
   while (! sux[la.Kind]) Scan();  // while (la ∉ sux) Scan();
   // la ∈ sux
}
```

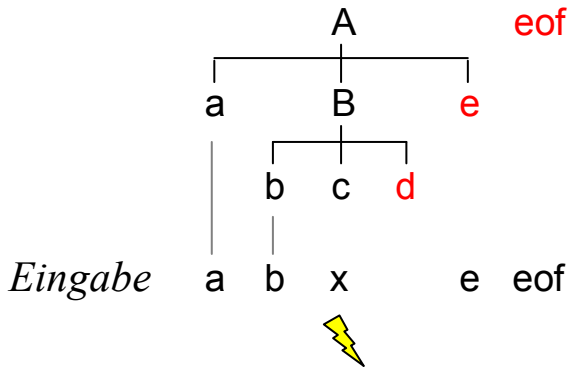
```
int errors = 0;  // Anzahl gefundener Syntaxfehler
```

Synchronisation mit der Grammatik

Beispiel

```
A = a B e.
B = b c d.
```

```
private void A (BitArray sux) {
    Check(a, Add(sux, fs1));
    B(Add(sux, fs2));
    Check(e, sux);
}
private void B (BitArray sux) {
    Check(b, Add(sux, fs3));
    Check(c, Add(sux, fs4));
    Check(d, sux);
}
```



$sux_A = \{eof\}$

$sux_B = \{e, eof\}$

Fehler wird hier entdeckt; Fangsymbole = {*d*, *e*, *eof*}

1. *x* wird überlesen; *la* == *e* ∈ Fangsymbole
2. Parser setzt fort: *Check(d, sux);*
3. liefert wieder einen Fehler; Fangsymbole = {*e*, *eof*}
4. es wird nichts überlesen, weil *la* == *e* ∈ Fangsymbole
5. Parser kehrt aus *B()* zurück und macht *Check(e, sux);*
6. Wiederaufsatz geglückt!

Parser "rumpelt" nach der Fehlerstelle weiter, bis er zu einer Stelle in der Grammatik kommt, an der das gefundene Fangsymbol erwartet wird.

Unterdrücken von Folgefehlermeldungen

Während des Wiederaufsatzes produziert der Parser Folgefehlermeldungen

Abhilfe durch einfache Heuristik

Wenn seit dem letzten Fehler weniger als 3 Symbole erkannt wurden, wird angenommen, dass es sich um einen Folgefehler handelt. Folgefehler werden nicht gemeldet.

```
int errDist = 3;  // "letzter Fehler" liegt mehr als 3 Symbole zurück
```

```
private void Scan () {
   ...
   errDist++;  // wieder ein Symbol erkannt
}
```

```
public void Error (string msg, BitArray sux) {
   if (errDist >= 3) {
      Console.WriteLine("-- line {0}, col {1}: {2}", la.line, la.col, msg);
      errors++;
   }
   while (!sux[la]) Scan();
   errDist = 0;  // Zählung beginnt von neuem
}
```

Erkennung von Alternativen

A = α | β | γ .

α, β, γ sind beliebige EBNF-Ausdrücke

```
private void A (BitArray sux) {
   // vorgezogene Fehlerabfrage, damit im Fehlerfall mit den Anfängen der
   // Alternativen synchronisiert werden kann
   if (la ∉ (First(α) ∪ First(β) ∪ First(γ)))
      Error("invalid A", sux ∪ First(α) ∪ First(β) ∪ First(γ));
   // la passt zu einer der nächsten Alternativen oder ist ein legaler Nachfolger von A
   if (la ∈ First(α)) ... parse α ...
   else if (la ∈ First(β)) ... parse β ...
   else ... parse γ ...  // keine Abfrage mehr; eventuelle Fehler wurden schon gemeldet
}
```

First(α) ∪ First(β) ∪ First(γ)	kann wieder statisch vorberechnet werden
sux ∪ ...	muss zur Laufzeit berechnet werden

Erkennung von Optionen und Iterationen

Optionen

A = [α] β.

```
private void A (BitArray sux) {
   // vorgezogene Fehlerabfrage, damit im Fehlerfall
   // auch mit α synchronisiert werden kann
   if (la ∉ (First(α) ∪ First(β)))
      Error("...", sux ∪ First(α) ∪ First(β));
   // la passt zu α oder β oder ist ein Nachfolger von A
   if (la ∈ First(α)) ... parse α ...;
   ... parse β ...
}
```

Iterationen

A = { α } β.

```
private void A (BitArray sux) {
   for (;;) {
      // Schleife auch betreten, wenn la ∉ First(α)
      if (la ∈ First(α)) ... parse α ...;                  // korrekter Fall 1
      else if (la ∈ First(β) ∪ sux) break;                 // korrekter Fall 2
      else Error("...", sux ∪ First(α) ∪ First(β));        // Fehlerfall
   }
   ... parse β ...
}
```

Beispiel


```
A = a B | b { c d }.
B = [ b ] d.
```

```
private void A (BitArray sux) {
   if (la != a && la != b)
      Error("invalid A", Add(sux, fs1));  // fs1 = {a, b}
   if (la == a) {
      Scan(); B(sux);
   } else if (la == b) {
      Scan();
      for (;;) {
         if (la == c) {
            Scan();
            Check(d, Add(sux, fs2));  // fs2 = {c}
         } else if (sux[la]) {
            break;
         } else {
            Error("c expected", Add(sux, fs2));
         }
      }
   }
}
```

```
private void B (BitArray sux) {
   if (la != b && la != d)
      Error("invalid B", Add(sux, fs3));  // fs3 = {b, d}
   if (la == b) Scan();
   Check(d, sux);
}
```

Zusammenfassung

Fehlerbehandlung mit allgemeinen Fangsymbolen

Vorteil

\+ systematisch anwendbar

Nachteile

- bremst die fehlerfreie Syntaxanalyse
- bläht den Parsercode auf
- ist kompliziert

Fehlerbehandlung mit speziellen Fangsymbolen

Synchronisation erfolgt nur an besonders "sicheren" Stellen

d.h. an Stellen, die mit Schlüsselwörtern beginnen, die an keiner anderen Stelle in der Grammatik vorkommen

Zum Beispiel

- Statement-Anfang: if, while, do, ...
- Deklarationsanfang: public, static, void, ...

Fangsymbolmengen

Allerdings Problem, dass an beiden Stellen auch *ident* vorkommen kann, das kein sicheres Fangsymbol ist ⇒ *ident* aus Fangsymbolmengen weglassen

Code, der an der Synchronisationsstelle eingefügt werden muss

```
...                                  Fangsymbolmenge an dieser Synchronisationsstelle
if (la ∉ expectedSymbols) {
   Error("...");  // keine Nachfolgermengen; kein Überlesen in Error()
   while (la ∉ (expectedSymbols ∪ {eof})) Scan();
}
...                                          damit es zu keiner Endlosschleife kommt
```

- Es müssen keine Nachfolgermengen mitgegeben werden
- Fangsymbolmengen können statisch berechnet werden
- Nach einem Fehler "rumpelt" der Parser bis zur nächsten Synchronisationsstelle weiter

Beispiel

Synchronisation am Statement-Anfang

```
private void Statement () {
   if (!firstStat[la]) {
      Error("invalid start of statement");
      while (!firstStat[la] && la != Token.EOF) Scan();
      errDist = 0;
   }
   if (la == Token.IF) { Scan();
      Check(Token.LPAR); Condition();
      Check(Token.RPAR);
      Statement();
      if (la == Token.ELSE) { Scan(); Statement(); }
   } else if (la == Token.WHILE) {
      ...
}
```

```
BitArray firstStat = new BitArray();
firstStat[Token.WHILE] = true;
firstStat[Token.IF] = true;
...
```

Rest der Syntaxanalyse bleibt wie wenn es keine Fehlerbehandlung gäbe

Keine Synchronisation in *Error()*

```
public void Error (string msg) {
   if (errDist >= 3) {
      Console.WriteLine("-- line {0}, col {1}: {2}", la.line, la.col, msg);
      errors++;
   }
   errDist = 0;
}
```

Heuristik mit *errDist* kann auch hier verwendet werden

Simulation eines Wiederaufsatzes


```
private void Statement () {
   if (!firstStat[la]) {
      Error("invalid start of statement");
      while (!firstStat[la] && la != Token.EOF) Scan();
      errDist = 0;
   }
   if (la == Token.IF) { Scan();
      Check(Token.LPAR); Condition();
      Check(Token.RPAR);
      Statement();
      if (la == Token.ELSE) { Scan(); Statement(); }
   ...
}
```

```
private void Check (TokenKind expected) {
   if (la.Kind == expected) Scan();
   else Error(...);
}
```

```
void Error (string msg) {
   if (errDist >= 3) {
      Console.WriteLine(...);
      errors++;
   }
   errDist = 0;
}
```

Fehlerhafte Eingabe: if a > b then max = a;

la	*Aktion*	
IF	Scan();	IF $\in$ *firstStat* $\Rightarrow$ ok
a	Check(LPAR);	Fehlermeldung: (expected
	Condition();	erkennt a > b
THEN	check(RPAR);	Fehlermeldung:) expected
	Statement();	THEN passt nicht $\Rightarrow$ Fehler, aber keine Meldung
		Überlesen von THEN, Synchronisation mit *ident* (falls in *firstStat*)
max		

Synchronisation am Anfang einer Iteration

Zum Beispiel

```
Block = "{" { Statement } "}".
```

Standardbehandlung bei Synchronisation am Statementanfang

```
static void Block () {
   Check(Token.LBRACE);
   while (firstStat[la])
      Statement();
   Check(Token.RBRACE);
}
```

Problem: Bei falschem Statement-Anfang wird die Schleife nicht betreten bzw. verlassen. Synchronisationsstelle in Statement wird gar nicht erreicht.

Synchronisation am Anfang einer Iteration

Zum Beispiel

```
Block = "{" { Statement } "}".
```

Besser, bereits am Anfang der Iteration synchronisieren

```
static void Block() {
   Check(Token.LBRACE);
   for (;;) {
      if (la ∈ First(Statement)) Statement();      // korrekter Fall 1
      else if (la ∈ {rbrace, eof}) break;          // korrekter Fall 2
      else {                                       // Fehlerfall
         Error("invalid start of Statement");
         do Scan(); while (la ∉ (First(Statement) ∪ {rbrace, eof}));
         errDist = 0;
      }
   }
   Check(Token.RBRACE);
}
```

Keine Synchronisation in *Statement()* mehr

```
static void Statement () {
   if (la == Token.IF) { Scan(); ...
}
```

Zusammenfassung

Fehlerbehandlung mit speziellen Fangsymbolen

Vorteile

+ bremst fehlerfreie Syntaxanalyse nicht
+ bläht Code des Parsers nicht auf
+ einfach

Nachteil

- erfordert Erfahrung und "Tuning"

4. Semantikanschluss und Attributierte Grammatiken

Semantikanschluss

Parser prüft nur syntaktische Korrektheit, führt aber noch keine Übersetzung durch.

Aufgaben des Semantikanschlusses

- **Symbollistenverwaltung**
  - Verwalten deklarierter Namen
  - Speichern von Typstrukturen
  - Verwalten von Gültigkeitsbereichen
- **Prüfung von Kontextbedingungen**
  - Ist jeder verwendete Name deklariert?
  - Haben Operanden die richtigen Typen?
- **Aufruf von Codeerzeugungsmethoden**

Semantische Aktionen werden in den Parser integriert
und mit *attributierten Grammatiken* beschrieben.

Semantische Aktionen

Bisher: Analyse der Eingabe

```
Expr = Term { "+" Term }.
```

Parser prüft, ob eine Eingabe syntaktisch korrekt ist.

Jetzt: *Übersetzung* der Eingabe (Semantische Verarbeitung)

Zum Beispiel: Zählen der Terme des Ausdruck

```
Expr =
   Term          (. int n = 1; .)
   { "+" Term    (. n++; .)
   }             (. Console.WriteLine(n); .)
   .
```

Semantische Aktionen

- beliebige C#-Anweisungen zwischen **(.** und **.)**
- werden vom Parser an der Stelle ausgeführt, an der sie in der Grammatik vorkommen

"Übersetzung" hier:

1+2+3	⇒	3
47+1	⇒	2
909	⇒	1

Attribute

Syntaxsymbole können Werte liefern (Ausgangsparameter)

```
Term<↑int val>
```

Term liefert als Ausgangsattribut seinen Wert

Attribute sind bei der Übersetzung nützlich

Zum Beispiel: Berechnung der Summe der Terme

```
Expr                        (. int sum, val; .)
= Term<↑sum>
  { "+" Term<↑val>          (. sum += val; .)
  }                         (. Console.WriteLine(sum); .)
  .
```

"Übersetzung" hier:

1+2+3 ⇒ 6
47+1 ⇒ 48
909 ⇒ 909

Eingangsattribute

Nonterminalsymbole können auch Eingangsattribute haben
(Parameter, die vom Rufer mitgegeben werden)

```
Expr<↓bool printHex>
```

printHex: Ergebnis der Addition hexadezimal ausgeben (sonst dezimal)

Beispiel

```
Expr<↓bool printHex>          (.  int sum, val; .)
=  Term<↑sum>
   { "+" Term<↑val>           (.  sum += val; .)
   }                          (.  if (printHex) Console.WriteLine("{0:X}", sum)
                                  else Console.WriteLine("{0:D}", sum);
                               .)
.
```

Attributierte Grammatiken

Notation zur Beschreibung von Übersetzungsprozessen

Bestehen aus 3 Teilen

1. Syntaxregeln in EBNF

```
Expr = Term { "+" Term }.
```

2. **Attribute** (Parameter von Syntaxsymbolen)

```
Term<↑int val>
Expr<↓bool printHex>
```

Ausgangsattribute (*synthesized*): liefern Übersetzungsergebnis
Eingangsattribute (*inherited*): stellen Kontext bereit

3. Semantische Aktionen

```
(.   ... beliebige C#-Anweisungen ...   .)
```

Beispiel

ATG zur Verarbeitung von Deklarationen

```
VarDecl                          (. Struct type; .)
= Type<↑type>
  IdentList<↓type>
  ";" .
```

```
IdentList<↓Struct type>
= ident                          (. Tab.Insert(token.str, type); .)
  { "," ident                    (. Tab.Insert(token.str, type); .)
  } .
```

Wird folgendermaßen in Parsercode umgesetzt

```
static void VarDecl () {
   Struct type;
   Type(out type);
   IdentList(type);
   Check(Token.SEMICOLON);
}
```

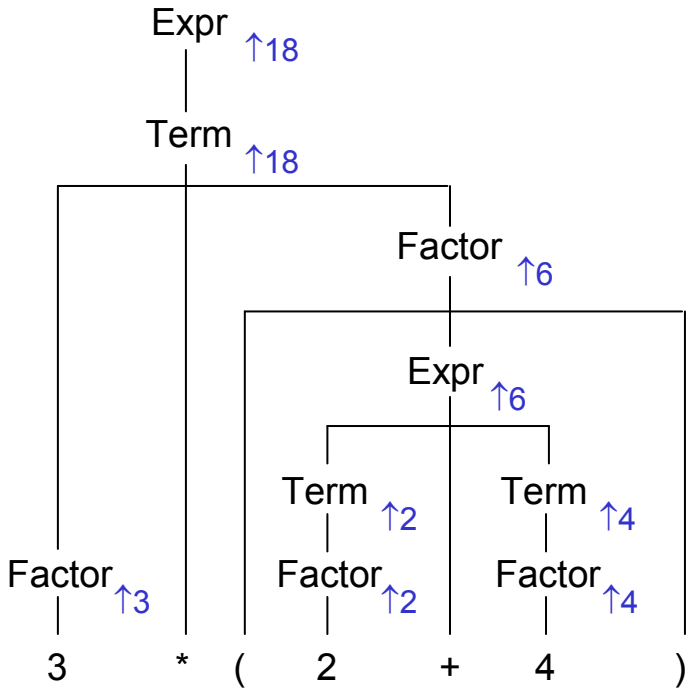
```
static void IdentList (Struct type) {
   Check(Token.IDENT);
   Tab.Insert(token.str, type);
   while (la == Token.COMMA) {
      Scan();
      Check(Token.IDENT);
      Tab.Insert(token.str, type);
   }
}
```

ATGs sind kürzer und
lesbarer als Parsercode

Beispiel: Übersetzung von Konstantenausdrücken

Eingabesatz: 3 * (2 + 4)
Übersetzungsergebnis: 18

```
Expr<↑int val>                (. int val1; .)
=  Term <↑val>
   { "+" Term <↑val1>         (. val += val1; .)
   |  "-" Term <↑val1>        (. val -= val1; .)
   }.

Term<↑int val>                (. int val1; .)
=  Factor <↑val>
   { "*" Factor <↑val1>       (. val *= val1; .)
   |  "/" Factor <↑val1>      (. val /= val1; .)
   }

Factor <↑int val>             (. int val1; .)
=  number                     (. val = token.val; .)
|  "(" Expr<↑val>  ")"
```



Umsetzung in Parsercode

Produktion

```
Expr<↑int val>              (. int val1; .)
=  Term<↑val>
   { "+" Term<↑val1>        (. val += val1; .)
   |  "-" Term<↑val1>       (. val -= val1; .)
   }.
```

Parser-Methode

```
static void Expr (out int val) {
   int val1;
   Term(out val);
   for (;;) {
      if (la == Token.PLUS) {
         Scan();
         Term(out val1);
         val += val1;
      } else if (la == Token.MINUS) {
         Scan();
         Term(out val1);
         val -= val1;
      } else break;
   }
}
```

Eingangsattribute ⇒ Parameter
Ausgangsattribute ⇒ *out*-Parameter
Sem. Aktionen ⇒ eingebetteter C#-Code

Terminalsymbole haben keine Eingangsattribute.

Bei uns haben sie auch keine Ausgangsattribute, sondern ihr Wert befindet sich in *token.str* oder *token.val*.

Beispiel: Verkaufsstatistik

ATGs sind auch außerhalb des Übersetzerbaus nützlich

Beispiel: Datei mit Verkaufszahlen

```
File    = { Article }.
Article = Code { Amount } "END"
Code    = number.
Amount  = number.
```

Immer wenn die Eingabe syntaktisch strukturiert ist, kann man sie mit ATGs verarbeiten.

Eingabe zum Beispiel:

```
3451    2  5  3  7  END
3452    4  8  1 END
3453    1  1 END
...
```

Gewünschte Ausgabe: Artikelnummer und Gesamtverkaufszahl

```
3451    17
3452    13
3453    2
...
```

ATG für Verkaufsstatistik


```
File                                    (. int code, amount; .)
= { Article<↑code, ↑amount>             (. Write(code + " " + amount); .)
  }.

Article<↑int code, ↑int amount>
= Value<↑code>
  {                                     (. int x; .)
    Value<↑x>                           (. amount += x; .)
  }
  "END".

Value<↑int x>
= number                                (. x = token.val; .)
  .
```

Parsercode

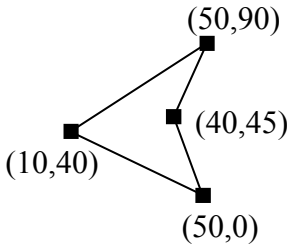
```
static void File () {
   int code, amount;
   while (la == number) {
      Article(out code, out number);
      Write(code + " " + amount);
   }
}
```

```
static void Article (out int code, out int amount) {
   Value(out code);
   while (la == number) {
      int x; Value(out x); amount += x;
   }
   Check(end);
}
```

```
static void Value (out int x) { Check(number); x = token.val; }
```

Terminalsymbole
number, end, eof

Beispiel: Bildbeschreibungssprache



beschrieben durch:

```
POLY
   (10,40)
   (50,90)
   (40,45)
   (50,0)
END
```

Syntax der Eingabe:

```
Polygon = "POLY" Point { Point } "END".
Point = "(" number "," number ")".
```

Gesucht: Lesen der Eingabe und Zeichnen des Polygons

```
Polygon                 (. Pt p, q; .)
=  "POLY"
   Point<↑p>            (. Turtle.Start(p); .)
   { "," Point<↑q>      (. Turtle.Move(q); .)
   }
   "END"                (. Turtle.Move(p); .)
   .

Point<↑p>               (. Pt p; int x, y; .)
=  "(" number           (. x = t.val; .)
   "," number           (. y = t.val; .)
   ")"                  (. p = new Pt(x, y); .)
   .
```

Zeichnen durch Turtle-Grafik

Turtle.Start(p);	setzt den Stift auf Punkt *p*
Turtle.Move(q);	bewegt den Stift nach *q* und zeichnet dabei eine Linie

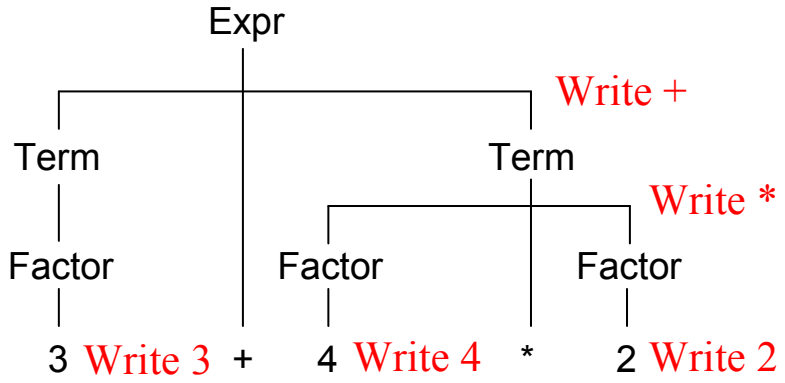
Beispiel: Transformation von Infix nach Postfix

Arithmetischer Ausdruck soll von Infix- nach Postfix-Notation übersetzt werden

3 + 4 * 2 ⇒ 3 4 2 * +

(3 + 4) * 2 ⇒ 3 4 + 2 *

```
Expr
= Term
   { "+" Term     (. Write("+"); .)
   |  "-" Term    (. Write("-"); .)
   }.

Term
= Factor
   { "*" Factor   (. Write("*"); .)
   |  "/" Factor  (. Write("/"); .)
   }.

Factor
= number          (. Write(token.val); .)
|  "(" Expr ")".
```



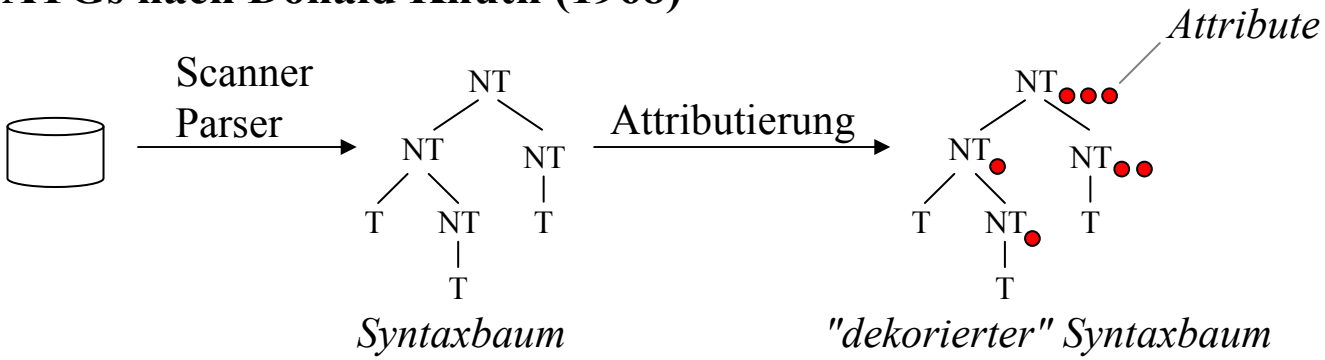
Attributierte Grammatiken nach Knuth

Idee

ATGs bisher: Prozedurale Beschreibung (Übersetzungsalgorithmus)

- Jede Produktion wird von links nach rechts abgearbeitet
- Dabei werden Attribute berechnet und semantische Aktionen ausgeführt

ATGs nach Donald Knuth (1968)



Nonterminalsymbole haben Attribute

- statische Eigenschaften (ändern sich nach der Berechnung nicht mehr)
- Beispiel: Typ eines Ausdrucks, Adresse einer Variablen, ...

Attributierung

Durchlaufen des Syntaxbaums (u.U. mehrmals auf und ab), bis alle Attribute berechnet sind.

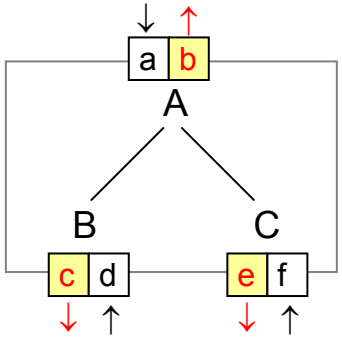
Attributierungsregeln

Definieren für jede Produktion

- *Eingangsattribute* der Symbole auf der *rechten Seite* der Produktion
- *Ausgangsattribute* der Symbole auf der *linken Seite* der Produktion
- wenn nötig, *Kontextbedingungen*

Beispiel

Produktion *p*: $A_{\downarrow a \uparrow b} = B_{\downarrow c \uparrow d}\ C_{\downarrow e \uparrow f}$.



Produktion p
stellt einen Ausschnitt des Syntaxbaums dar

Es müssen alle Attribute definiert werden, die *p* verlassen (d.h. *b*, *c*, *e*)

Attributierungsregeln *R*(*p*) zum Beispiel:

```
c = a;
e = d + foo(a);
b = d + f;
```

Kontextbedingung *CC*(*p*) zum Beispiel:

```
d >= f
```

Beispiel: Berechnung des Werts einer Hex-Zahl

Grammatik (muss in BNF vorliegen, damit man einen Syntaxbaum aufbauen kann)

```
Number = Digits.          // Dezimalzahl
Number = Digits "H".      // Hexadezimalzahl
Digits = hex.             // hex ... 0..9, A..F
Digits = Digits hex.
```

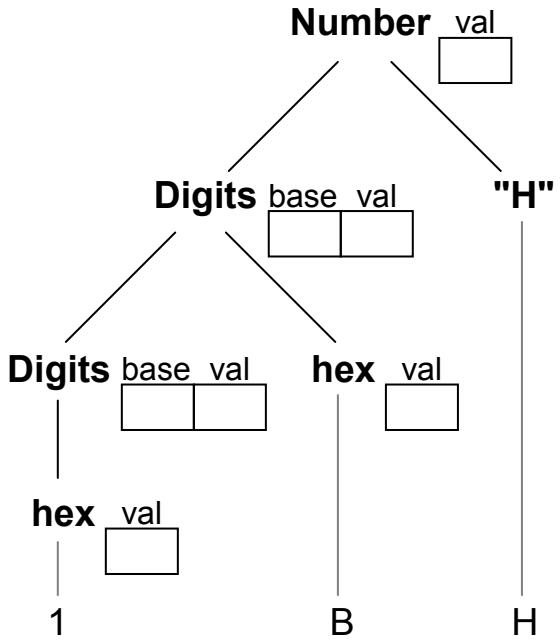
Attribute

Number ↑val

Digits ↓base ↑val

hex ↑val

Syntaxbaum für die Eingabe: 1BH



Attribute sind noch nicht berechnet

Attributierungsregeln

Produktion 1

$Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$.
Digits.base = 10; Number.val = Digits.val;

Produktion 2

$Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$ "H".
Digits.base = 16; Number.val = Digits.val;

Produktion 3

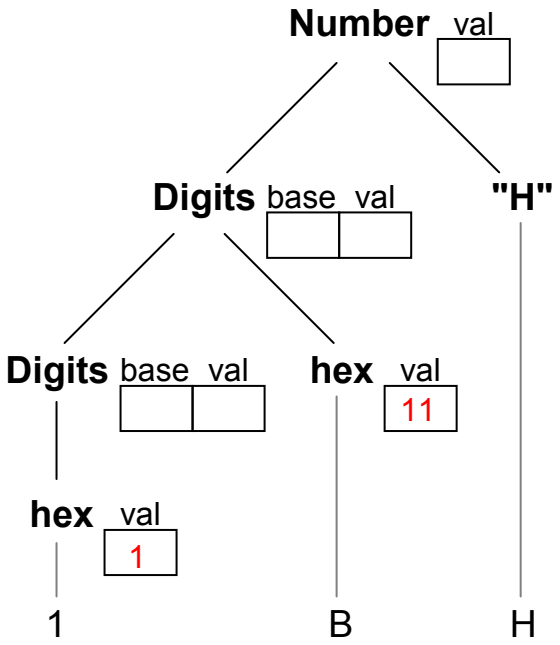
$Digits_{\downarrow base \uparrow val} = hex_{\uparrow val}$.
Digits.val = hex.val; CC: Digits.base == 10 && $0 \leq$ hex.val ≤ 9 || Digits.base == 16 && $0 \leq$ hex.val ≤ 15

Produktion 4

$Digits_{\downarrow base \uparrow val} = Digits1_{\downarrow base \uparrow val}\ hex_{\uparrow val}$.
Digits1.base = Digits.base; Digits.val = Digits1.val * Digits.base + hex.val; CC: Digits.base == 10 && $0 \leq$ hex.val ≤ 9 || Digits.base == 16 && $0 \leq$ hex.val ≤ 15

Attributierung des Baums

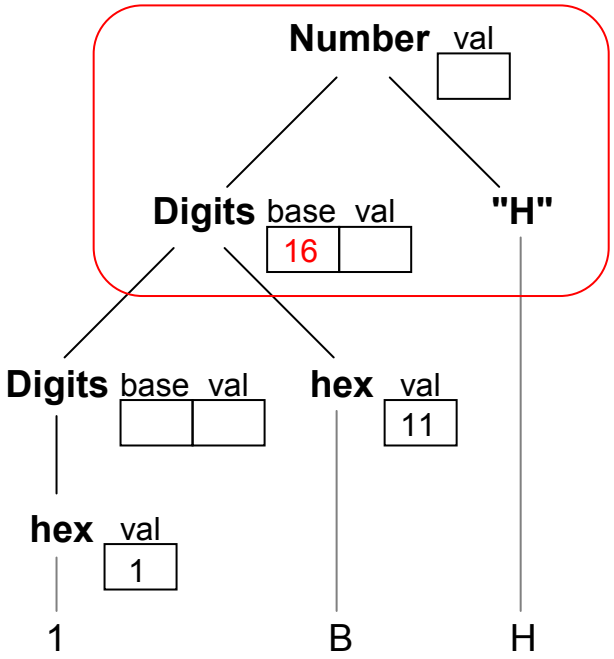
Scanner füllt die Attributwerte der Terminalsymbole



Attributierung des Baums (Forts.)

Durchlauf des Baums von oben nach unten

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind.



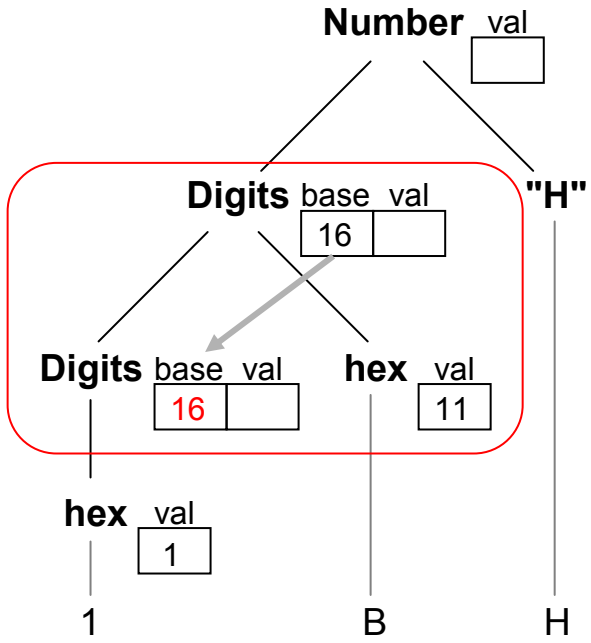
Produktion 2

$Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$ "H".
Digits.base = 16; Number.val = Digits.val;

Attributierung des Baums (Forts.)

Durchlauf des Baums von oben nach unten

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind



Produktion 4

$Digits_{\downarrow base \uparrow val} = Digits1_{\downarrow base \uparrow val}\ hex_{\uparrow val}$.

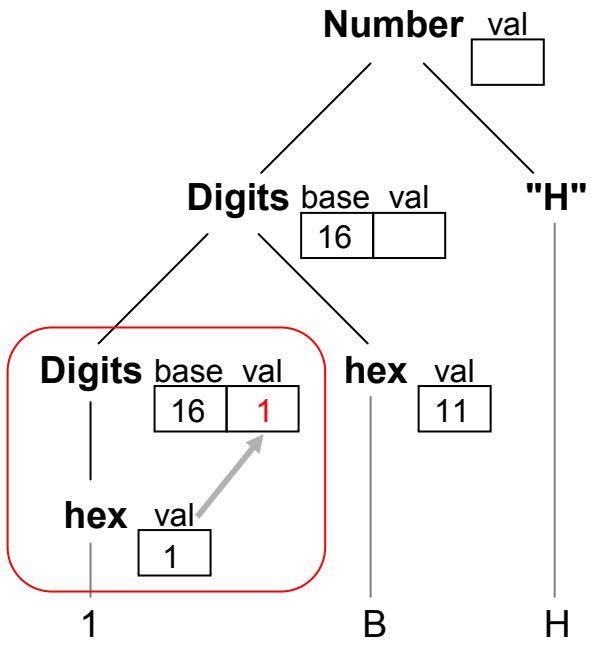
Digits1.base = Digits.base;
Digits.val = Digits1.val * Digits.base + hex.val;

CC: Digits.base == 10 && $0 \leq$ hex.val ≤ 9
|| Digits.base == 16 && $0 \leq$ hex.val ≤ 15

Attributierung des Baums (Forts.)

Durchlauf des Baums von oben nach unten

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind



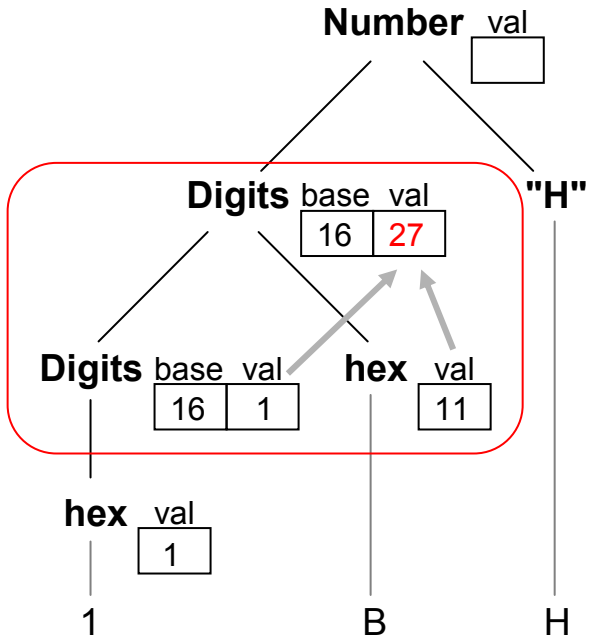
Produktion 3

$Digits_{\downarrow base \uparrow val} = hex_{\uparrow val}$.
Digits.val = hex.val; CC: Digits.base == 10 && 0 ≤ hex.val ≤ 9 || Digits.base == 16 && 0 ≤ hex.val ≤ 15

Attributierung des Baums (Forts.)

Durchlauf des Baums von unten nach oben

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind



Produktion 4

$Digits_{\downarrow base \uparrow val} = Digits1_{\downarrow base \uparrow val}\ hex_{\uparrow val}$.

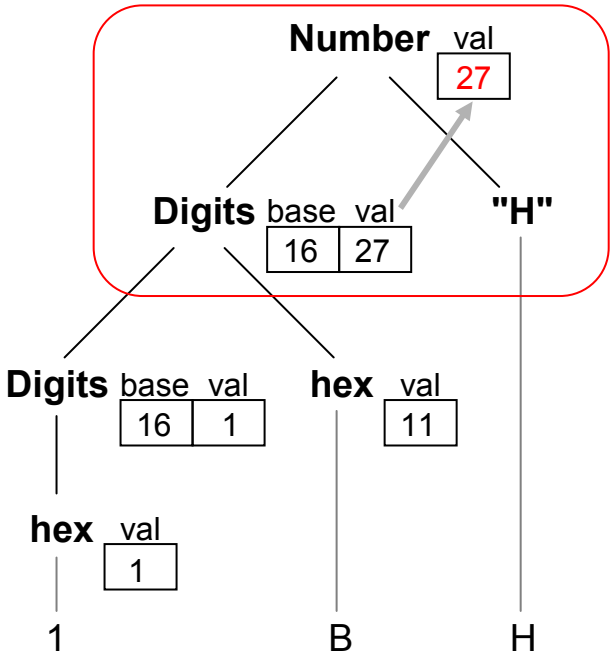
Digits1.base = Digits.base;
Digits.val = Digits1.val * Digits.base + hex.val;

CC: Digits.base == 10 && $0 \le$ hex.val ≤ 9
|| Digits.base == 16 && $0 \le$ hex.val ≤ 15

Attributierung des Baums (Forts.)

Durchlauf des Baums von unten nach oben

Für jede Produktion wird geprüft, welche Attributierungsregeln bereits ausführbar sind



Produktion 2

$Number_{\uparrow val} = Digits_{\downarrow base \uparrow val}$ "H".
Digits.base = 16; Number.val = Digits.val;

Alle Attribute sind berechnet ⇒ Ende des Durchlaufs

Definition einer ATG nach Knuth

Attributierte Grammatik

ATG = (CFG, A, R, CC)

CFG ...	kontextfreie Grammatik
A ...	Menge von Attributen
R ...	Menge von Attributierungsregeln
CC ...	Menge von Kontextbedingungen

Kontextfreie Grammatik

CFG = (T, N, P, S)

T ...	Terminalsymbole
N ...	Nonterminalsymbole
P ...	Produktionen
S ...	Startsymbol

Attribute

A(X) ... Attribute des Symbols X (geschrieben als X.a, X.b, ...)
- AS(X) ... Ausgangsattribute von X (synthesized)
- AI(X) ... Eingangsattribute von X (inherited)

Attributierungsregeln

R(p) ... Attributierungsregeln für Produktion p: X0 = X1 ... Xn
- R(p) = {Xi.a = f(Xj.b, ..., Xk.c)}
- für alle AS der linken und alle AI der rechten Seite von p

Kontextbedingungen

CC(p) ... Kontextbedingungen für Produktion p: X0 = X1 ... Xn
- in Form eines Booleschen Ausdrucks B(Xi.a, ..., Xj.b)
- prüfen die "statische Semantik", d.h. ob die Eingabe semantisch korrekt ist

Vollständige ATGs

Definition

Eine ATG heißt *vollständig*, wenn für alle Produktionen
p: X = Y1 ... Yn
gilt: alle AS(X) und alle AI(Yi) werden in R(p) eingestellt

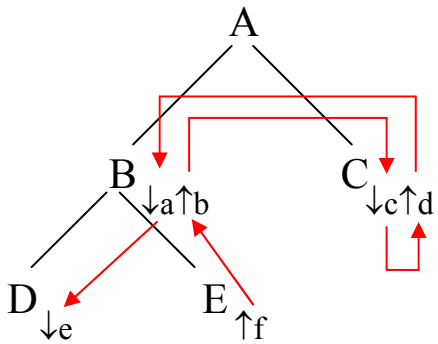
Wohlgeformte ATGs (WAGs)

Definition

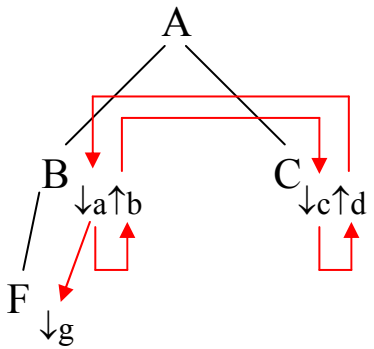
Eine ATG heißt *wohlgeformt* (well-defined, WAG)
- wenn die ATG vollständig ist und
- wenn die Attributbeziehungen in *jedem möglichen Syntaxbaum* kreisfrei sind

Mit anderen Worten:
Für jeden möglichen Syntaxbaum lässt sich eine Attributierungsreihenfolge finden.

Beispiel



wohlgeformt



zirkulär ⇒ nicht wohlgeformt

Prüfung auf Wohlgeformtheit ist *NP-vollständig* (exponentielles Zeitverhalten)!
Es gibt aber Unterklassen von WAG, bei denen das einfacher festzustellen ist.

Geordnete ATGs (OAGs)

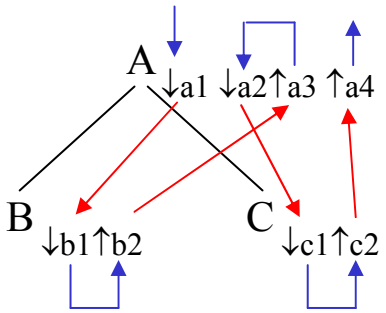
Definition

Eine ATG heißt *geordnet* (OAG), wenn sich *für jede Produktion* unabhängig von ihrem Kontext im Syntaxbaum eine *fixe Attributierungsreihenfolge* angeben lässt.

Attributierungscode einer Produktion *p* kann z.B. durch folgende Operationen angegeben werden:

$comp_i$	...	führe Attributierungsregel *i* in R(*p*) aus
up	...	gehe zum Vater im Syntaxbaum
$down_i$	...	gehe zu Sohn *i* im Syntaxbaum

Beispiel

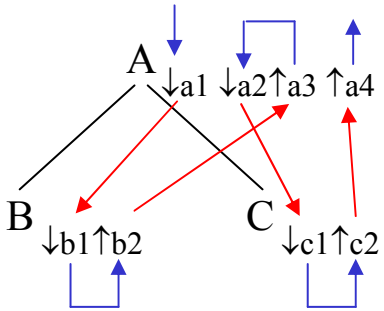


Attributierungscode

comp (b1 = a1)
$down_B$
comp (a3 = b2)
up
comp (c1 = a2)
$down_C$
comp (a4 = c2)
up

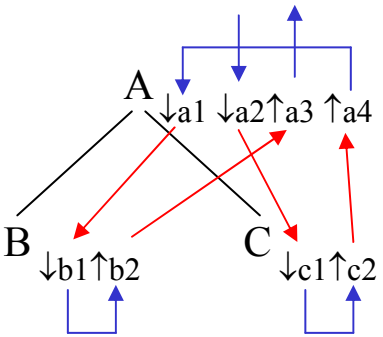
Gegenbeispiel

ATG, die nicht geordnet ist



Attributierungscode

comp (b1 = a1)
$down_B$
comp (a3 = b2)
up
comp (c1 = a2)
$down_C$
comp (a4 = c2)
up



Attributierungscode

comp (c1 = a2)
$down_C$
comp (a4 = c2)
up
comp (b1 = a1)
$down_B$
comp (a3 = b2)
up

Reihenfolge der Attributierung hängt davon ab, wo *A* im Syntaxbaum vorkommt.

Prüfung, ob eine Grammatik OAG ist, kann in Polynomialzeit erfolgen.

L-attributierte ATGs (LAGs)

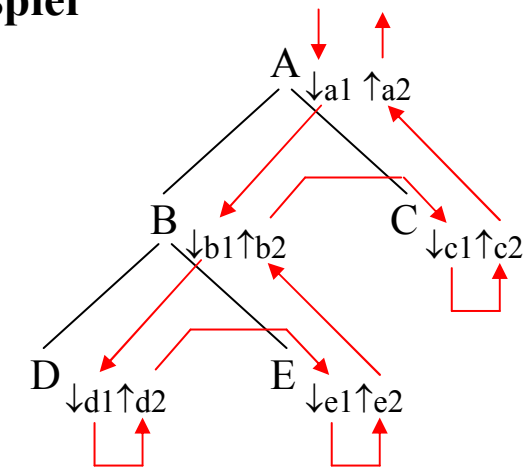
Definition

Eine ATG heißt *L-attributiert* (LAG), wenn sämtliche Attribute im Syntaxbaum sich in einem einzigen Durchgang (runter und rauf, von links nach rechts) berechnen lassen.

Mit anderen Worten:

Wenn man die Attribute schritthaltend mit der Syntaxanalyse berechnen kann.
Für LAGs muss nicht einmal ein Syntaxbaum aufgebaut werden (entspricht unseren prozeduralen ATGs).

Beispiel



Information, die weiter vorne im Programm steht, kann nach hinten transportiert werden, aber nicht umgekehrt!

Beziehungen zwischen ATG-Klassen

ATG ⊃ WAG ⊃ OAG ⊃ LAG

das heißt
- jede LAG ist geordnet
- jede OAG ist wohlgeformt

WAG ist mächtiger als OAG.
OAG ist mächtiger als LAG.

5. Symbolliste

Aufgaben der Symbolliste

1. Speicherung aller deklarierten Namen und ihrer Attribute

- Typ
- Wert (bei Konstanten)
- Adresse (bei lokalen Variablen und Methodenargumenten)
- Parameter (bei Methoden)
- ...

2. Suchen der Attribute eines Namens

- Abbildung: Name $\Rightarrow$ (Typ, Wert, Adresse, ...)

Inhalt der Symbolliste

- Symbolknoten: Informationen über deklarierte Namen
- Strukturknoten: Informationen über Typstrukturen

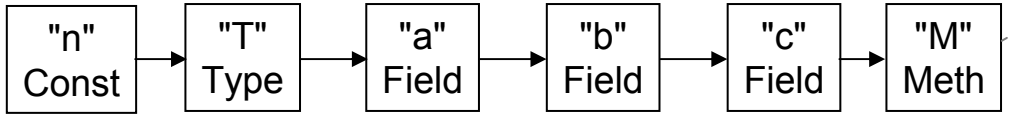
=> Es bietet sich eine dynamische Datenstruktur an
- Lineare Liste
- Binärbaum
- Hashtabelle

Symbolliste als lineare Liste

Gegeben: folgende Deklarationen

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

Symbolliste als lineare Liste



für jeden deklarierten Namen gibt es einen Symbolknoten

\+ einfach
\+ enthält gleichzeitig Deklarationsreihenfolge (wichtig falls Adressvergabe erst später erfolgt)

\- lange Suchzeiten, wenn es viele Deklarationen gibt

Einfachste Schnittstelle

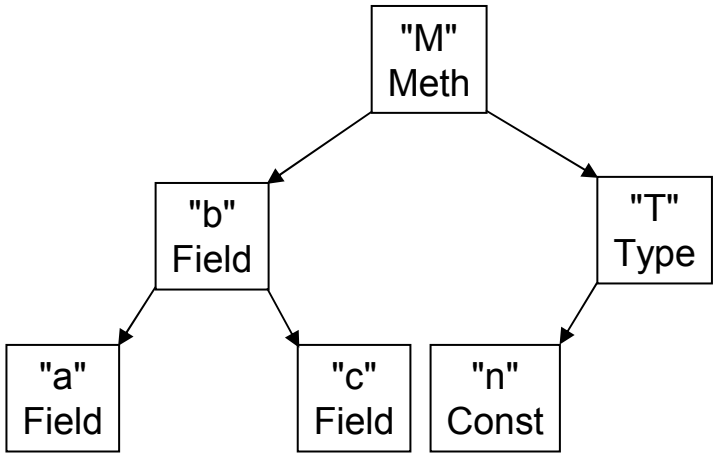
```
public class Tab {
   public static Symbol Insert (Symbol.Kinds kind, string name, ...);
   public static Symbol Find (string name);
}
```

Symbolliste als Binärbaum

Deklarationen

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

Symbolliste als Binärbaum



\+ schnellere Suchzeiten

\- kann degenerieren, wenn nicht balanciert
\- mehr Speicherbedarf
\- Deklarationsreihenfolge geht verloren

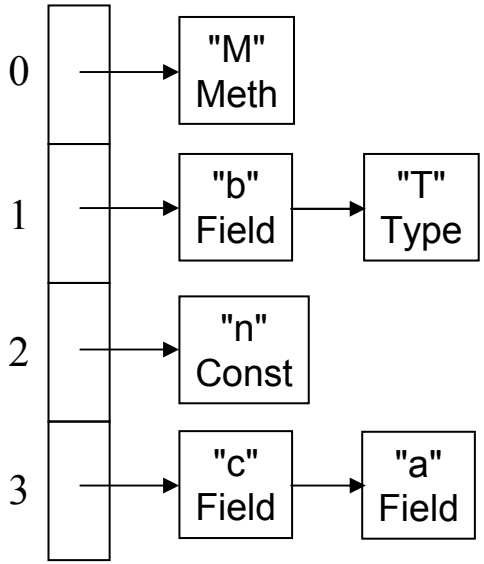
Nur bei sehr vielen Deklarationen sinnvoll.

Symbolliste als Hashtabelle

Deklarationen

```
const int n = 10;
class T { ... }
int a, b, c;
void M () { ... }
```

Symbolliste als Hashtabelle



\+ schnelle Suchzeiten

\- komplizierter als lineare Liste
\- Deklarationsreihenfolge geht verloren

Für unserer Zwecke reicht eine lineare Liste

- Jeder Scope ist ohnehin eine eigene Liste
- Ein Scope hat kaum mehr als 10 Namen

5. Symbolliste

Symbolknoten

Jeder deklarierte Name wird in einem Symbolknoten gespeichert.

Arten von Symbolen in Micro

- Konstanten
- Methodenargumente
- Lokale Variablen
- Typen
- Funktionen

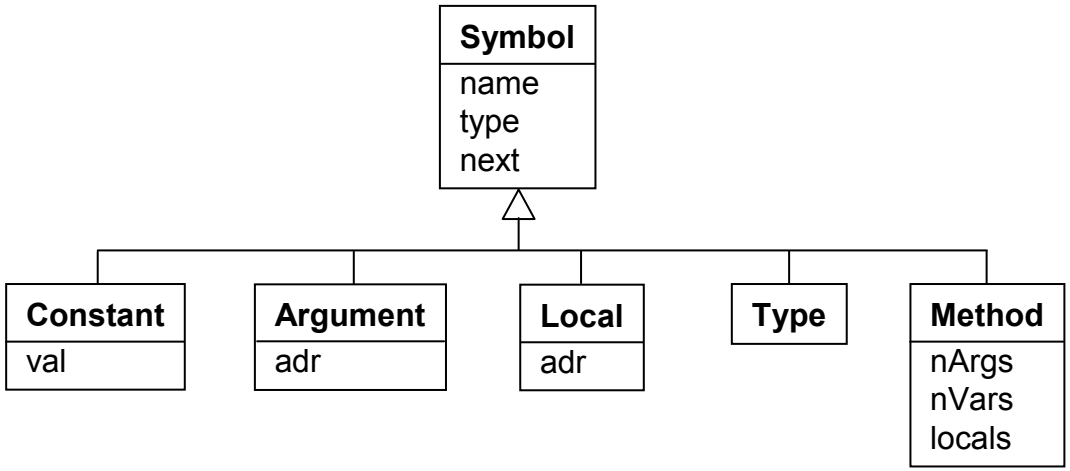
```
public enum SymbolKinds {
   Const,
   Arg,
   Local,
   Type,
   Func
}
```

Was muss über Symbole gespeichert werden?

• für alle Symbole	Name, Typstruktur, Symbolart, Next-Zeiger
• für Konstanten	Wert
• für Methodenargumente	Adresse (= Deklarationsreihenfolge)
• für lokale Variablen	Adresse (= Deklarationsreihenfolge)
• für Methoden	Argumentzahl, Variablenzahl, lokale Symbole (Args + lok. Vars)

Mögliche objektorientierte Struktur

Folgende Klassenhierarchie wäre naheliegend



Ist aber umständlich wegen dauernder Type Casts

```
Symbol sym = Tab.Find("x");
if (sym is Argument) ((Argument) sym).adr = ...;
else if (sym is Method) ((Method) sym).nArgs = ...;
...
```

Daher "flache Implementierung": Alle Attribute in derselben Klasse.
Ist vertretbar, da

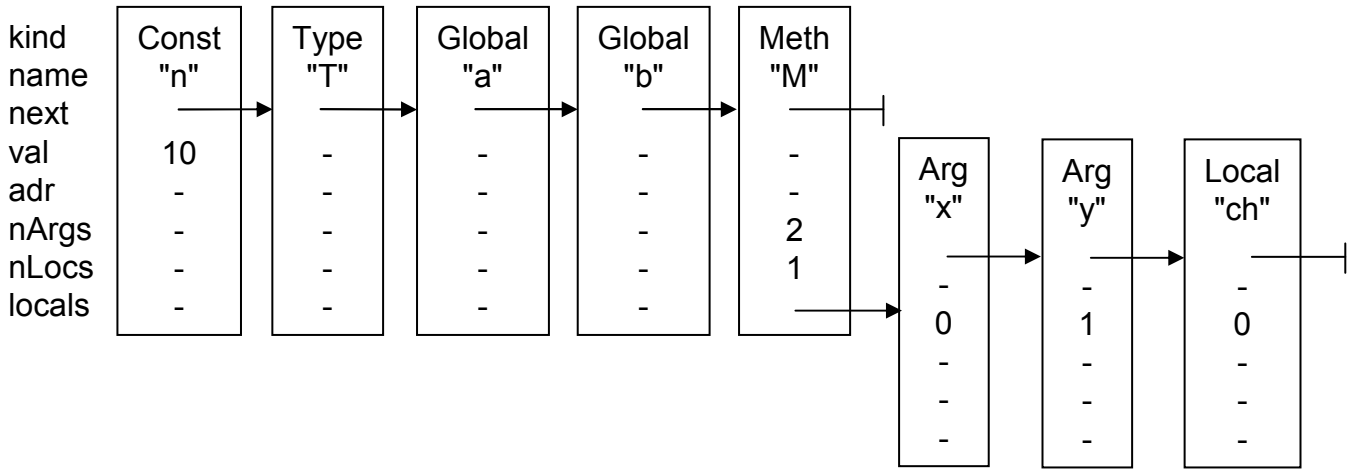
- Anzahl der Symbol-Varianten fix (keine Erweiterbarkeit erforderlich)
- Keine dynamische Bindung nötig

Klasse Symbol


```
class Symbol {
   SymbolKind kind;
   string     name;
   Struct     type;
   Symbol     next;
   int        val;      // Const: value
   int        adr;      // Arg, Local: address
   int        nArgs;    // Meth: number of arguments
   int        nLocs;    // Meth: number of local variables
   Symbol     locals;   // Meth: parameters & local variables; Prog: symbol table of program
}
```

Beispiel

```
const int n = 10;
class T { ... }
int a, b;
void M (int x, int y)
   char ch;
{ ... }
```



Eintragen von Namen in die Symbolliste

Bei jeder Deklaration wird folgende Methode aufgerufen

```
Symbol sym = Tab.Insert(kind, name, type);
```

- erzeugt neuen Symbolknoten mit *kind*, *name*, *type*
- vergibt fortlaufende Adressen für Argumente und Variablen (jeweils bei 0 beginnend)
- prüft, ob *name* bereits deklariert ist (wenn ja, Fehlermeldung)
- hängt den neuen Knoten ans Ende der Symbolliste
- gibt den neuen Knoten als Funktionswert zurück

Beispiel für den Aufruf von *Insert()*

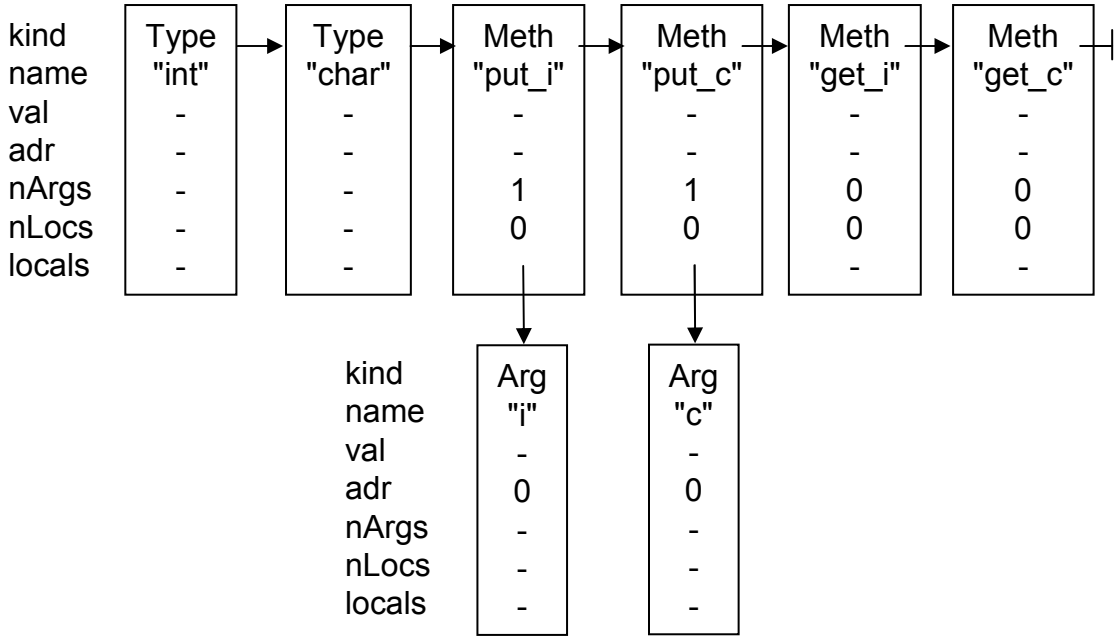
```
VarDecl<↓Symbol.Kinds kind>
=  Type<↑type>
   ident                    (. Tab.Insert(kind, token.str, type); .)
   { "," ident              (. Tab.Insert(kind, token.str, type); .)
   }.
```

Vordeklarierte Namen

Welche Namen sind in Micro vordeklariert?

- Standardtypen: int, char
- Standardmethoden: put_i(int), put_c(char), get_i(), get_c()

Vordeklarierte Namen werden in der Symbolliste gespeichert ("Universum")



Sondernamen als Schlüsselwörter

***int* und *char* könnten auch als Schlüsselwörter implementiert werden.**

erfordert aber Sonderbehandlung in der Grammatik

```
Type<↑Struct type>
=  ident                (. Symbol sym = Tab.Find(token.str); type = sym.type; .)
|  "int"                (. type = Tab.intType; .)
|  "char"               (. type = Tab.charType; .)
.
```

Es ist einfacher, sie in der Symbolliste vorzudeklarieren.

```
Type<↑Struct type>
=  ident                (. Symbol sym = Tab.Find(token.str); type = sym.type; .)
```

\+ einheitliche Behandlung vordefinierter und benutzerdefinierter Typen
\- jemand kann "int" als Namen verwenden

5. Symbolliste

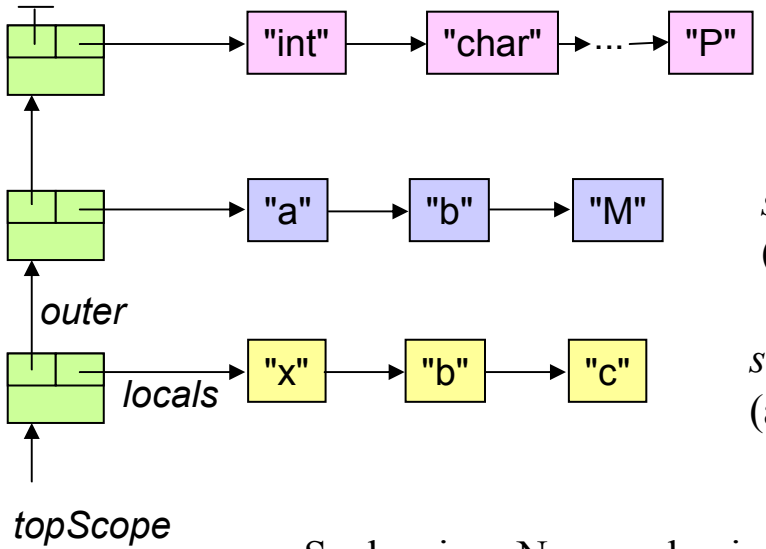
Scope = Gültigkeitsbereich für Namen

Jeweils 1 Scope für

- "Universum" enthält vordeklarierte Namen (und Programm-Symbol)
- Programm enthält globale Namen (= Konstanten, globale Variablen, Klassen, Methoden)
- jede Methode enthält lokale Namen (= Argumente und lokale Variablen)
- jede Klasse enthält Felder

Beispiel

```
class P
   int a, b;
{
   void M (int x)
      int b, c;
   {
      ...
   }
   ...
}
```



Universum
(alle vordeklarierten Namen)

scope P
(alle in *P* deklarierten Namen)

scope M
(alle in *M* deklarierten Namen)

- Suche eines Namens beginnt immer in *topScope*
- Wenn nicht gefunden, im nächstäußeren Scope suchen
- Beispiel: Suche von *b*, *a* und *int*

Scope-Knoten


```
class Scope {
   Scope   outer;   // zum nächstäußeren Scope
   Symbol  locals;  // zur Liste der Symbole in diesem Scope
   int     nArgs;   // Anzahl der Argumente in diesem Scope (zur Adressvergabe)
   int     nLocs;   // Anzahl der lokalen Variablen in diesem Scope (zur Adressvergabe)
}
```

Erzeugen von Scopes

```
static void OpenScope() { // in class Tab
   Scope s = new Scope();
   s.nArgs = 0; s.nLocs = 0;
   s.outer = topScope;
   topScope = s;
}
```

- aufgerufen am Beginn einer Methode oder Klasse
- verkettet neuen Scope mit den bestehenden
- neuer Scope wird *topScope*
- *Tab.Insert()* trägt Namen immer in *topScope* ein

Schließen von Scopes

```
static void CloseScope() { // in class Tab
   topScope = topScope.outer;
}
```

- aufgerufen am Ende einer Methode oder Klasse
- macht nächstäußeren Scope zu *topScope*

Einfügen von Namen in Scopes

Namen werden bei ihrer Deklaration immer in *topScope* eingetragen.

```
class Tab {
   static Scope topScope;            // Zeiger auf aktuellen Scope
   ...
   static Symbol Insert (Symbol.Kinds kind, string name, Struct type) {
      //--- erzeugen
      Symbol sym = new Symbol(name, kind, type);

      if (kind == Symbol.Kinds.Arg) sym.adr = topScope.nArgs++;
      else if (kind == Symbol.Kinds.Local) sym.adr = topScope.nLocs++;

      //--- einfügen
      Symbol cur = topScope.locals, last = null;
      while (cur != null) {
         if (cur.name == name) Error(name + " declared twice");
         last = cur; cur = cur.next;
      }
      if (last == null) topScope.locals = sym;
      else last.next = sym;
      return sym;
   }
   ...
}
```

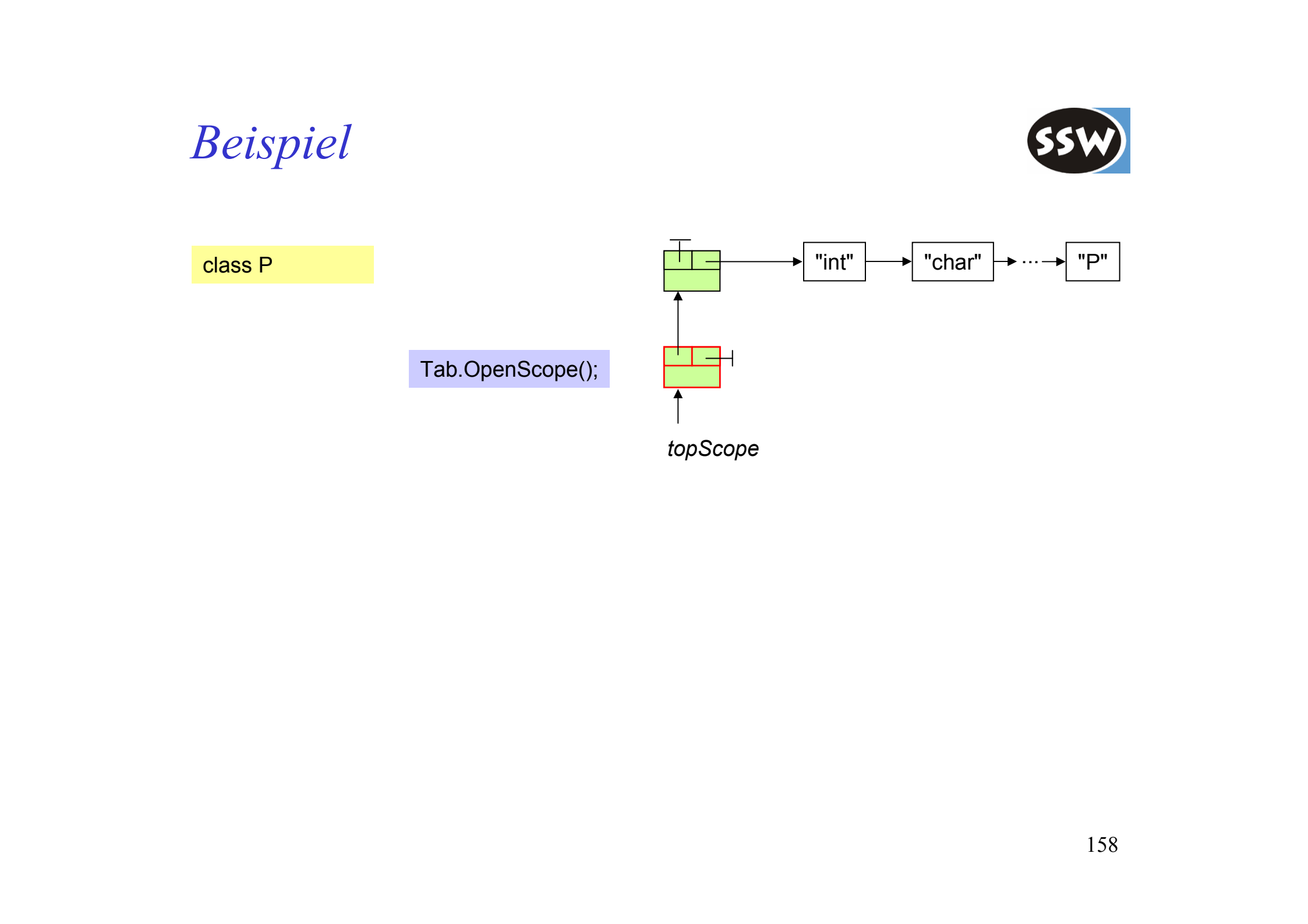
Öffnen und Schließen von Scopes


```
Function                (. Struct type; .)
=  Type<↑type>
   ident                (.  curMethod = Tab.insert(Symbol.Kinds.Meth, token.str, type);
                            Tab.OpenScope();
                        .)
   ...
   "{"
   ...
   "}"                  (.  curMethod.nArgs = topScope.nArgs;
                            curMethod.nLocs = topScope.nLocs;
                            curMethod.locals = Tab.topScope.locals;
                            Tab.CloseScope();
                        .)
   .
```

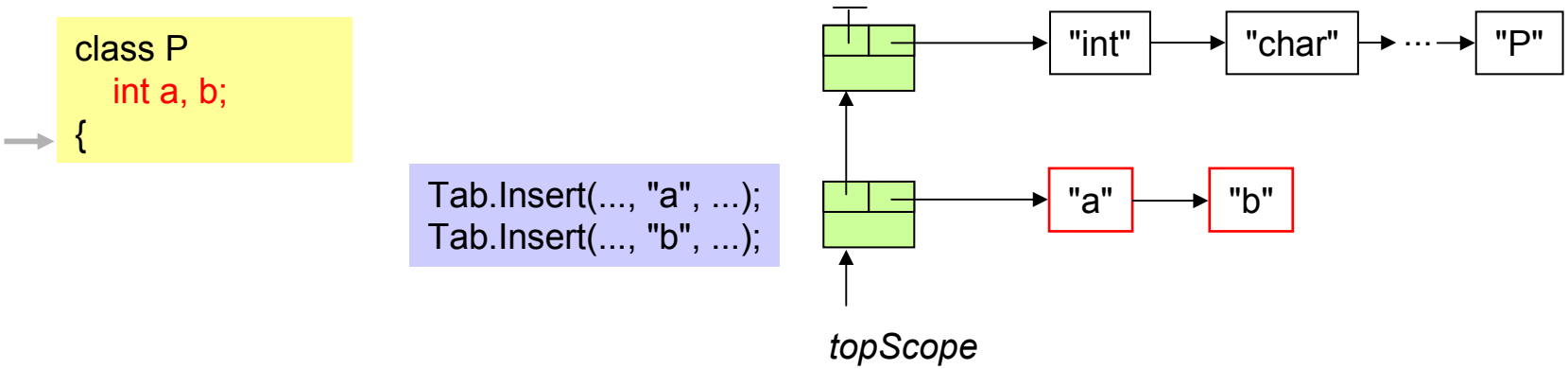
Beachte

- Methodenname wird noch in äußeren Scope eingetragen
- Bevor der Scope geschlossen wird, werden seine lokalen Symbole an *m.locals* gehängt
- Auch bei Klassen wird ein Scope geöffnet und wieder geschlossen.

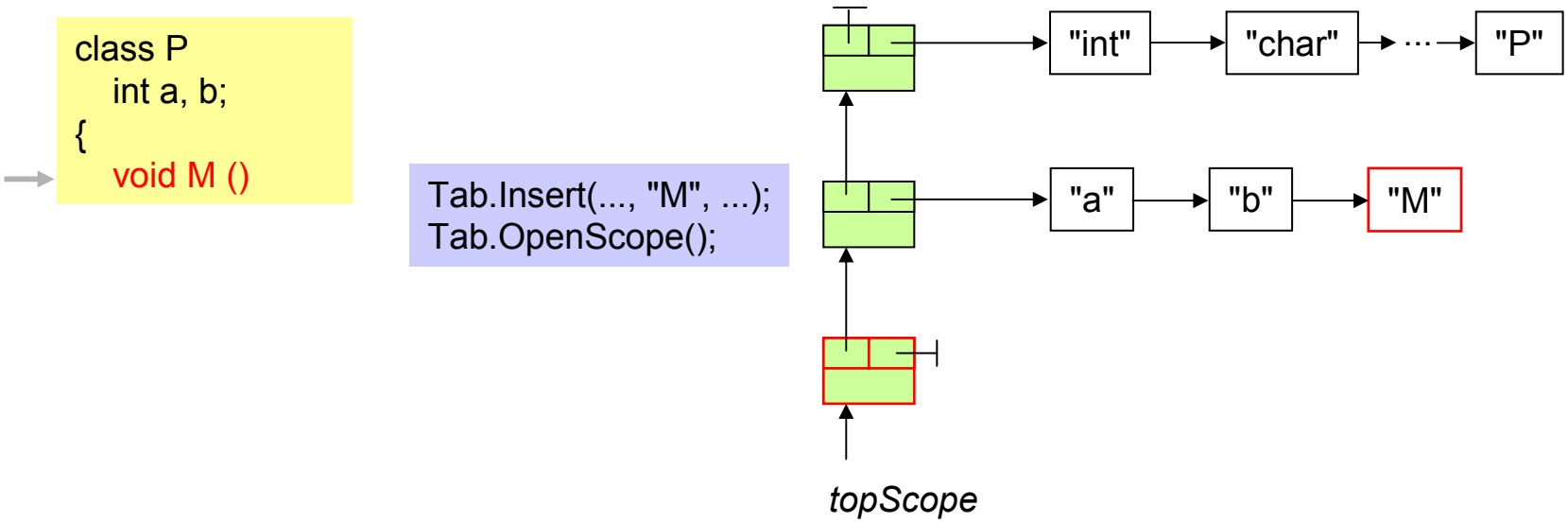
Beispiel



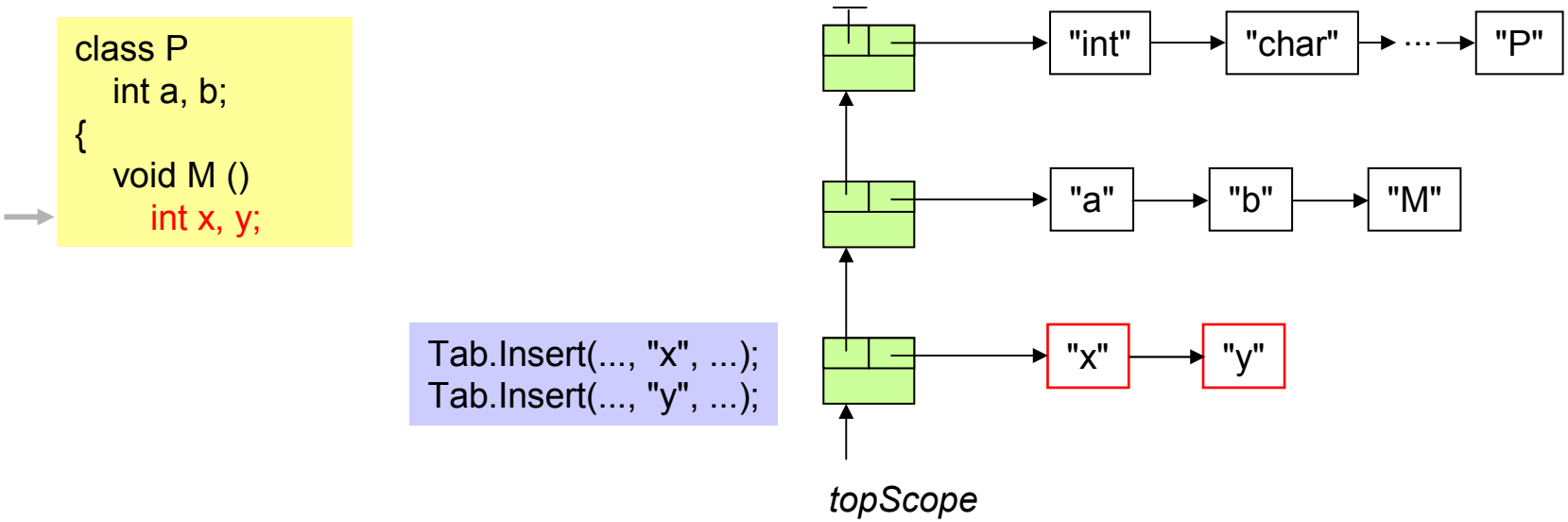
Beispiel



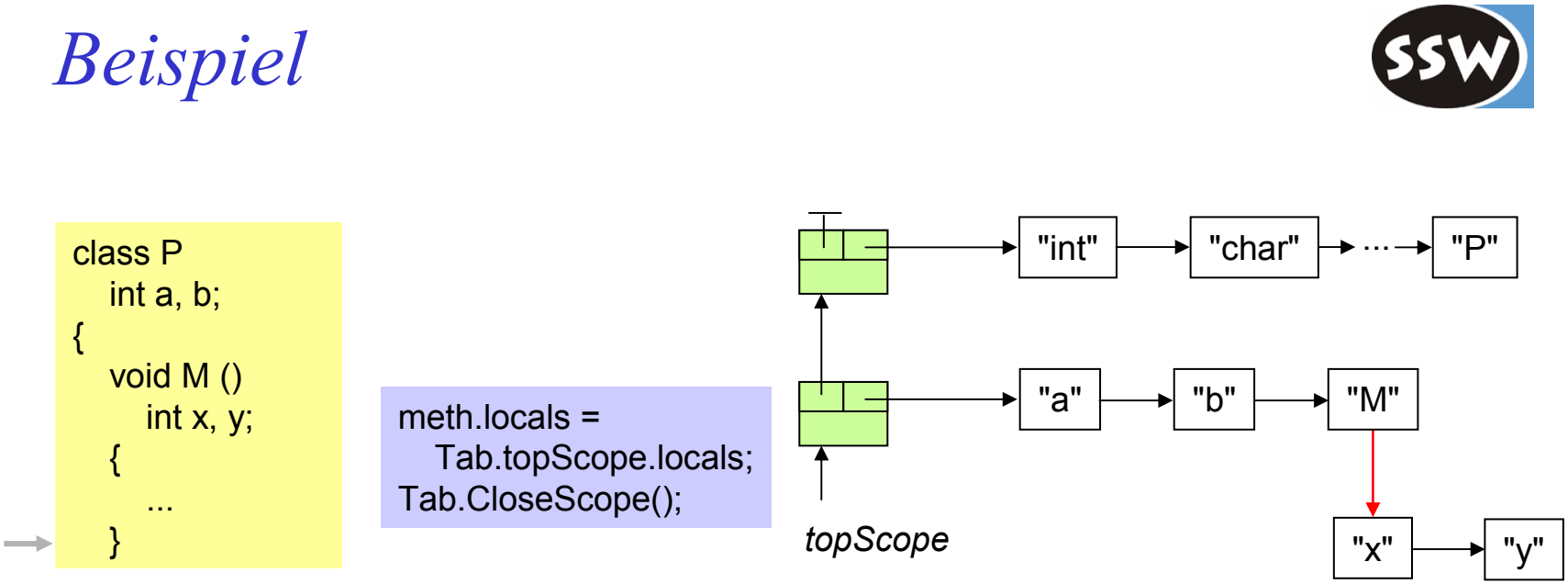
Beispiel


class P
int a, b;
{
void M ()
Tab.Insert(..., "M", ...);
Tab.OpenScope();
"int"
"char"
···
"P"
"a"
"b"
"M"
topScope

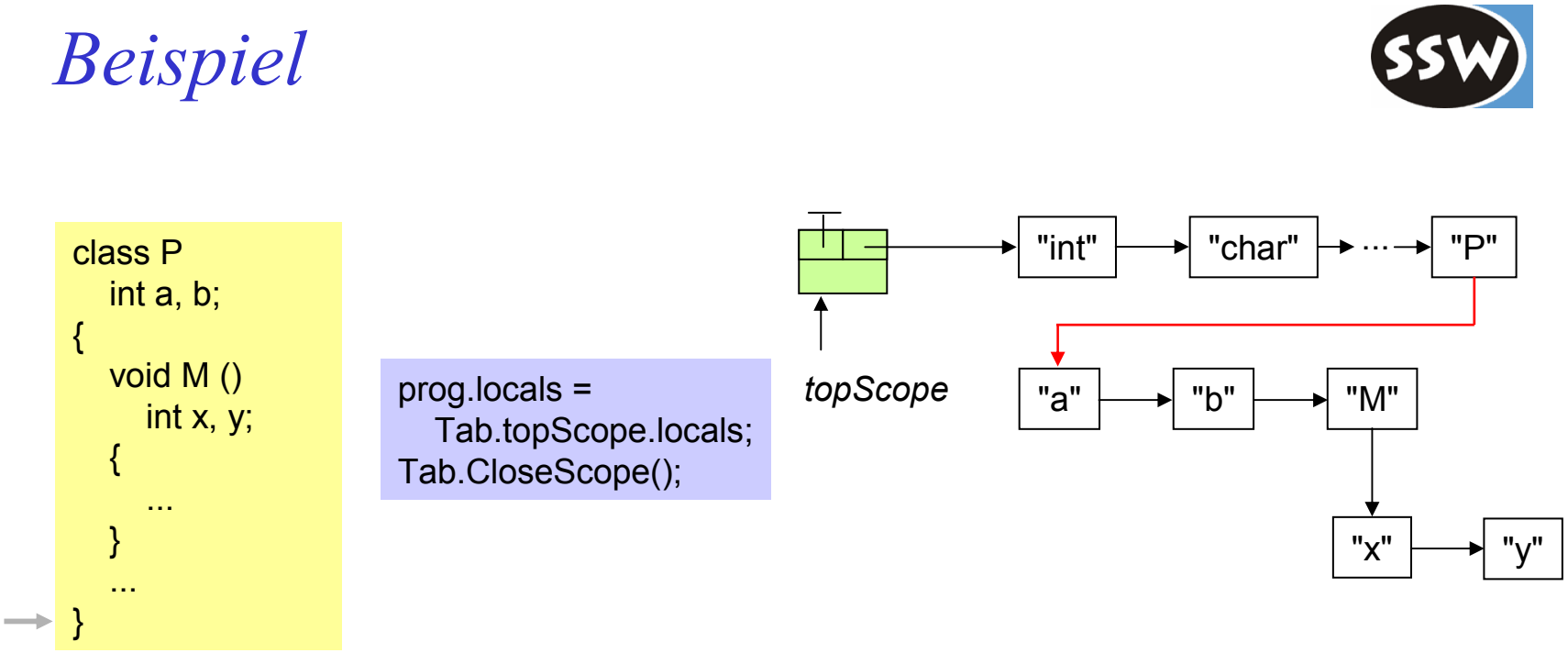
Beispiel


class P
int a, b;
{
void M ()
int x, y;
Tab.Insert(..., "x", ...);
Tab.Insert(..., "y", ...);
"int"
"char"
···
"P"
"a"
"b"
"M"
"x"
"y"
topScope

Beispiel


class P
int a, b;
{
void M ()
int x, y;
{
...
}
meth.locals =
Tab.topScope.locals;
Tab.CloseScope();
"int"
"char"
···
"P"
"a"
"b"
"M"
"x"
"y"
topScope

Beispiel



```
class P
   int a, b;
{
   void M ()
      int x, y;
   {
      ...
   }
   ...
}
```

```
prog.locals =
   Tab.topScope.locals;
Tab.CloseScope();
```

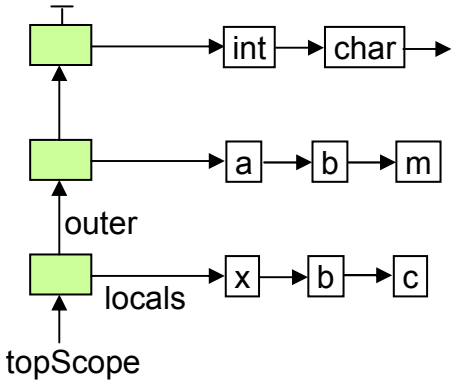
Suchen von Namen in der Symbolliste

Bei jedem Auftreten eines Namens wird folgende Methode aufgerufen

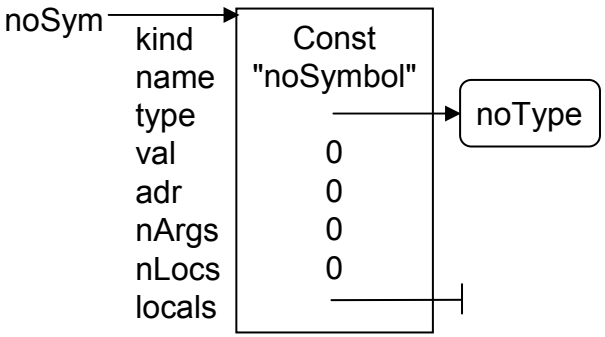
```
Symbol sym = Tab.Find(name);
```

- Suche beginnt in *topScope*
- Falls nicht gefunden, im nächstäußeren Scope weitersuchen

```
static Symbol Find (string name) {
   for (Scope s = topScope; s != null; s = s.outer)
      for (Symbol sym = s.locals; sym != null; sym = sym.next)
         if (sym.name == name) return sym;
   Parser.Error(name + " is undeclared");
   return noSym;
}
```



Falls Name nicht gefunden, *noSym* liefern



- vordefiniertes Dummy-Symbol
- besser als *null*, weil es sonst Folgefehler (Exceptions) gibt

5. Symbolliste

Typen

Symbole haben einen Typ mit folgenden Eigenschaften

- Größe (in .Net in Metadaten verwaltet)
- Struktur (Felder bei Klassen, Elementtyp bei Arrays, ...)

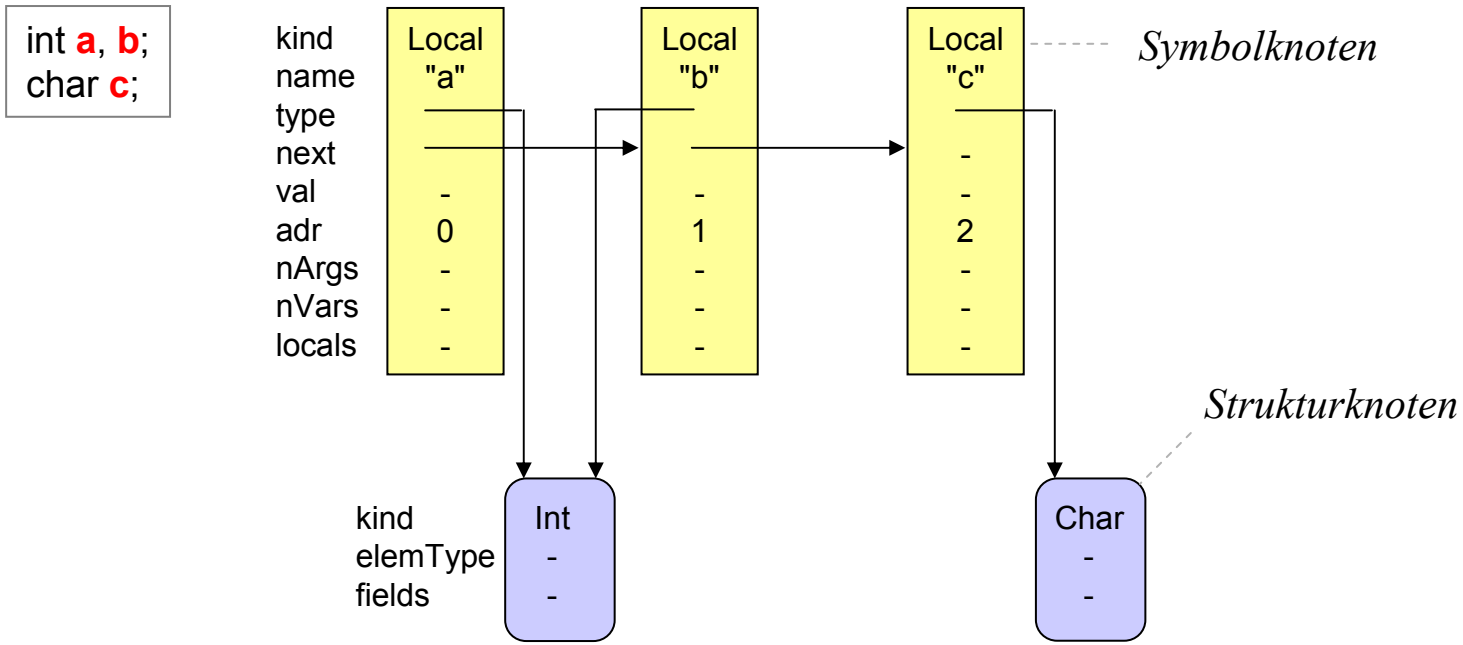
Welche Typarten gibt es in Micro?

- einfache Typen (int, char)

Typen werden durch Strukturknoten dargestellt

```
class Struct {
   public enum Kinds { Int, Char }
   Kinds    kind;
   Type     sysType;      // .Net specific to emit code
}
```

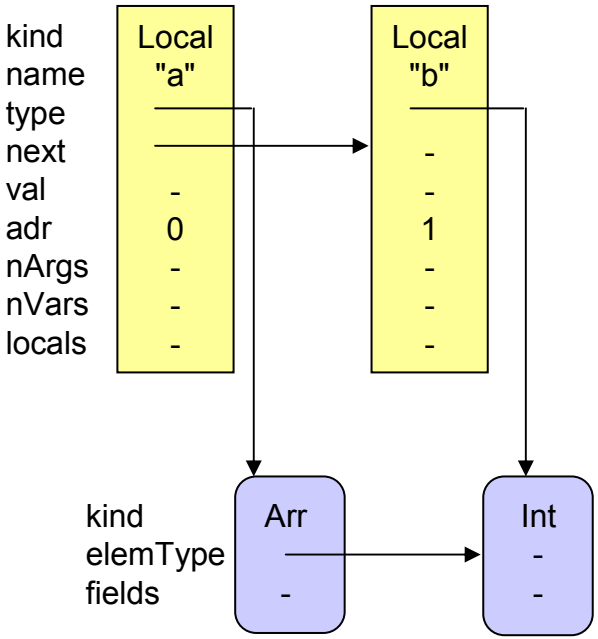
Strukturknoten für einfache Typen



Es gibt in der gesamten Symbolliste nur einen einzigen Strukturknoten für *int*.
Alle Symbole vom Typ *int* verweisen auf ihn.
Dasselbe gilt für den Strukturknoten von *char*.

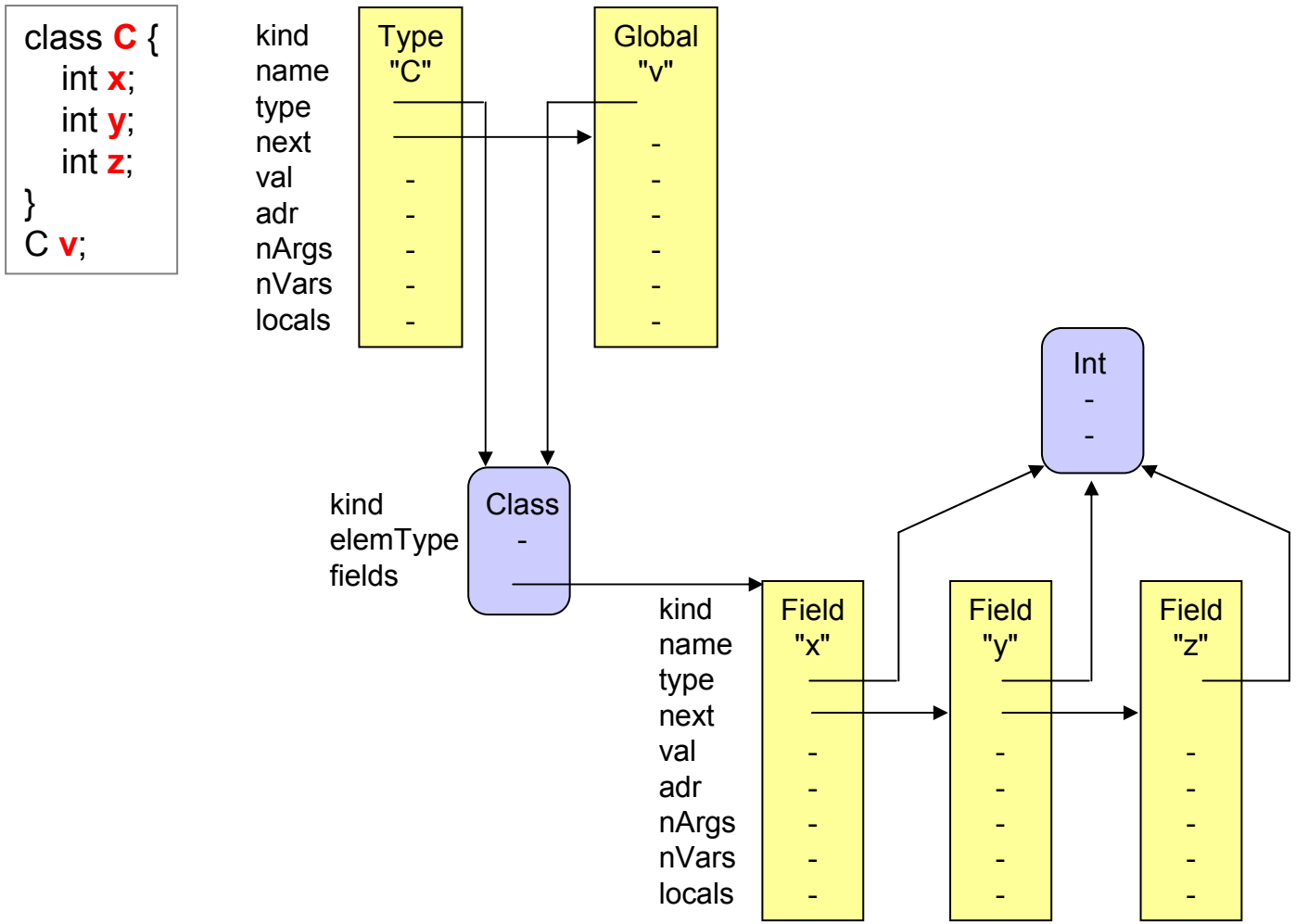
Strukturknoten für Arrays

```
int[] a;
int b;
```



Die Länge eines Arrays ist statisch nicht bekannt.
Sie wird zur Laufzeit im Array-Objekt gespeichert

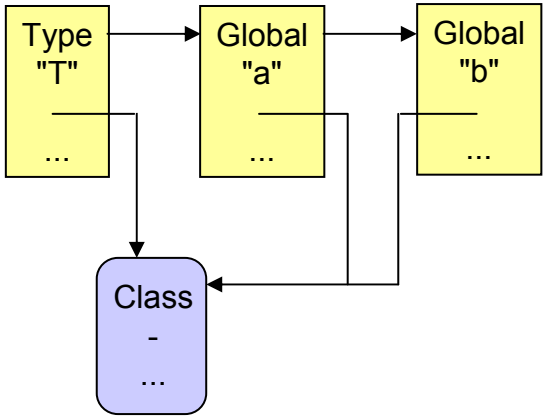
Strukturknoten für Klassen



Typkompatibilität: Namensäquivalenz

Typen sind gleich, wenn sie durch denselben Typknoten dargestellt werden (d.h. wenn sie durch den gleichen Typnamen bezeichnet werden)

```
class T { ... }
T a;
T b;
```



Die Typen von *a* und *b* sind gleich.

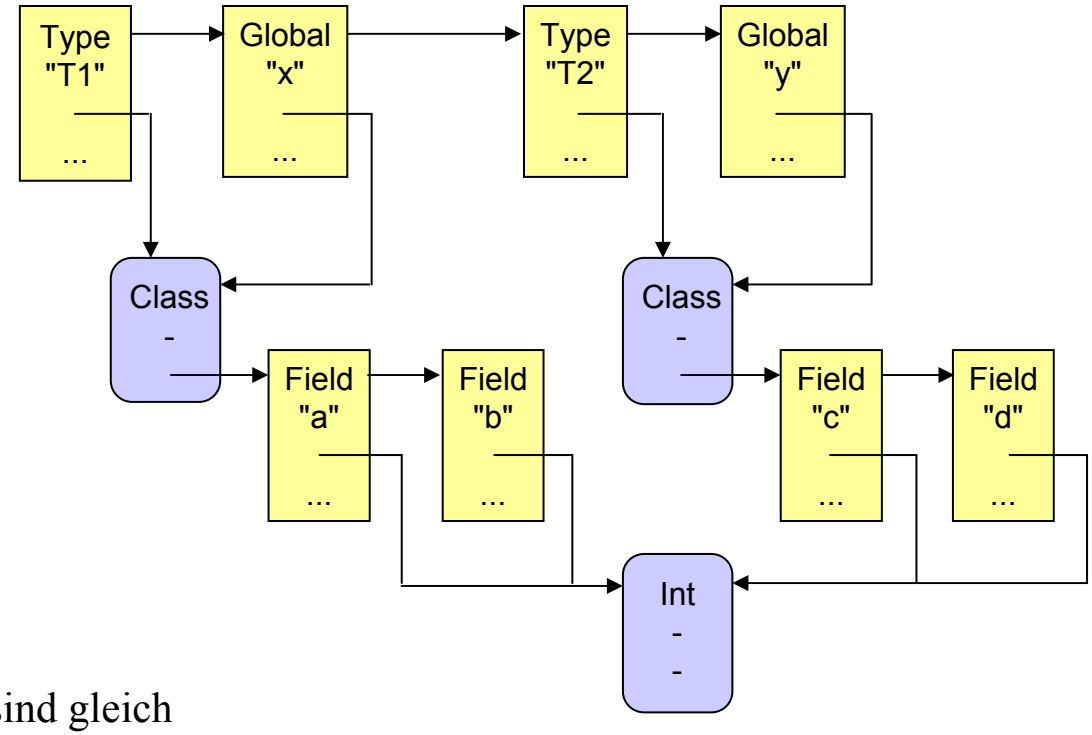
Gilt in Java, C/C++/C#, Pascal, ...

Ausnahme: Arraytypen sind gleich, wenn sie denselben Elementtyp haben!

Typkompatibilität: *Strukturäquivalenz*

Typen sind gleich, wenn sie die gleiche Struktur haben
(d.h. gleiche Felder vom gleichen Typ, gleichen Elementtyp, ...)

```
class T1 { int a, b; }
class T2 { int c, d; }
T1 x;
T2 y;
```



Die Typen von *x* und *y* sind gleich

z.B. in Modula-3

Methoden zur Prüfung der Typkompatibilität


```
class Struct {
   ...
   // prüft, ob zwei Typen kompatibel sind (z.B. in Vergleichen)
   public bool CompatibleWith (Struct other) {
      return  this.Equals(other) ||
              this == Tab.nullType && other.IsRefType() ||
              other == Tab.nullType && this.isRefType();
   }

   // prüft, ob this an dest zuweisbar ist
   public bool AssignableTo (Struct dest) {
      return  this.Equals(dest) ||
              this == Tab.nullType && dest.IsRefType() ||
              kind == Kinds.Arr && dest.kind == Kinds.Arr && dest.elemType == Tab.noType;
   }

   // prüft, ob zwei Typen gleich sind (bei Arrays Strukturäquivalenz, sonst Namensäquivalenz)
   public bool Equals (Struct other) {
      if (kind == Kinds.Arr)
         return  other.kind == Kinds.Arr && elemType.Equals(other.elemType);
      return other == this;
   }

   public bool IsRefType() { return kind == Kinds.Class || kind == Kinds.Arr; }
}
```

nötig wegen Standardfunktion len(arr)

6. Codeerzeugung

Aufgaben der Codeerzeugung

Erzeugung von Maschinenbefehlen

- Auswahl passender Instruktionen
- Auswahl passender Adressierungsarten

Umsetzung von Verzweigungen und Schleifen in Sprünge

Verwalten von Aktivierungssätzen für lokale Variablen

Eventuell Optimierungen

Ausgabe der Objektdatei

Wie geht man vor?

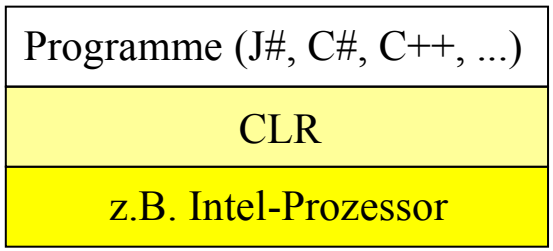
1. **Studieren der Zielmaschine**
   Register, Datenformate, Adressierungsarten, Instruktionen, Befehlsformate, ...
2. **Festlegen von Laufzeitdatenstrukturen**
   Layout von Aktivierungssätzen, globalen Daten, Heapobjekten, Stringkonstantenspeicher, ...
3. **Verwaltung des Codespeichers**
   Bit-Codierung der Befehle, Patchen des Codes, ...
4. **Registerverwaltung**
   entfällt für Micro, da die CLR eine Stackmaschine ist
5. **Implementierung der Codeerzeugungsmethoden** (in folgender Reihenfolge)
   - Laden von Werten und Adressen (in Register oder auf den Stack)
   - Verarbeitung zusammengesetzter Bezeichner (x.y, a[i], ...)
   - Übersetzung von Ausdrücken
   - Verwaltung von Sprüngen und Marken
   - Übersetzung von Anweisungen
   - Übersetzung von Methoden und Parametern

6. Codeerzeugung

Architektur der CLR

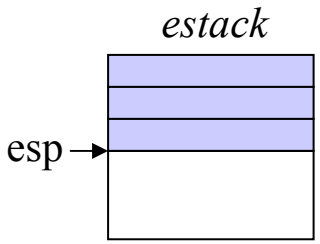
Was ist eine Virtuelle Maschine (VM)?

- Eine in Software implementierte CPU
- Befehle werden interpretiert / JIT-übersetzt
- andere Beispiele: Java-VM, Smalltalk-VM, Pascal P-Code



Die CLR ist eine Stackmaschine

- keine Register
- stattdessen *Expression Stack* (auf den Werte geladen werden)



max. Größe wird für jeder Methode in den Metadaten gespeichert

esp ... expression stack pointer

Die CLR führt JIT-übersetzen Bytecode aus

- jede Methode wird erst direkt vor der ersten Ausführung übersetzt (= just-in-time)
- Operanden werden in IL symbolisch adressiert (aus Informationen in Metadaten)

Arbeitsweise einer Stackmaschine

Beispiel

Anweisung **i = i + j * 5;**

Angenommene Werte von *i* und *j*

locals

0	3	i
1	4	j

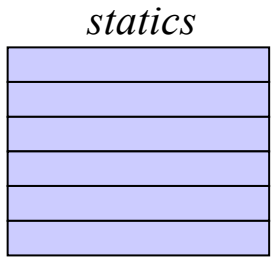
Abarbeitung

Befehle	*Stack*	
ldloc.0	3	lade lokale Variable von Adresse 0 (d.h. i)
ldloc.1	3 4	lade lokale Variable von Adresse 1 (d.h. j)
loc.i4.5	3 4 5	lade Konstante 5
mul	3 20	multipliziere die obersten beiden Stackelemente
add	23	addiere die obersten beiden Stackelemente
stloc.0		speichere oberstes Stackelement auf Adresse 0

Am Ende jeder Anweisung ist der Expression Stack wieder leer!

Datenbereiche der CLR

Globale Variablen



- werden in der CLR zu statischen Klassenfeldern der Programmklasse
- leben während der gesamten Programmausführung (= während Programmklasse geladen ist)
- Adressierung über Metadaten-Tokens
  z.B. ldsfld T_{fld} lädt den Wert des von T_{fld} referenzierten statischen Felds auf den *estack*

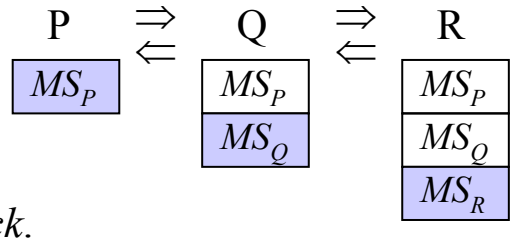
Metadaten-Token sind 4 Byte große Werte, die Zeilen in Metadaten-Tabellen referenzieren.

Token-Type (1 Byte)	Index in Metadaten-Tabelle (3 Byte)

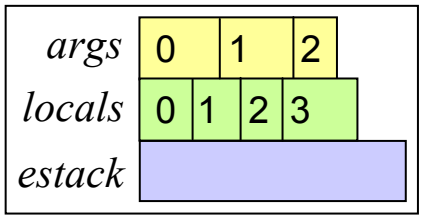
Datenbereiche der CLR

Methodenzustand

- verwaltet eigene Bereiche für
  - Argumente (*args*)
  - lokale Variablen (*locals*)
  - Expression Stack (*estack*)
- jeder Methodenaufruf hat seinen eigenen Methodenzustand (method state, *MS*)
- Methodenzustände werden kellerartig verwaltet. Daher spricht man auch von *Stack Frame* und *Methodenstack*.



- Jeder Parameter und jede lokale Variable belegen ein Fach ("Slot") von typabhängiger Größe.
- Adressen sind fortlaufende Nummern, die der Deklarationsreihenfolge entsprechen
  z.B. *ldarg.0* lädt den Wert des ersten Methodenarguments auf den *estack*
  ldloc.2 lädt den Wert der dritten lokalen Variablen auf den *estack*

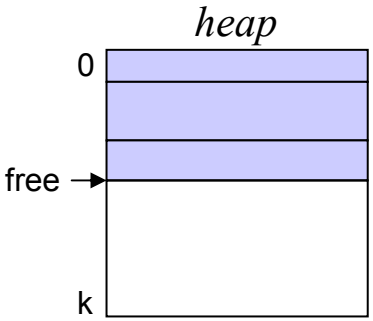


method state

Datenbereiche der CLR

Heap

- enthält Klassen- und Array-Objekte

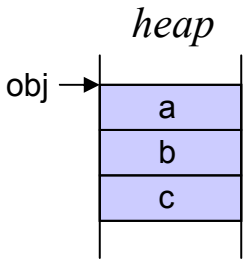


- neue Objekte werden an der Stelle *free* angelegt und *free* wird erhöht; durch die CIL-Befehle *newobj* und *newarr*
- Objekte werden vom Garbage Collector freigegeben

Datenbereiche der CLR

Klassen-Objekte

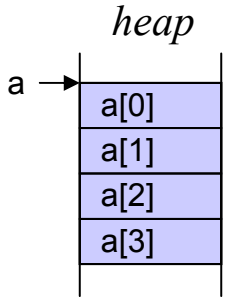
```
class X {
    int a, b;
    char c;
}
X obj = new X();
```



- Adressierung durch Field-Token relativ zu *obj*

Array-Objekte

```
int[] a = new int[4];
```



- Adressierung durch Indexwert relativ zu *a*

- Vektoren können mit den speziellen CIL-Anweisung newarr, ldlen, ldelem, stelem bearbeitet werden.

Codespeicher

- Micro kennt nur statische Methoden, die zum Typ der Hauptklasse gehören
- jede Methode besteht aus einem Strom von CIL-Anweisungen und Metadaten, z.B. *.entrypoint*, *.locals*, *.maxstack*, Zugriffsattribute, ...

System.Reflection.Emit.ILGenerator verwaltet den CIL-Strom.

```
class ILGenerator {
   /* überladen (siehe nächste Folie) */
   virtual void Emit (OpCode op, ...);
   /* für Methodenaufrufe */
   void EmitCall (
      OpCode op,
      MethodInfo meth,
      Type[] varArgs);
   ...
}
```

Befehlscodes sind in Klasse *OpCodes* deklariert.

z.B.: Ausgabe von *ldloc.2*

```
il.Emit(OpCodes.Ldloc2);
```

Emit-Methoden von ILGenerator


```
class ILGenerator {                          // z.B.: für folgende IL-Anweisungen:
   void Emit (OpCode);                       // ldarg.n, ldloc.n, stloc.n, ldnull, ldc.i4.n, ld.i4.m1
                                             // add, sub, mul, div, rem, neg,
                                             // ldlen, ldelem... , stelem... , dup, pop, ret, throw
   void Emit (OpCode, byte);                 // ldarg.s, starg.s, ldloc.s, stloc.s
   void Emit (OpCode, int);                  // ldc.i4
   void Emit (OpCode, FieldInfo);            // ldsfld, stsfld, ldfld, stfld
   void Emit (OpCode, LocalBuilder);         // ldloc.s, stloc.s
   void Emit (OpCode, ConstructorInfo);      // newobj
   void Emit (OpCode, Type);                 // newarr
   void Emit (OpCode, Label);                // br, beq, bge, bgt, ble, blt, bne.un

   /* für Methodenaufrufe */
   void EmitCall (OpCode, MethodInfo, Type[]);  // call (letztes Argument (varargs) auf null setzen)
   ...
}
```

Metadaten

beschreiben Eigenschaft der Elemente des Assemblies (Typen, Felder, Methoden, ...).
Klasse *CodeGen* hat Felder zur Verwaltung wichtiger Metadaten-Elemente:

```
AssemblyBuilder assembly;      // the program assembly
ModuleBuilder module;          // the program module
TypeBuilder program;           // the main class
```

Die Methode *CreateMetadata* der Klasse *Code* erzeugt Metadatenobjekte aus Symbolknoten:

```
public void CreateMetadata (ISymbol sym) {
   if (SymbolKind.Local == sym.Kind) {
      il.DeclareLocal(sym.Type.SysType);
   }
   else if (SymbolKind.Func == sym.Kind) {
      // build argument list
      Type[] args = new Type[sym.NArgs]; ISymbol arg = sym.Locals;
      while (arg != null && arg.Kind == SymbolKind.Arg) {
         args[arg.Adr] = arg.Type.SysType; arg = arg.Next;
      }
      sym.Meth = program.DefineMethod(sym.Name, MethodAttributes.Public | MethodAttributes.Static,
            sym.Type.SysType, args);
      il = sym.Meth.GetILGenerator();
      if ("main".Equals(sym.Name)) { assembly.SetEntryPoint(sym.Meth); }
   }
}
```

Instruktionssatz der CLR

Common Intermediate Language (CIL) (hier nur eine Untermenge notwendig)

- sehr kompakt: die meisten Befehle sind nur 1 Byte lang
- meist ungetypt; Typangaben in den Befehlen beziehen sich of auf Ergebnistyp

ungetypt

```
ldloc.0
starg.1
add
```

getypt

```
ldc.i4.3
ldelem.i2
stelem.ref
```

Befehlsformat

Sehr einfach im Vergleich zu Intel, PowerPC oder SPARC

```
Code        = { Instruction }.
Instruction = opcode [ operand ].
```

opcode ... 1 oder 2 Byte
operand ... primitiver Datentyp oder Metadaten-Token

Beispiele

0 Operanden	`add`	hat zwei implizite Operanden am Stack
1 Operand	`ldc.i4.s 9`	

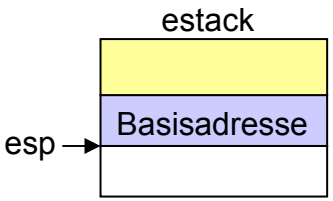
Instruktionssatz der CLR

Adressierungsarten

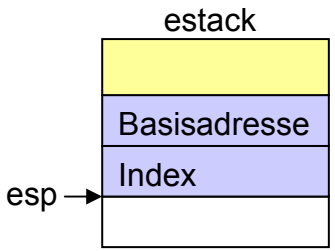
Wie kann man Operanden in Befehlen ansprechen?

Adressierungsart	*Beispiel*	
• **Immediate**	ldc.i4 123	für Konstanten
• **Arg**	ldarg.s 5	für Methodenargumente
• **Local**	ldloc.s 12	für lokale Variablen
• **Static**	ldsfld *fld*	für statische Felder (*fld* = Metadaten-Token)
• **Stack**	add	für geladene Werte am *estack*
• **Relative**	ldfld *fld*	für Objektfelder (Objektzeiger liegt am *estack*)
• **Indexed**	ldelem.i4	für Arrayelemente (Arrayzeiger und Index liegen am *estack*)

Relative



Indexed



Instruktionssatz der CLR

Laden und Speichern von Methodenargumenten

ldarg.s b	... ..., val	Load push(args[b]);
ldarg.*n*	... ..., val	Load (n = 0..3) push(args[n]);
starg.s b	..., val ...	Store args[b] = pop();

Laden und Speichern lokaler Variablen

ldloc.s b	... ..., val	Load push(locals[b]);
ldloc.*n*	... ..., val	Load (n = 0..3) push(locals[n]);
stloc.s b	..., val ...	Store locals[b] = pop();
stloc.*n*	..., val ...	Store (n = 0..3) locals[n] = pop();

Operandentypen

b ... unsigned byte
i ... signed integer
T ... Metadaten-Token

Instruktionssatz der CLR

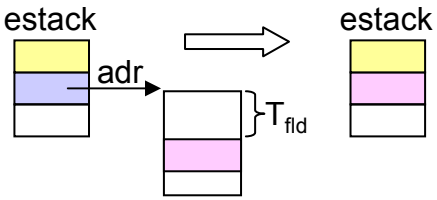
Laden und Speichern globaler Variablen

ldsfld	T_{fld}	... ..., val	Load static variable push(statics[T_{fld}]);
stsfld	T_{fld}	..., val ...	Store static variable statics[T_{fld}] = pop();

Laden und Speichern von Objektfeldern

ldfld	T_{fld}	..., obj ..., val	Load object field obj = pop(); push(heap[obj+T_{fld}]);

stfld	T_{fld}	..., obj, val
		...
			Store object field
			val = pop(); obj = pop();
			heap[obj+T_{fld}] = val;

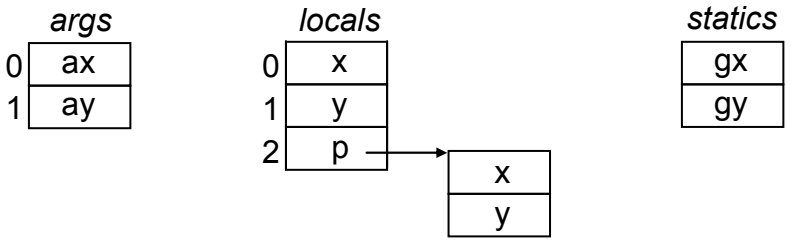


Instruktionssatz der CLR

Laden von Konstanten

ldc.i4 i	... ..., i	Load constant push(i);
ldc.i4.*n*	... ..., *n*	Load constant (n = 0..8) push(n);
ldc.i4.m1	... ..., -1	Load minus one push(-1);
ldnull	... ..., null	Load null push(null);

Beispiele: Laden und Speichern



	Code	*Bytes*	*estack*
ax = ay;	ldarg.1	1	ay
	starg.s 0	2	-
x = y;	ldloc.1	1	y
	stloc.0	1	-
gx = gy;	ldsfld T_{fld_gy}	5	gy
	stsfld T_{fld_gx}	5	-
p.x = p.y;	ldloc.2	1	p
	ldloc.2	1	p p
	ldfld T_{fld_y}	5	p p.y
	stfld T_{fld_x}	5	-

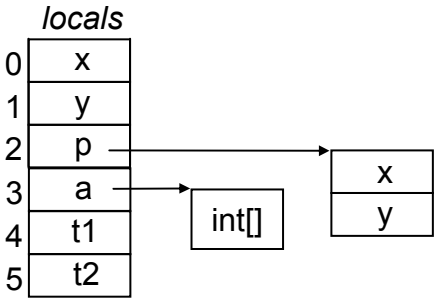
Instruktionssatz der CLR

Arithmetik

add	..., val1, val2 ..., val1+val2	Add push(pop() + pop());
sub	..., val1, val2 ..., val1-val2	Subtract push(-pop() + pop());
mul	..., val1, val2 ..., val1*val2	Multiply push(pop() * pop());
div	..., val1, val2 ..., val1/val2	Divide x = pop(); push(pop() / x);
rem	..., val1, val2 ..., val1%val2	Remainder x = pop(); push(pop() % x);
neg	..., val ..., -val	Negate push(-pop());

Beispiele: Arithmetik

	Code	*Bytes*	*Stack*
x + y * 3	ldloc.0	1	x
	ldloc.1	1	x y
	ldc.i4.3	1	x y 3
	mul	1	x y*3
	add	1	x+y*3



x++;	ldloc.0	1	x
	ldc.i4.1	1	x 1
	add	1	x+1
	stloc.0	1	-
p.x++	ldloc.2	1	p
	dup	1	p p
	ldfld T_{px}	5	p p.x
	ldc.i4.1	1	p p.x 1
	add	1	p p.x+1
	stfld T_{px}	5	-

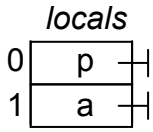
x--;	ldloc.0	1	x
	ldc.i4.m1	1	x -1
	add	1	x-1
	stloc.0	1	-
a[2]++	ldloc.3	1	a
	ldc.i4.2	1	a 2
	stloc.s 5	2	a
	stloc.s 4	2	-
	ldloc.s 4	2	a
	ldloc.s 5	2	a 2
	ldloc.s 4	2	a 2 a
	ldloc.s 5	2	a 2 a 2
	ldelem.i4	1	a 2 a[2]
	ldc.i4.1	1	a 2 a[2] 1
	add	1	a 2 a[2]+1
	stelem.i4	1	-

Instruktionssatz der CLR

Objekterzeugung

newobj T_{ctor}	... [arg0, ..., argN] ..., obj	New object erzeugt neues Objekt vom durch Constructor-Token angegebenen Typ und führt dann den Konstruktor aus (Argumente am Stack)
newarr T_{eType}	..., n ..., arr	New array erzeugt Array mit Platz für n Elemente vom durch Type-Token angegebenen Typ

Beispiele: Objekterzeugung



	Code	*Bytes*	*Stack*
Person p = new Person();	newobj $T_{P()}$	5	p
	stloc.0	1	-
int[] a = new int[5];	ldc.i4.5	1	5
	newarr T_{int}	5	a
	stloc.1	1	-

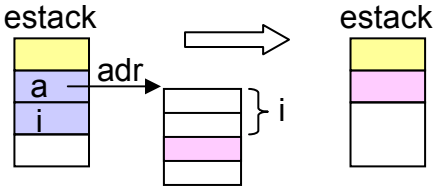
Instruktionssatz der CLR

Arrayzugriff

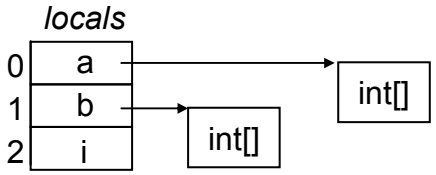
ldelem.u2
ldelem.i4
ldelem.ref
	..., adr, i
	..., val
		Load array element
		i = pop(); adr = pop();
		push(heap[adr+i]);
		Typ des Ergebnisses am Stack:
		char, int, Objektreferenz

stelem.i2
stelem.i4
stelem.ref
	...,adr, i, val
	...
		Store array element
		val=pop(); i=pop(); adr=pop()/4+1;
		heap[adr+i] = val;
		Typ des zu speichernden Elements:
		char, int, Objektreferenz

ldlen	..., adr ..., len	Get array length



Beispiel: Arrayzugriff



	Code	*Bytes*	*Stack*
a[i] = b[i+1];	ldloc.0	1	a
	ldloc.2	1	a i
	ldloc.1	1	a i b
	ldloc.2	1	a i b i
	ldc.i4.1	1	a i b i 1
	add	1	a i b i+1
	ldelem.i4	1	a i b[i+1]
	stelem.i4	1	-

Instruktionssatz der CLR

Stackmanipulation

pop	..., val ...	Remove topmost stack element dummy = pop();
dup	..., val ..., val, val	Duplicate topmost stack element x = pop(); push(x); push(x);

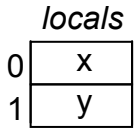
Sprünge

br i	... ...	Branch unconditionally pc = pc + i

b<*cond*> i	..., x, y
	...
		Branch conditionally
		(<*cond*> = eq | ge | gt | le | lt | ne.un)
		y = pop(); x = pop();
		if (x cond y) pc = pc + i;

pc markiert aktuelle Instruktion;
i (Sprungdistanz) relativ zum Beginn der nächsten Instruktion

Beispiel: Sprünge



	Code	*Bytes*	*Stack*
if (x > y) {	ldloc.0	1	x
...	ldloc.1	1	x y
}	ble ...	5	-
...			

Instruktionssatz der CLR

Methodenaufruf

call T_{meth}	... [arg0, ... argN] ... [retVal]	Call method nimmt Argumente von *estack* des Rufers und gibt sie an *args* des Gerufenen; nimmt Rückgabewert von *estack* des Gerufenen und legt ihn auf *estack* des Rufers
ret	... ...	Return from method

Sonstiges

throw	..., exc ...	Throw exception

Instruktionssatz der CLR

Ein-/Ausgabe

Die Funktionalität der vordefinierten Funtionen *put_i, put_c* und *get_i, get_c* ist in eigenen Methoden der Hauptklasse implementiert. Diese verwenden für die Ein-/Ausgabe die Klasse System.Console.

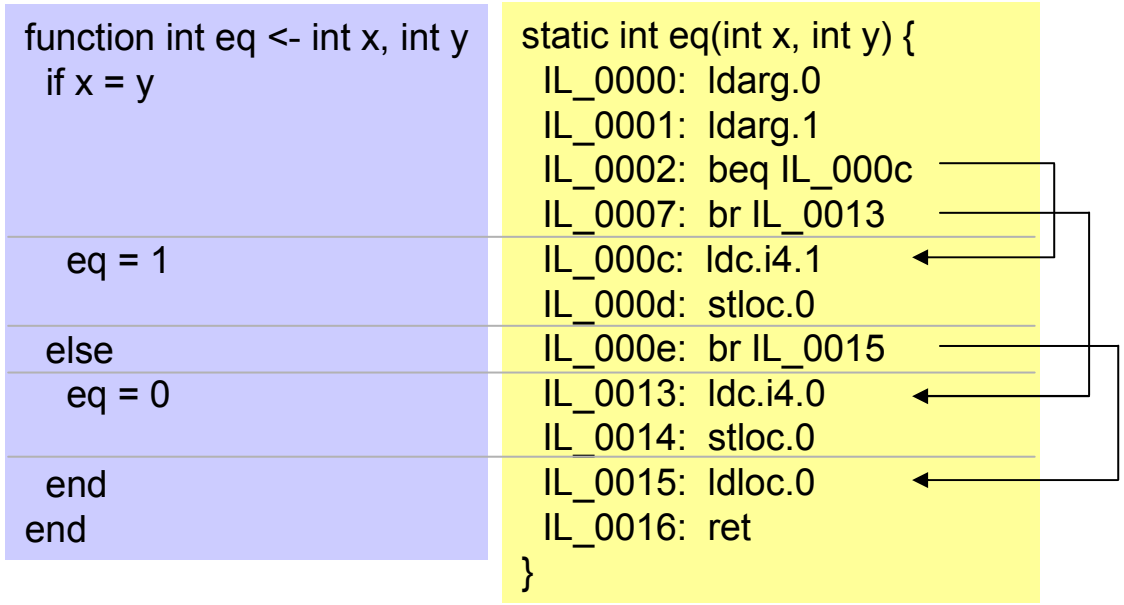
```
static int put_i(int i);
static int put_c(int i);
static int get_i();
static char get_c();
```

Diese Funktionen werden vom Compiler zur Hauptklasse hinzugefügt.
Und können im Programm verwendet werden.

```
IStruct intType = symTab.FindType("int").Type;
IStruct charType = symTab.FindType("char").Type;
ISymbol putiSym = symTab.Insert(SymbolKind.Func,
      "put_i", intType);
symTab.OpenScope();
symTab.Insert(SymbolKind.Arg, "i", intType);
symTab.SetLocals(putiSym);
BuildPutI(putiSym);
symTab.CloseScope();
…
```

```
private void BuildPutI(ISymbol puti) {
   // static int put_i (int x)
   codeGen.CreateMetadata(puti);
   codeGen.Emit(OpCodes.Ldarg_0);
   MethodInfo writeLine =
      typeof(Console).GetMethod("WriteLine",
      new Type[] {typeof (int)}, null);
   codeGen.EmitCall(writeLine);
   // return 0;
   codeGen.Emit(OpCodes.Ldc_I4_0);
   // }
   codeGen.Emit(OpCodes.Ret);
}
```

Beispiel



```
function int eq <- int x, int y
  if x = y
    eq = 1
  else
    eq = 0
  end
end
```

```
static int eq(int x, int y) {
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  beq IL_000c
  IL_0007:  br IL_0013
  IL_000c:  ldc.i4.1
  IL_000d:  stloc.0
  IL_000e:  br IL_0015
  IL_0013:  ldc.i4.0
  IL_0014:  stloc.0
  IL_0015:  ldloc.0
  IL_0016:  ret
}
```

Adressen

Argumente

x ... 0

y ... 1

Lokale Variablen

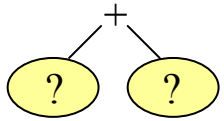
mul ... 0

6. Codeerzeugung

Operanden der Codeerzeugung

Beispiel

Es sollen zwei Werte addiert werden



Gewünschtes Codemuster

Lade Operand 1
Lade Operand 2
add

Je nach Operandenart müssen andere Lade-Instruktionen erzeugt werden

Operandenart	*Was muss erzeugt werden?*
• Konstante	ldc.i4 x
• Methodenargument	ldarg.s a
• lokale Variable	ldloc.s a
• globale Variable	ldsfld T_{fld}
• Objektfeld	ldfld T_{fld}
• Arrayelement	ldelem
• geladener Wert am Stack	---

Wir brauchen einen Deskriptor, der uns die Art des Operanden beschreibt.

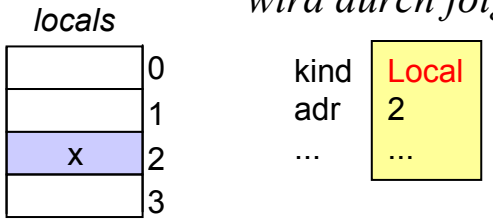
Items

Deskriptoren, die die Art und den Speicherort von Operanden beschreiben

Beispiel

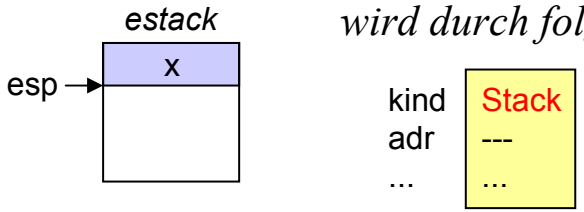
Lokale Variable *x* im *locals*-Bereich des Methodenzustands

wird durch folgendes Item beschrieben



Nach dem Laden mittels *ldloc.2* steht der Wert nun am *estack*

wird durch folgendes Item beschrieben



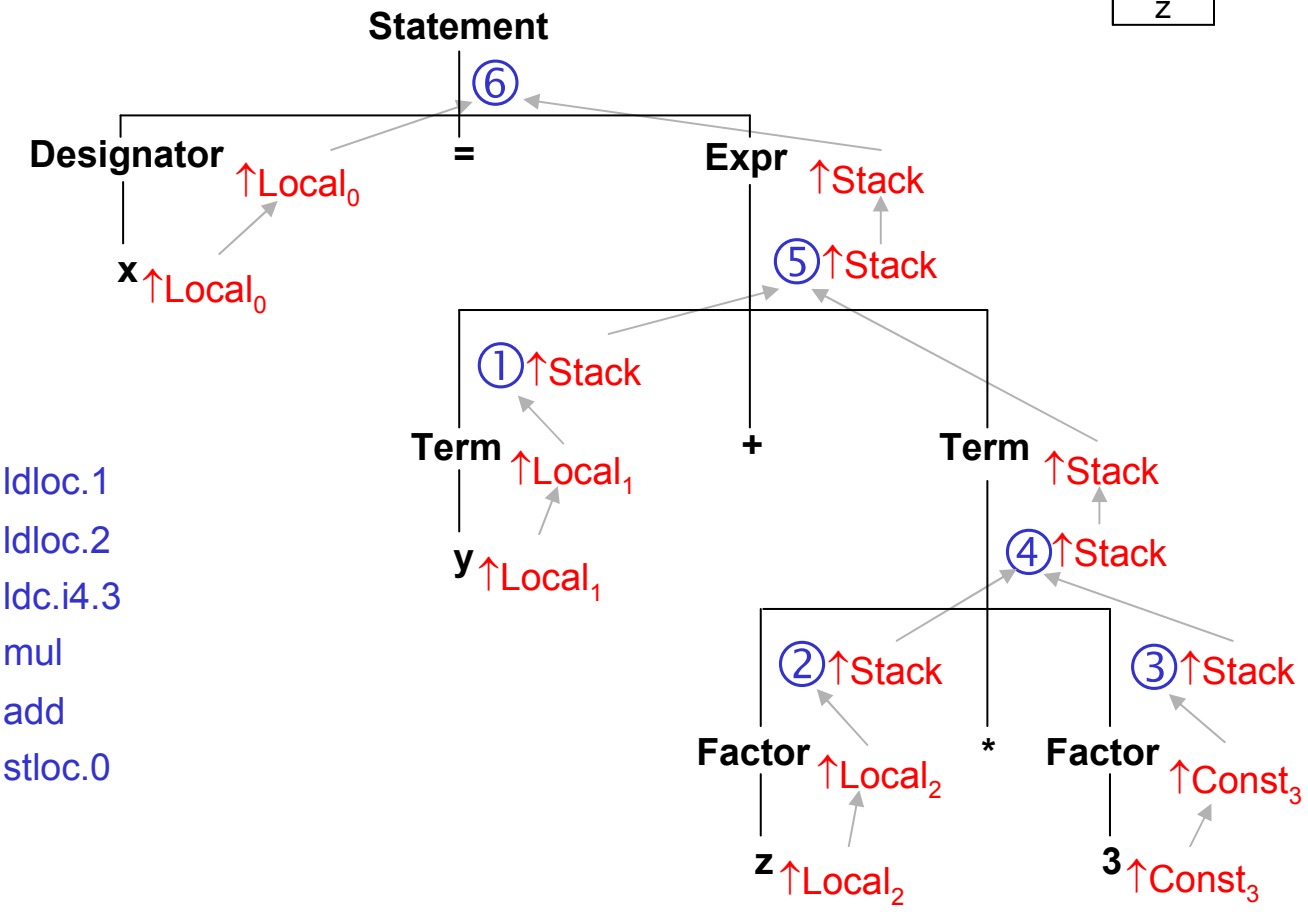
Beispiel: Entstehung von Items

Die meisten Parsermethoden liefern Items (als Ergebnis ihrer Übersetzung)

Beispiel: Übersetzung der Zuweisung `x = y + z * 3;`

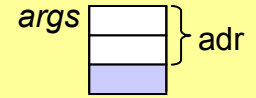
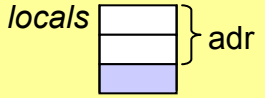
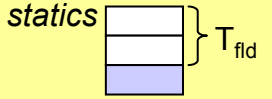
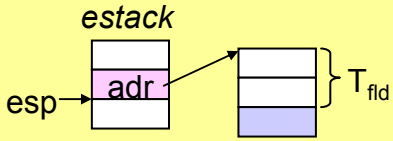
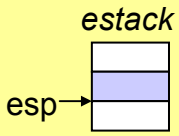
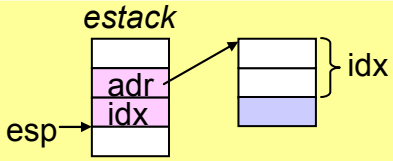
locals

x
y
z



① ldloc.1
② ldloc.2
③ ldc.i4.3
④ mul
⑤ add
⑥ stloc.0

Arten von Items

Operandenart	*Itemart*	*nötige Infos über Operand*	
Konstante	**Const**	Konstantenwert	
Argument	**Arg**	Adresse	
lokale Variable	**Local**	Adresse	
globale Variable	**Static**	Field-Symolknoten	
Objektfeld	**Field**	Field-Symbolknoten	
Wert am Stack	**Stack**	---	
Arrayelement	**Elem**	---	
Methode	**Meth**	Methoden-Symbolknoten	

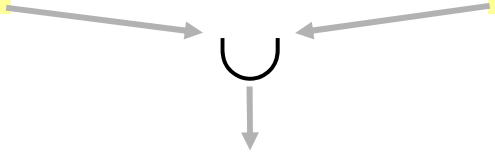
Wie findet man die nötigen Itemarten?

Adressierungsarten
abhängig von Zielmaschine

- Immediate
- Arg
- Local
- Static
- Stack
- Relative
- Indexed

Objektarten
abhängig von Quellsprache

- Const
- Var
- Type
- Meth



Itemarten

- Const
- Arg
- Local
- Static
- Field
- Stack
- Elem
- Meth

Type-Items braucht man in Z# nicht,
da Typen nicht als Operanden vorkommen
können (z.B. keine Type Casts)

Interface IItem


```
public interface IItem {
   ItemKind Kind { get; set; } // Const, Arg, Local, Stack, Func, Cond
   IStruct Type { get; } // item type
   int Val { get; } // Const: value
   int Adr { get; } // Arg, Local: offset
   TokenKind Relop { get; } // Cond: token code of relational operator
   ISymbol Sym { get; } // Field, Func: node from symbol table
   Label Label { get; } // Cond: target
}
```

Erzeugen von Items

Erzeugung aus einem Symbollistenknoten

```
public IItem CreateItem(ISymbol sym)
   Item item = new Item(); item.Type = sym.Type; item.Sym = sym;
   switch (sym.Kind) {
      case SymbolKind.Arg: item.Kind = ItemKind.Arg; item.Adr = sym.Adr; break;
      case SymbolKind.Local: item.Kind = ItemKind.Local; item.Adr = sym.Adr; break;
      case SymbolKind.Func: item.Kind = ItemKind.Func; break;
      case SymbolKind.Const: item.Kind = ItemKind.Const; item.Val = sym.Val; break;
      default: fError.Error("Cannot create code item for this kind of symbol table object"); break;
   }
   return item;
}
```

Erzeugung aus einer Konstanten

```
public IItem CreateItem(int val, IStruct type) {
   Item item = new Item(); item.Kind = ItemKind.Const; item.Val = val;
   item.Type = type; return item;
}
```

Laden von Werten (1)

geg.: Ein Wert, der durch ein Item beschrieben wird (Const, Arg, Local, Static, ...)
ges.: Lade den Wert auf den Stack

```
public void Load(IItem x) {
   switch (x.Kind) {
      case ItemKind.Const: LoadConst(x.Val); break;
      case ItemKind.Arg:
         switch (x.Adr) {
               case 0:il.Emit(OpCodes.Ldarg_0); break;
               case 1:il.Emit(OpCodes.Ldarg_1); break;
               case 2:il.Emit(OpCodes.Ldarg_2); break;
               case 3:il.Emit(OpCodes.Ldarg_3); break;
               default:il.Emit(OpCodes.Ldarg, x.Adr); break;
         } break;
      case ItemKind.Local:
         switch (x.Adr) {
               case 0: il.Emit(OpCodes.Ldloc_0); break;
               case 1: il.Emit(OpCodes.Ldloc_1); break;
               case 2: il.Emit(OpCodes.Ldloc_2); break;
               case 3: il.Emit(OpCodes.Ldloc_3); break;
               default: il.Emit(OpCodes.Ldloc, x.Adr); break;
         } break;
      case ItemKind.Stack: break; // nothing to do (already loaded)
      default:
         fError.Error("Compiler error in Code.load, unexpected item kind");
         break;
   }
   x.Kind = ItemKind.Stack;
}
```

Fallunterscheidungen

Je nach Itemart muss ein anderer Ladebefehl erzeugt werden

Ergebnis ist immer ein *Stack*-Item

Laden von Werten (2)

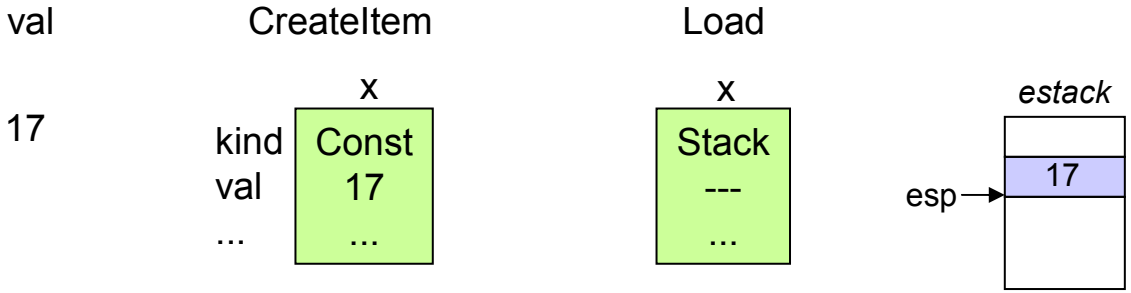

```
private void LoadConst(int n) {
   switch (n) {
      case -1:il.Emit(OpCodes.Ldc_I4_M1);break;
      case 0:il.Emit(OpCodes.Ldc_I4_0);break;
      case 1:il.Emit(OpCodes.Ldc_I4_1);break;
      case 2:il.Emit(OpCodes.Ldc_I4_2);break;
      case 3:il.Emit(OpCodes.Ldc_I4_3);break;
      case 4:il.Emit(OpCodes.Ldc_I4_4);break;
      case 5:il.Emit(OpCodes.Ldc_I4_5);break;
      case 6:il.Emit(OpCodes.Ldc_I4_6);break;
      case 7:il.Emit(OpCodes.Ldc_I4_7);break;
      case 8:il.Emit(OpCodes.Ldc_I4_8);break;
      default:il.Emit(OpCodes.Ldc_I4, n);break;
   }
}
```

Beispiel: Laden einer Konstanten

Beschreibung durch eine ATG

```
Factor<↑IItem x>
=  number          (. // x.kind = Const
                      x = codeGen.CreateItem(token.val, symTab.FindType("int").Type);
                      // x.kind = Stack
                      codeGen.Load(x);
                   .)
```

Visualisierung



Ausgabe

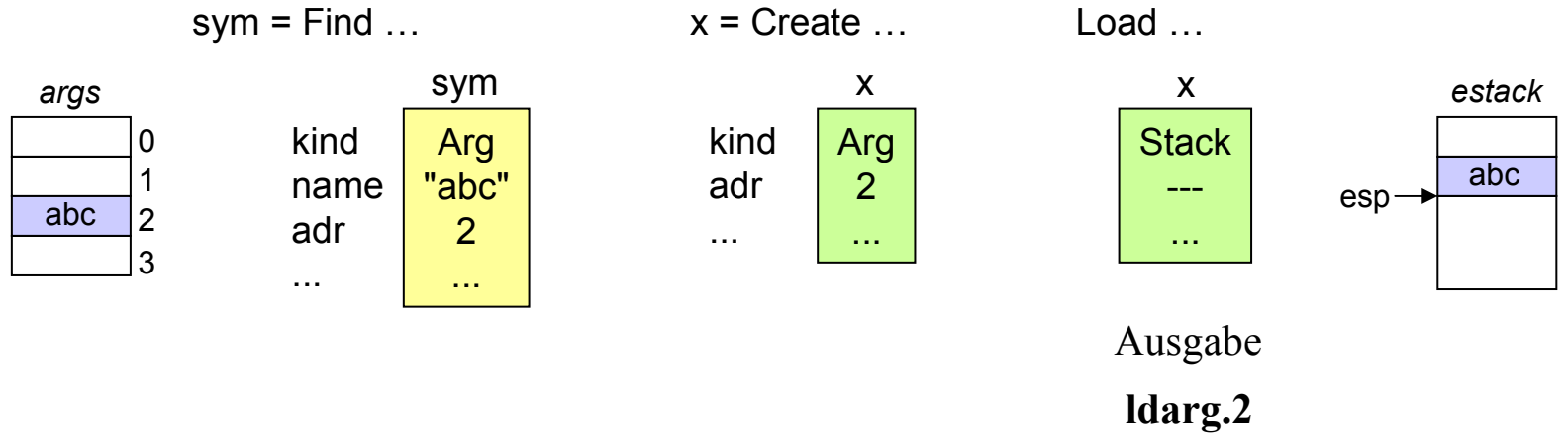
ldc.i4 17

Beispiel: Laden einer Variablen

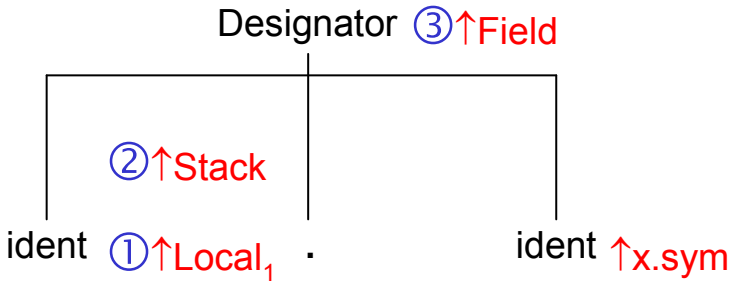
Beschreibung durch eine ATG

```
ident                (. ISymbol sym = symTab.FindVar(token.Str); // sym.kind = Const | Arg | Local
                        IItem x = codeGen.CreateItem (sym);  // x.kind = Const | Arg | Local
                        codeGen.Load(x);                     // x.kind = Stack
                     .) .
```

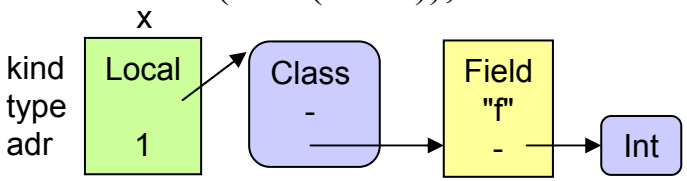
Visualisierung

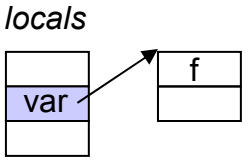


Beispiel: Laden eines Objektfeldes

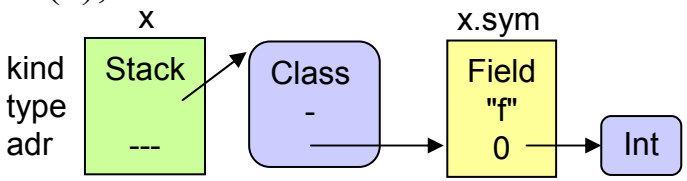


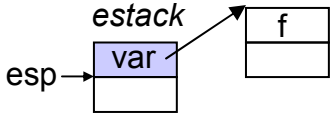
① x = CreateItem(Find(name));





② Load(x);

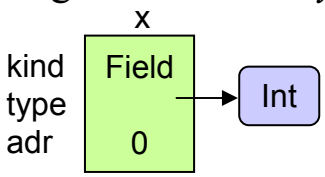


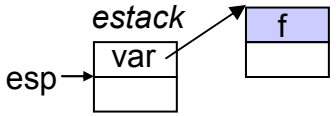


Ausgabe

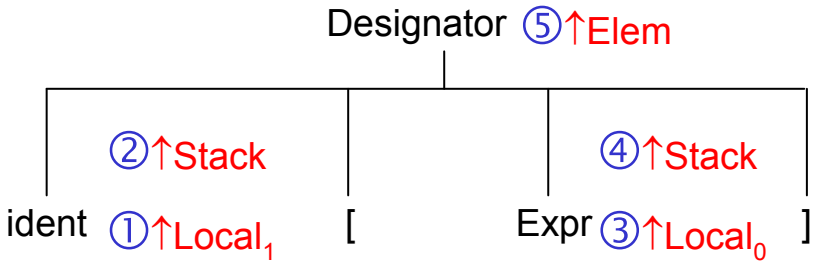
ldloc.1

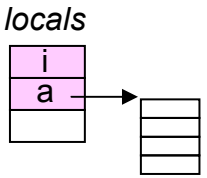
③ erzeuge aus *x* und *x.sym* ein *Field*-Item



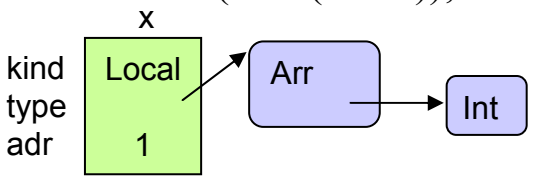


Beispiel: Laden eines Arrayelements

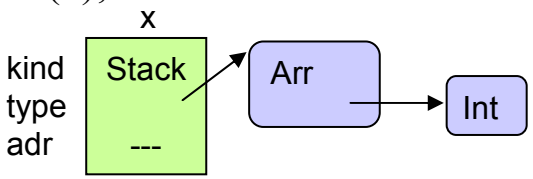




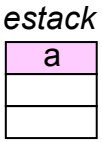
① x = CreateItem(Find(name));



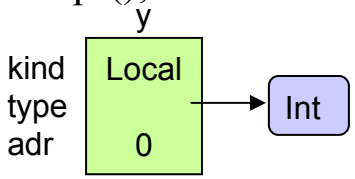
② Load(x);



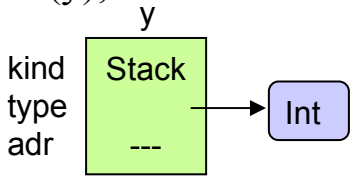
ldloc.1



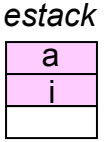
③ y = Expr();



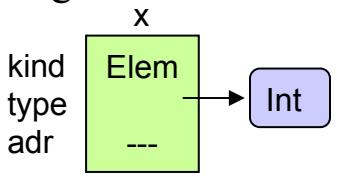
④ Load(y);



ldloc.0



⑤ erzeuge aus *x* ein *Elem*-Item



6. Codeerzeugung

Übersetzung von Ausdrücken

Gewünschtes Schema für x + y + z

```
lade x
lade y
add
lade z
add
```

Kontextbedingungen

Expr = Expr Addop Term.

- *Expr* und *Term* müssen vom gleichen Typ sein.
- Additionsoperation muss auf dem Typ definiert sein (kein Problem in Micro).

Beschreibung durch eine ATG

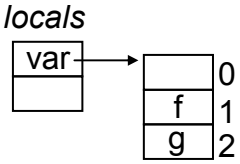
```
Expr<↑IItem x>          (.  IItem y; OpCode op; .)
=
   Term<↑x>
   {  ( "+"             (. op = OpCodes.Add; .)
      | "-"             (. op = OpCodes.Sub; .)
      )                 (.  codeGen.Load(x); .)
      Term<↑y>          (.  codeGen.Load(y);
                            if (!x.Type.Equals(y.Type)) Error("Incompatible types");
                            Code.il.Emit(op);
                        .)
   }
.
```

Übersetzung von Term

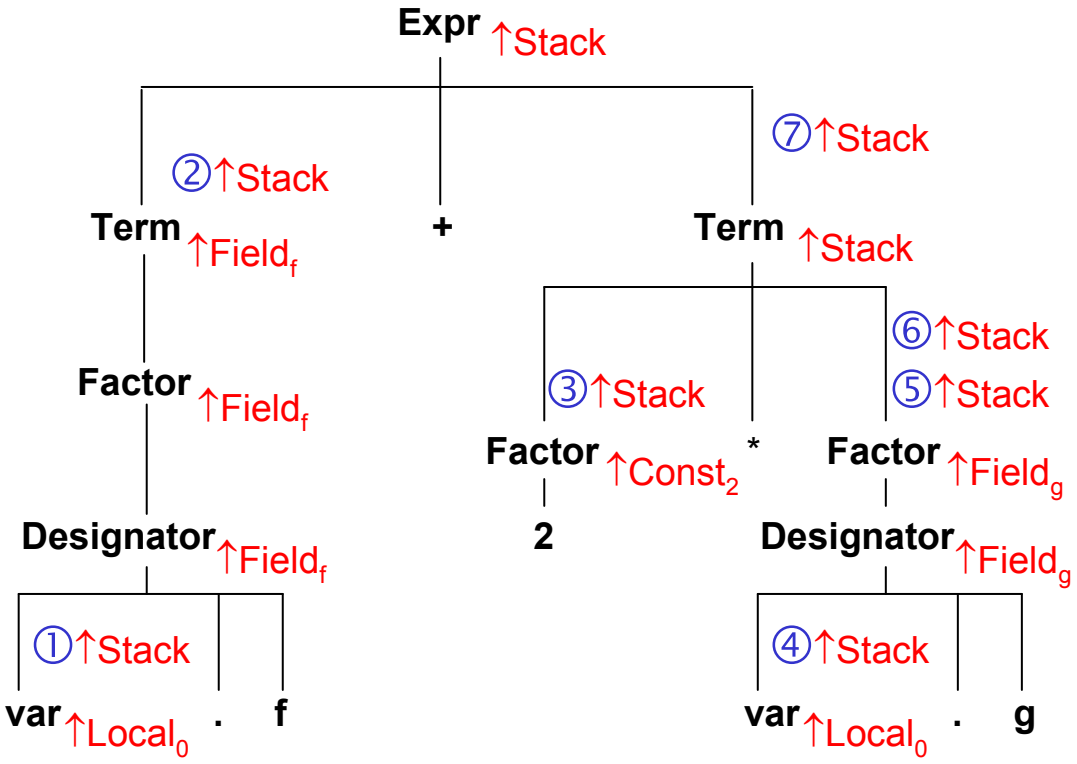

```
Term<↑IItem x>
=  ident                  (. x = create item from symbol … .)
|  numberLit              (.  x = create constant item … .)
|  charLit                (.  x = create constant char item … .)
|  Call< ↑x>
.
```

Beispiel

var.f + 2 * var.g



① ldloc.0
② ldfld T_{fld_f}
③ ldc.i4.2
④ ldloc.0
⑤ ldfld T_{fld_g}
⑥ mul
⑦ add



6. Codeerzeugung

Codemuster für Zuweisungen

5 Fälle, je nach Art der linken Seite

Die blauen Instruktionen werden bereits von *Designator* erzeugt!

arg = expr;	localVar = expr;	globalVar = expr;	obj.f = expr;	a[i] = expr;
... load expr ... **starg arg**	*... load expr ...* **stloc localVar**	*... load expr ...* **stsfld $T_{globalVar}$**	**ldloc obj** *... load expr ...* **stfld T_f**	**ldloc a** **ldloc i** *... load expr ...* **stelem.i2** **i4** **ref**

abhängig vom Elementtyp (char, int, Objektreference)

Übersetzung von Zuweisungen

Kontextbedingung

Statement = ident "=" Expr ";".

- *ident* muss eine Variable bezeichnen.
- Der Typ von *Expr* muss mit dem Typ von *ident* zuweisungskompatibel sein.

Beschreibung durch eine ATG

```
Assignment                (.  IItem x, y; .)
=  ident                  (. x = create item from symbol … .)
   "=" Expr<↑y>           (.  if (y.Type.AssignableTo(x.Type))
                                 codeGen.Assign(x, y);  // x: Arg | Local
                              else Error("incompatible types in assignment");
                          .)
.
```

Zuweisungskompatibilität

y ist zuweisungskompatibel mit *x*, wenn

- *x* und *y* gleiche Typen haben (*x.type* == *y.type*) oder
- *x* und *y* Arrays mit gleichen Elementtypen sind oder
- *x* hat einen Referenztyp (Klasse oder Array) und *y* ist *null*

6. Codeerzeugung

Bedingte und unbedingte Sprünge

Unbedingte Sprünge

```
br offset
```

Bedingte Sprünge

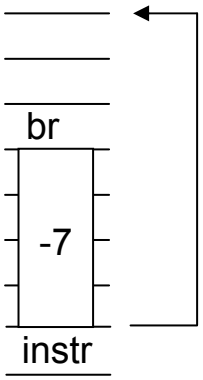
```
... load operand1 ...
... load operand2 ...
beq offset
```

if (operand1 == operand2) br offset

beq	jump on equal
bge	jump on greater or equal
bgt	jump on greater than
ble	jump on less or equal
blt	jump on less than
bne.un	jump on not equal

Vorwärts- und Rückwärtssprünge

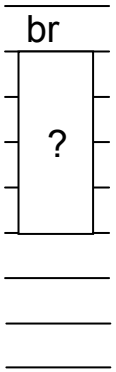
Rückwärtssprünge



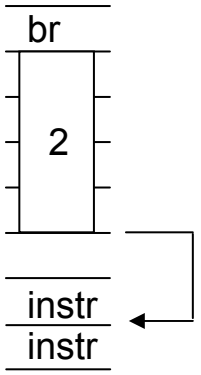
Sprungziel schon bekannt
(da man den Befehl an dieser Stelle schon erzeugt hat)

Sprungdistanz
- 4 Bytes lang (CIL bietet auch Kurzform mit 1 Bytesprungadressen)
- relativ zum Anfang des nächsten Befehls (= Ende des Sprungbefehls)

Vorwärtssprünge



Sprungziel noch unbekannt
⇒ offen lassen
⇒ "Fixupadresse" merken



nachtragen, wenn Zieladresse bekannt wird
(Fixup)

Struct System.Reflection.Emit.Label

Repräsentation von Sprungmarken

```
struct Label { ... }
```

Label werden über *ILGenerator* verwaltet.

```
class ILGenerator {
    ...
    Label DefineLabel ();          // erzeugt eine noch undefinierte Sprungmarke
    void MarkLabel (Label);        // definiert die Marke an der aktuelle Position im IL-Strom
}
```

Benutzung

```
Label label = codeGen.CreateLabel();
...
codeGen.Jump(condition, label);    // Sprung zu noch undefinierter Marke
..
codeGen.MarkLabel(label);          // Sprung zu label führt nun hierher
```

Bedingungen

Conditions

```
if ( a > b ) ...
```

Condition

Codemuster

```
ldloc a
ldloc b
ble ...
```

- *Condition* liefert den Vergleichsoperator
  Vergleich findet dann in Sprungbefehl statt

Cond-Item

Condition liefert eine neue *Item-Art* (ItemKind.Cond) mit folgendem Inhalt:

- Token-Code des Vergleichsoperators:
  ```
  TokenKind relop;  // TokenKind: lss, gtr, eql, neq
  ```
- Sprungmarken für Sprünge aus der Bedingung heraus
  - `Label tLabel` : für True-Jumps
  - `public Label fLabel` : für False-Jumps

} nötig bei Bedingungen, die mit && und || zusammengesetzt sind (In Micro genügt ein Label, keine zusammengesetzten Bedingungen)

True-Jumps und False-Jumps

True-Jump	springt, wenn die Bedingung wahr ist	a > b ⇒ bgt ...
False-Jump	springt, wenn die Bedingung falsch ist	a > b ⇒ ble ...

6. Codeerzeugung

while-Anweisung

Gewünschtes Codemuster

```
while                top:
(Condition)                ... Code für Condition ...
                           FJump end
Statement                  ... Code für Statement ...
                           br top
                   end:    ...
```

Beschreibung durch eine ATG

```
WhileStatement                (.  IItem x; .)
=  "while"                    (.  Label top = DefineLabel();
                                  MarkLabel(top);
                               .)
   "(" Condition<↑x> ")"      (.  FJump(x); .)
   Statement                  (.  Jump(top);
                                  MarkLabel(x.fLabel);
                               .)
   .
```

Beispiel

```
while (a > b) a = a - 2;

10   ldloc.0      <-- top
11   ldloc.1
12   ble 9 (=26)
17   ldloc.0
18   ldc.i4.2
19   sub
20   stloc.0
21   br -16 (=10)
26   ...                    x.fLabel
```

if-Anweisung

Gewünschtes Codemuster

```
if                          ... Condition ...
(Condition)                 FJump end
                            ... Statement ...
Statement               end: ...
```

```
if                          ... Condition ...
(Condition)                 FJump else
                            ... Statement ...
Statement                   br end
                        else:
else                        ... Statement ...
Statement               end: ...
```

Beschreibung durch eine ATG

```
IfStatement                 (.  IItem x; Label end; .)
=  "if"
   "(" Condition<↑x> ")"    (.  FJump(x); .)
   Statement
   (   "else"               (.  end = DefineLabel();
                                Jump(end);
                                MarkLabel(x.fLabel);
                            .)
       Statement            (.  MarkLabel(end); .)
   |                        (.  MarkLabel(x.fLabel; .)
   ).
```

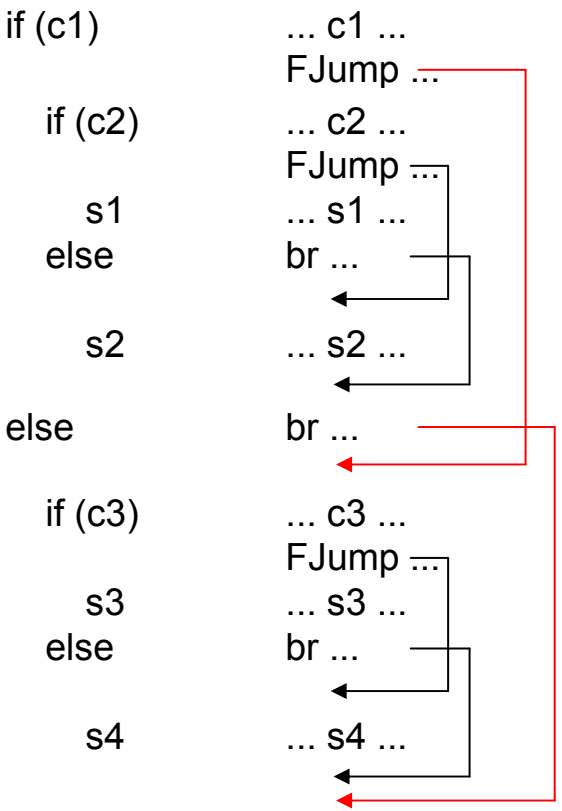
Beispiel

```
if (a > b) max = a; else max = b;
```

```
10   ldloc.0
11   ldloc.1
12   ble 7 (=24)
17   ldloc.0
18   stloc.2
19   br 2 (=26)
24   ldloc.1          x.fLabel
25   stloc.2
26   ...              end 230
```

Funktioniert auch bei geschachtelten ifs


```
IfStatement                    (. IItem x; Label end; .)
=  "if"
   "(" Condition<↑x> ")"       (. FJump(x); .)
   Statement
   (   "else"                  (. end = DefineLabel();
                                  Jump(end);
                                  MarkLabel(x.fLabel);
                               .)
       Statement               (. MarkLabel(end); .)
   |                           (. MarkLabel(x.fLabel); .)
   ).
```



break-Anweisung

Aussprung aus Schleifen

- Marke *breakLab* ans Ende jeder Schleife setzen
- bei Auftreten einer break-Anweisung: Code.Jump(breakLab);

Geschachelte Schleifen

- Jede Schleife braucht ihr eigenes *newBreakLab*
- *newBreakLab* wird als Attribut an innere Anweisungen weitergegeben

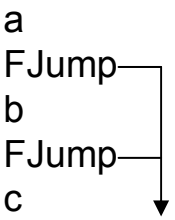
```
Statement<↓Label breakLab>
=  "while"                          (.  Label newBreakLab = DefineLabel(); ... .)
   "(" Condition<↑x> ")"            (. ... .)
   Statement<↓newBreakLab>          (. MarkLabel(newBreakLab); .)

|  "{" { Statement<↓breakLab> } "}"

|  "if"
   ...
    Statement<↓breakLab>
   ...

|  "break"                          (.  if (breakLab.Equals(undef)) Error("break outside a loop");
                                       Jump(breakLab);
                                    .)

|  ... .
```

static readonly Label **undef**;
weil Structs früher nicht null sein konnten

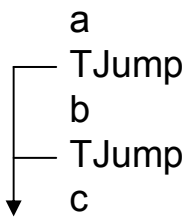
Kurzschlussauswertung Boolescher Ausdrücke

- Boolesche Ausdrücke können mit && und || zusammengesetzt sein
- Berechnung des Ausdrucks wird abgebrochen, sobald sein Ergebnis feststeht

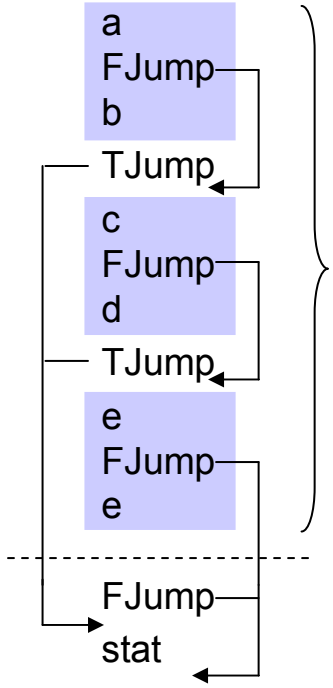
`a && b && c`



`a || b || c`



`if (a && b || c && d || e && f) stat;`



Condition

zwei undefinierte Sprungmarken am Ende von *Condition*

Kleine Änderung in ATG von if-Anweisung

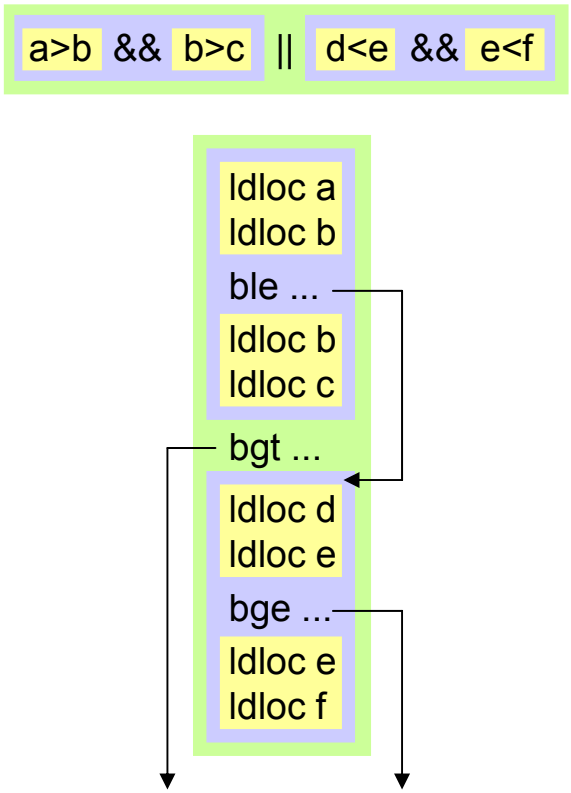
```
IfStatement
= "if"
   "(" Condition<↑x> ")"   (. FJump(x);
                              MarkLabel(x.tLabel);
                           .)
   Statement               (. MarkLabel(x.fLabel); .)
   .
```

Übersetzung Boolescher Ausdrücke


```
CondFactor<↑IItem x> (. IItem y; int op; .)
= Expr<↑x>           (. Load(x); .)
  Relop<↑op>
  Expr<↑y>           (. Load(y);
                        if (!x.type.CompatibleWith(y.type))
                           Error("type mismatch");
                        else if (x.type.IsRefType() &&
                                 op!=TokenKind.eql &&
                                 op!=TokenKind.neq)
                           Error("only equality checks ...");
                        x = Create item from op and x.type …
                     .)
  .
```

```
CondTerm<↑IItem x>   (. IItem y; .)
= CondFactor<↑x>
  { "&&"             (. FJump(x); .)
    CondFactor<↑y>   (. x.relop = y.relop; x.tLabel = y.tLabel; .)
  }.
```

```
Condition<↑IItem x>  (. IItem y; .)
= CondTerm<↑x>
  { "||"             (. TJump(x);
                        MarkLabel(x.fLabel);
                     .)
    CondTerm<↑y>     (. x.relop = y.relop; x.fLabel = y.fLabel; .)
  }.
```



6. Codeerzeugung

Funktionsaufruf

Codemuster

Wird eine Funktion als Prozedur benutzt, muss der Rückgabewert in Statement vom Stack geholt werden!

```
call m <- a, b end
x = call m <- a, b end
```

```
ldloc.s a
ldloc.s b
call T_meth
```

Parameter werden am *estack* übergeben

Beschreibung durch eine ATG

```
Call<↑IItem m>          (.  int fpars, apars = 0; ISymbol fp; IItem ap; .)
=  "call" ident         (. m = CreateItem(FindFunc(token.Str));
                           fPars = m.Sym.NArgs; fp = m.Sym.Locals; .)
   [ "<-" Expr<↑ap>     (. ++apars; Load(ap);
                           if (fp != null && fp.Kind == SymbolKind.Arg) {
                             if (!ap.Type.Equals(fp.Type)) {
                               Error("Parameter type mismatch");
                             }
                             fp = fp.Next;
                           }
                         .)
     { "," Expr<↑ap>     (. … analog to first expression … .)
     } "end"
   ]                     (.  if (aPars > fPars) { Error("Too many actual parameters"); }
                             else if (aPars < fPars) { Error("Too few actual parameters"); }
                             EmitCall(m.Sym); m.Kind = ItemKind.Stack; .)
.
```

Funktionsdeklaration


```
Function                (. IStruct type; ISymbol func .)
=  ident                (. type = FindType(token.Str).Type; .)
   ident                (. func = Insert(SymbolKind.Func, token.Str, type);
                           OpenScope();
                        .)
   [ "<-" FormPar
      { "," FormPar }
   ]                    (. … take arguments from current scope and set them in func
                           CreateMetadata(func);
                           … insert a local variable into the scope with the name and type
                           … of the current function and create metadata for it.
                        .)
   Block                (. … take local variable from current scope and set them in func
                           CloseScope();
                           Load(item for return variable);
                           Emit(OpCodes.ret);
                        .)
   "end"
.
```

Rückgabewert der Funktion.

Formale Parameter

- in Symbolliste eintragen (als *Arg*-Symbole des Methoden-Scopes)
- Methoden-Scope zählt die Anzahl der formalen Parameter (in *nArgs*)

```
FormPar                 (.  IStruct type; .)
=
   ident                (. type = FindType(token.Str).Type; .)
   ident                (.  symTab.Insert(SymbolKind.Arg, token.Str, type); .)
.
```

7. Bottomup-Syntaxanalyse

Arbeitsweise eines Bottomup-Parsers

Beispiel

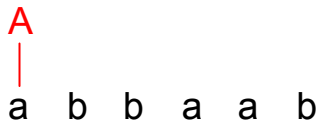
```
S = A B | S A B.
A = a | a a b.
B = b | b b a.
```

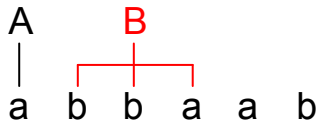
nicht LL(1)!

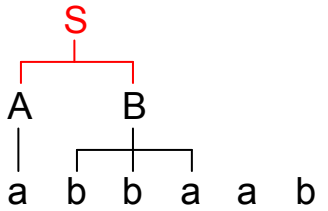
- für Topdown-Analyse ungeeignet
- kein Problem für Bottomup-Analyse

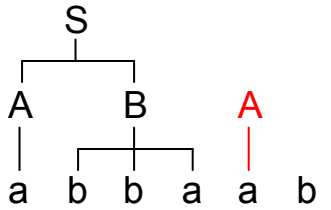
Eingabesatz: a b b a a b

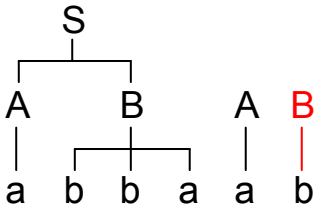
Syntaxbaum wird bottomup aufgebaut

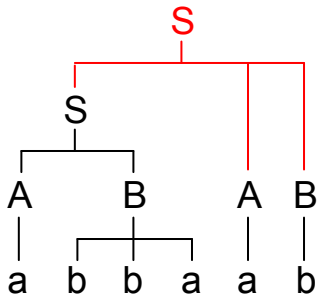












Erkennung mit Hilfe eines Kellers

Grammatik

```
S = A B | S A B.
A = a | a a b.
B = b | b b a.
```

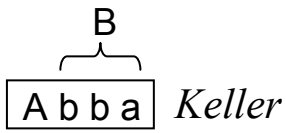
Eingabesatz

a b b a a b # (# ... eof-Symbol)

Keller	*Eingabe*	*Aktion*
	a b b a a b #	lies
a	b b a a b #	reduziere *a* zu *A*
A	b b a a b #	lies
A b	b a a b #	lies (nicht reduzieren, weil sonst Sackgasse!)
A b b	a a b #	lies
A b b a	a b #	reduziere *b b a* zu *B*
A B	a b #	reduziere *A B* zu *S*
S	a b #	lies
S a	b #	reduziere *a* zu *A*
S A	b #	lies
S A b	#	reduziere *b* zu *B*
S A B	#	reduziere *S A B* zu *S*
S	#	Satz erkannt (Keller enthält Satzsymbol, Eingabe leer)

Erkennung mit Hilfe eines Kellers

Was muss der Parser wissen?



- Steht am Kellerende etwas, das reduziert werden kann?
- Zu welchem NTS soll man reduzieren?
- Soll man lesen oder reduzieren, damit man nicht in eine Sackgasse gerät?

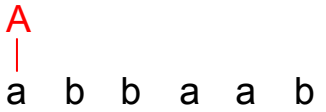
Vier Analyseaktionen

Shift	Lies und kellere nächstes Eingabesymbol (TS)
Reduce	Reduziere Kellerende zu einem NTS
Accept	Satz erkannt (nur bei S . #)
Error	Analyse kann nicht weiter ($\Rightarrow$ Fehlerbehandlung)

Bottomup-Parser heißen daher

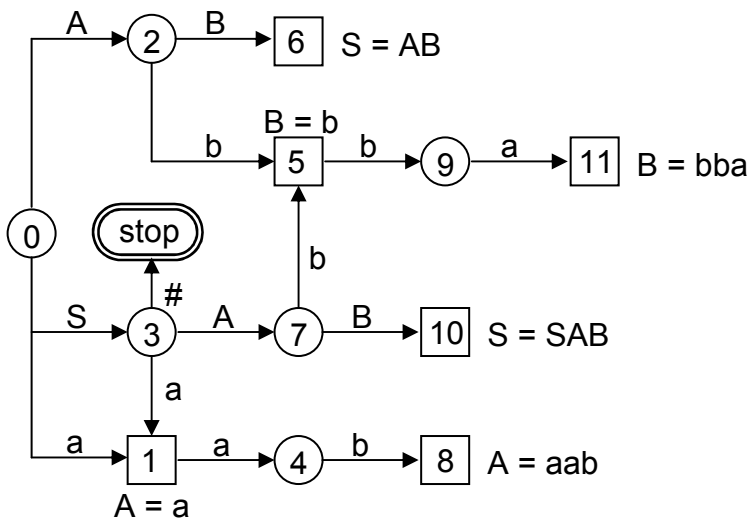
- **Shift-Reduce-Parser**
- **LR-Parser**
  Erkennung von **L**inks nach rechts mit **R**echtskanonischen Ableitungen

  Rechtskanonische Ableitung = linkskanonische Reduktion
  d.h. die linkeste einfache Phrase (der *Ansatz*) wird reduziert



Parser als Kellerautomat

Kellerautomat



Grammatik

1 S = A B.
2 S = S A B.
3 A = a.
4 A = a a b
5 B = b
6 B = b b a

nur reine BNF möglich!

Zustandsübergangstabelle (Parsertabelle)

	TS			NTS		
	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

s1 ... Shift in Zustand 1
r3 ... reduziere nach Produktion 3
- ... Fehler

Erkennung mit Shift-Reduce-Aktionen

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

1 S = A B.
2 S = S A B.
3 A = a.
4 A = a a b
5 B = b
6 B = b b a

Erkennung von a b b a a b #

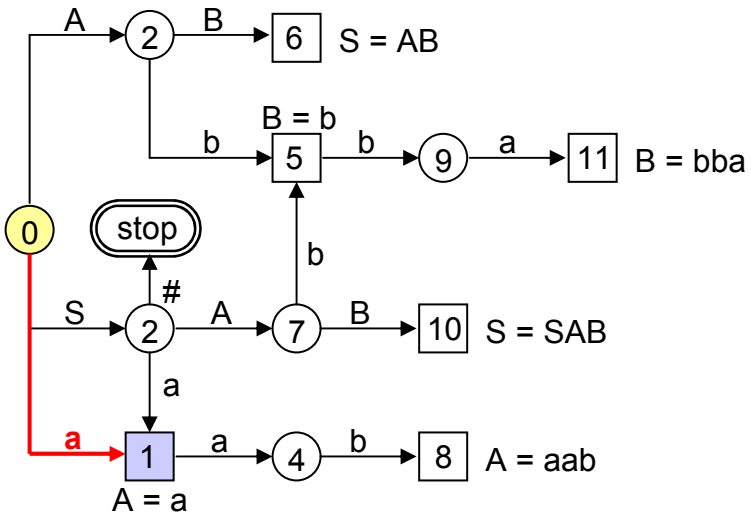
Keller		*Eingabe*	*Aktion*
0		a b b a a b #	s1
0 1		b b a a b #	r3 (A=a)
0	A	b b a a b #	s2 (shift mit A!)
0 2		b b a a b #	s5
0 2 5		b a a b #	s9
0 2 5 9		a a b #	s11
0 2 5 9 11		a b #	r6 (B=bba)
0 2	B	a b #	s6
0 2 6		a b #	r1 (S=AB)
0	S	a b #	s3
0 3		a b #	s1
0 3 1		b #	r3 (A=a)
0 3	A	b #	s7
0 3 7		b #	s5
0 3 7 5		#	r5 (B=b)
0 3 7	B	#	s10
0 3 7 10		#	r2 (S=SAB)
0	S	#	s3
0 3		#	acc

r3 ... reduziere nach Produktion 3 (A = a)

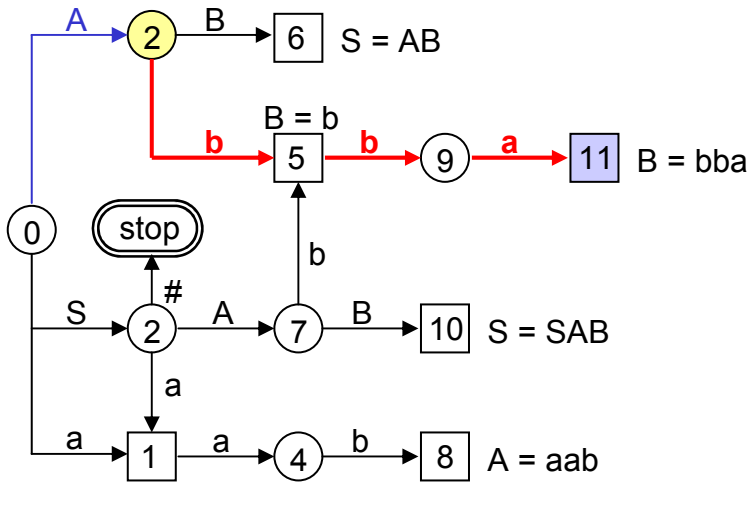
- 1 Zustand auskellern (weil rechte Seite 1 Symbol lang)
- linke Seite (*A*) in die Eingabe einfügen
- Nach reduce erfolgt immer ein shift mit einem NTS

Simulation am Kellerautomaten

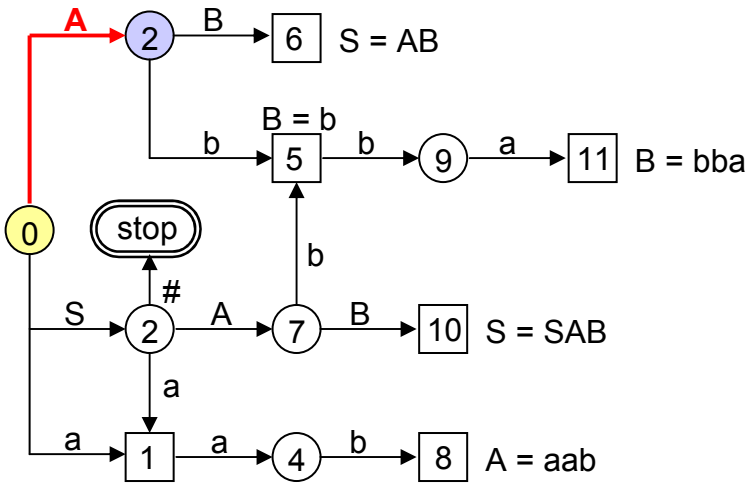
a b b a a b #



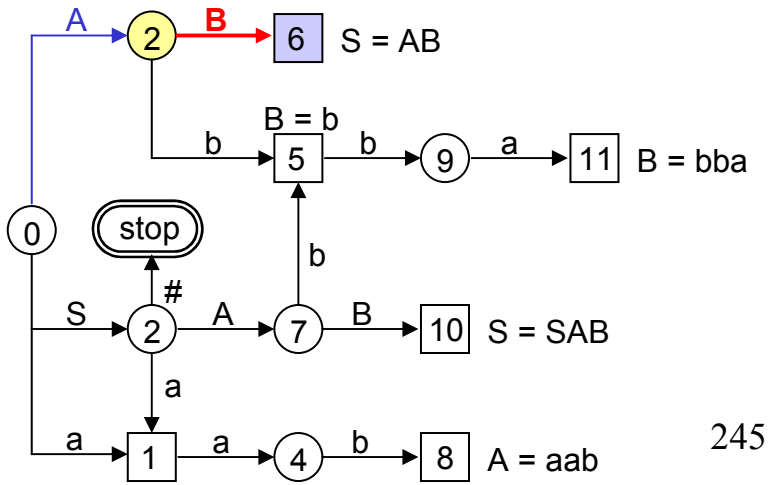
b b a a b #



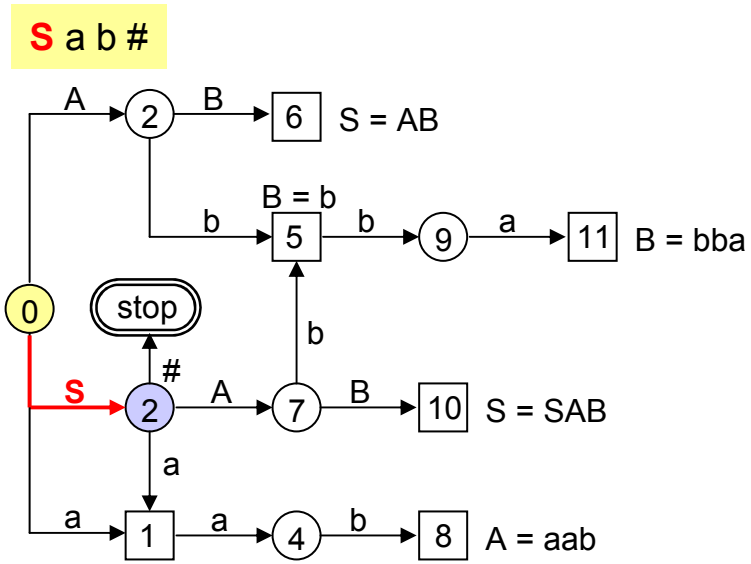
A b b a a b #

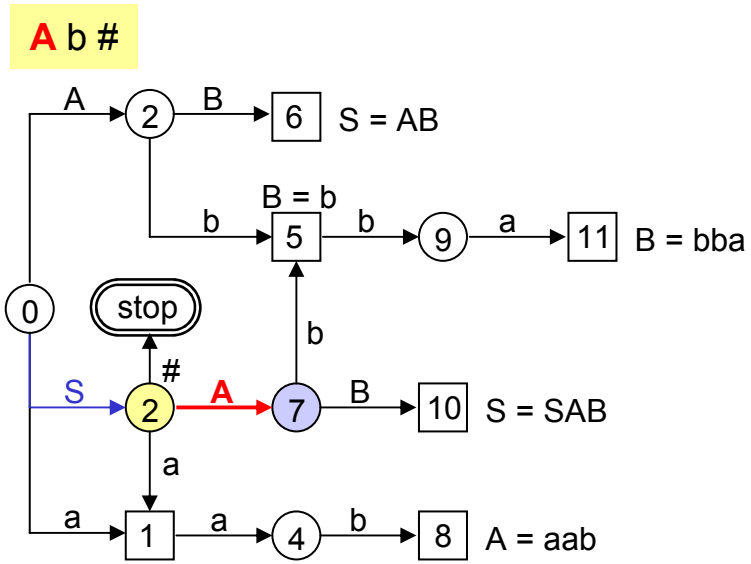


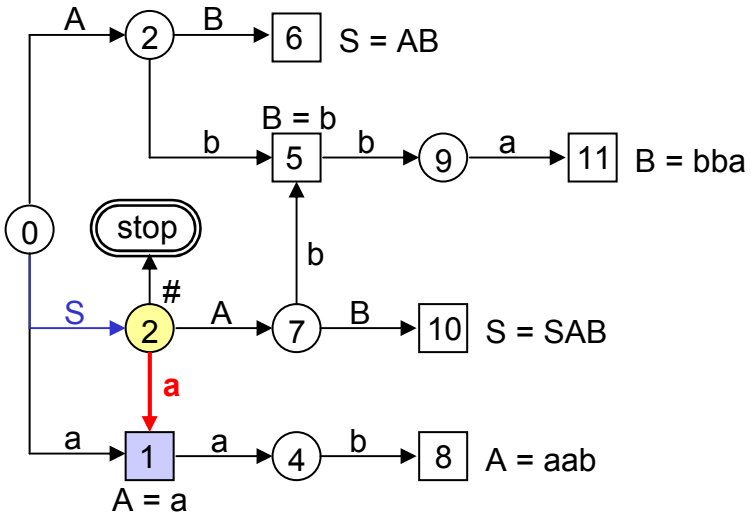
B a b #

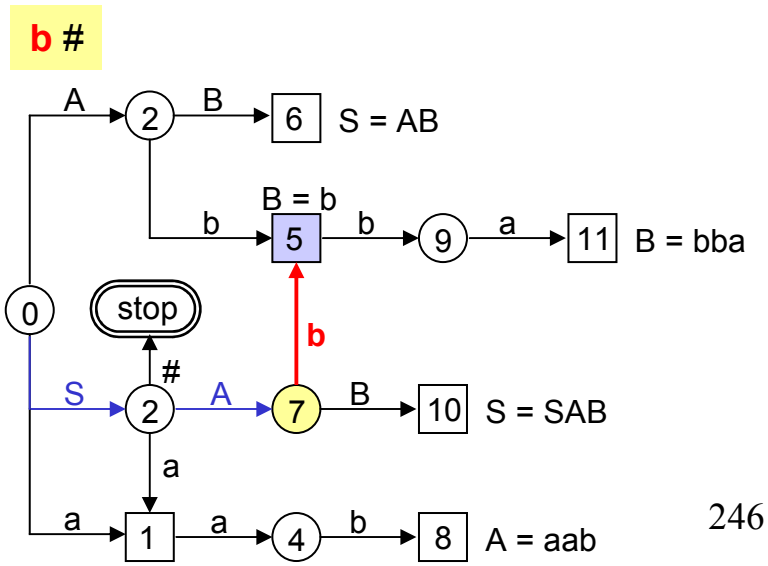


Simulation am Kellerautomaten

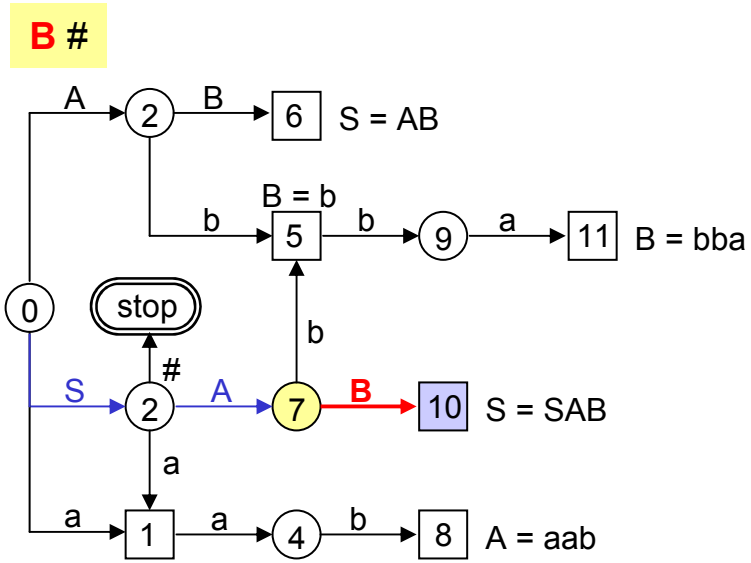


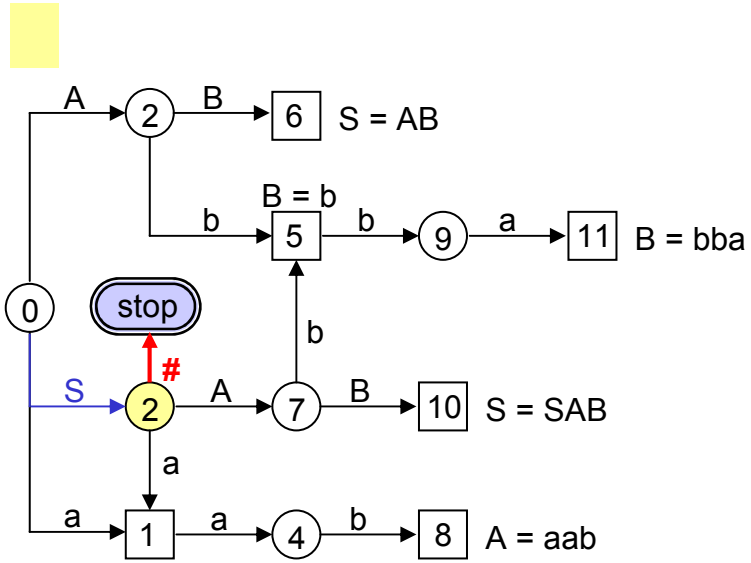


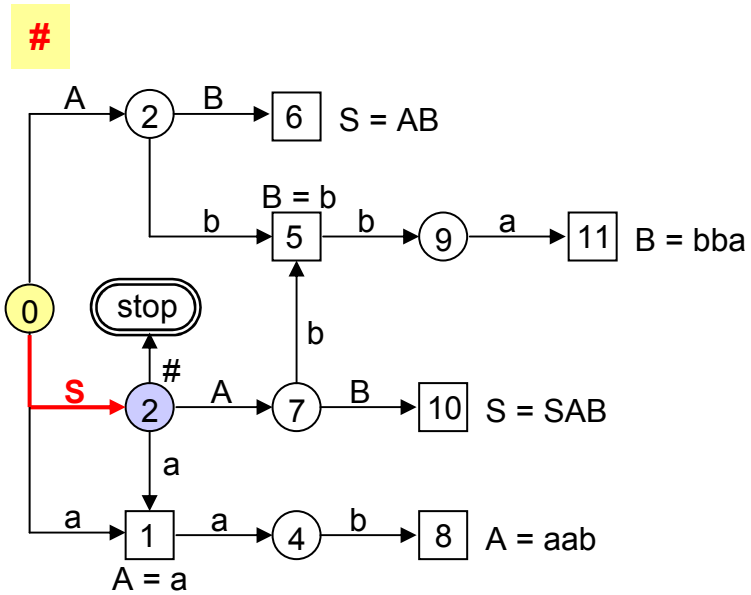




Simulation am Kellerautomaten





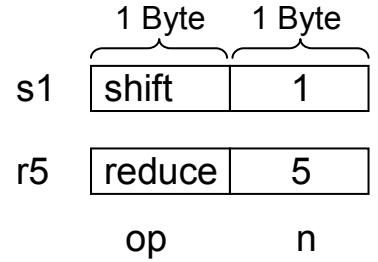


Implementierung des LR-Parsers


```
void Parse () {
   short[,]   action = { {...}, {...}, ...};   // state transition table
   byte[]     length = { ... };                // production lengths
   byte[]     leftSide = { ... };              // left side NTS of every production
   byte       state;                           // current state
   int        sym;                             // next input symbol
   int        op, n, a;

   ClearStack();
   state = 0; sym = Next();
   for (;;) {
      Push(state);
      a = action[state, sym]; op = a / 256; n = a % 256;
      switch (op) {
         case shift:       // shift n
            state = n; sym = Next(); break;
         case reduce:      // reduce n
            for (int i = 0; i < length[n]; i++) Pop();
            a = action[Top(), leftSide[n]]; n = a % 256;  // shift n
            state = n;
            break;
         case accept: return;
         case error: throw new Exception();  // error handling is still missing
      }
   }
}
```

Tabellengesteuertes Programm für beliebige Grammatiken

action-Eintrag (2 Bytes)



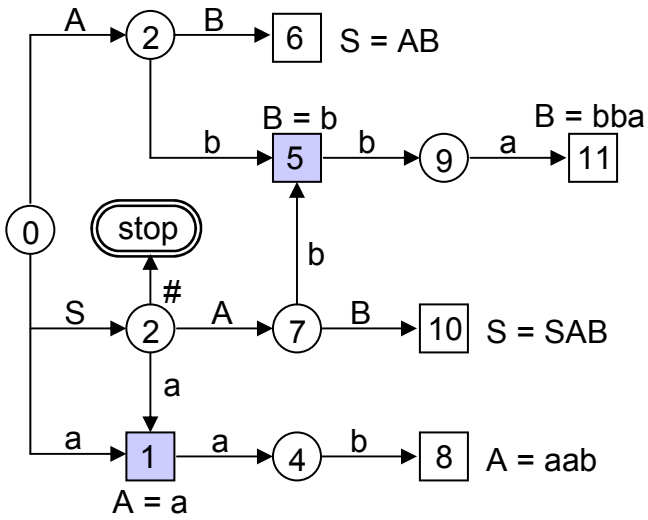
7. Bottomup-Syntaxanalyse

LR(0)- und LR(1)-Grammatiken

LR(0) Erkennbar von **L**inks nach rechts
mit **R**echtskanonischen Ableitungen
und **0** Vorgriffssymbolen

Eine Grammatik ist LR(0)

- wenn es keinen Reduce-Zustand gibt, aus dem auch ein Shift weggeht
- wenn in jedem Reduce-Zustand nur nach einer einzigen Produktion reduziert werden kann



Diese Grammatik ist nicht LR(0)!

LR(1) Erkennbar von **L**inks nach rechts
mit **R**echtskanonischen Ableitungen
und **1** Vorgriffssymbol

Eine Grammatik ist LR(1), wenn in jedem Zustand mit 1 Vorgriffssymbol entscheidbar ist

- ob Shift oder Reduce ausgeführt werden soll
- zu welchem NTS reduziert werden soll

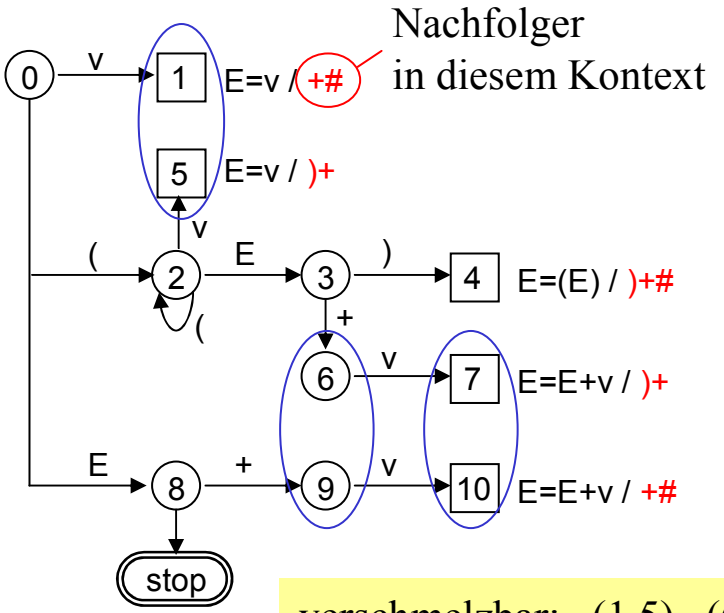
LALR(1)-Grammatiken

Lookahead-LR(1)

- Teilmenge der LR(1)-Grammatiken
- Haben kleinere Tabellen als LR(1)-Grammatiken, weil Zustände mit gleichen Aktionen aber unterschiedlichen Vorgriffssymbolen verschmolzen werden können.

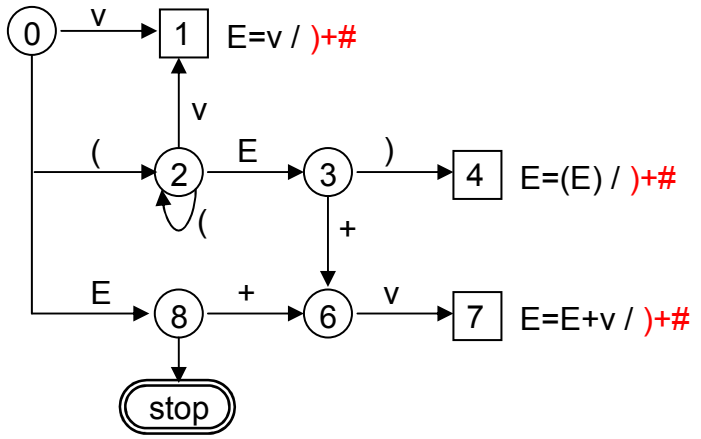
Beispiel E = v | E "+" v | "(" E ")"

LR(1)-Tabellen



verschmelzbar: (1,5) (6,9) (7,10)

LALR(1)-Tabellen



LR-Analyse versus rekursiver Abstieg

Vorteile

- LALR(1) ist mächtiger als LL(1)
  - linksrekursive Produktionen erlaubt
  - Produktionen mit gleichen terminalen Anfängen erlaubt
- LR-Parser sind kompakter als Parser im rekursiven Abstieg
  (Tabellen benötigen allerdings auch viel Speicher)
- Tabellengesteuerter Parser ist universeller Algorithmus, der mit Tabellen parametrisiert wird
- Tabellengesteuerter Parser erlaubt bessere Fehlerbehandlung

Nachteile

- LALR(1)-Tabellen sind für große Grammatiken schwer zu konstruieren (man braucht Werkzeuge)
- LR-Parser sind etwas langsamer als RD-Parser
- Semantikanschluss ist beim LR-Parser komplizierter
- Die Analyse ist beim LR-Parser schwer verfolgbar

LR-Parser lohnen sich nur für komplexe Sprachen.
Für kleinere Sprachen (z.B. Kommandosprachen) ist rekursiver Abstieg besser.

7. Bottomup-Syntaxanalyse

LR-Items

Beispiel: Analyse von

```
S = a A b
A = c
```

Parser (gekennzeichnet durch Punkt) **bewegt sich durch die Produktion**

```
S = . a A b
S = a . A b   ←
A = . c
A = c .
S = a A . b
S = a A b .
```

wenn der Parser vor *A* steht, steht er gleichzeitig vor der rechten Seite der *A*-Produktion

LR-Item

Schnappschuss der Syntaxanalyse

$X = \alpha \ . \ \beta \ / \ \gamma$

- γ: Nachfolger von *X* in diesem Kontext
- β: noch ungelesener Teil der Produktion
- .: Parserposition
- α: gelesener und gekellerter Teil der Produktion

Steuersymbol

Symbol, das auf den Punkt folgt.
z.B.:

A = a . a b / c	Steuersymbol = a
A = a a b . / c	Steuersymbol = c

Shift-Item	$X = \alpha \ . \ \beta \ / \ \gamma$	Punkt steht nicht am Regelende
Reduce-Item	$X = \alpha \ . \ / \ \gamma$	Punkt steht am Regelende

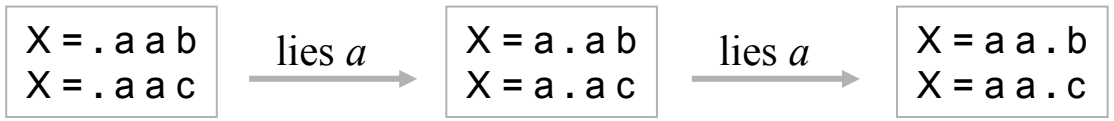
Parserzustand als Itemmenge

Zustand des Parsers

Repräsentiert durch die Menge der Items, an denen der Parser gerade arbeitet

```
S = A . B c    / #
B = . b        / c
B = . b b a    / c
```

Topdown-Parser arbeitet immer nur an *einer* Produktion.
Bottomup-Parser kann an *mehreren* Produktionen *gleichzeitig* arbeiten.



erst jetzt werden die beiden Produktionen an Hand des Vorgriffs-symbols unterschieden

Bottomup-Parser sind daher mächtiger als Topdown-Parser.

Kern und Hülle eines Zustands

Kern

Alle Items eines Zustands, die nicht mit Punkt beginnen (außer in Produktion des Startsymbols)

```
S = A . B c / #
```

Aus dem Kern lassen sich alle anderen Items des Zustands ableiten

Hülle

```
do {
   if (Zustand hat ein Item der Art X = α . Y β / γ)
      füge alle Items Y = . ω / First(β) zum Zustand hinzu;
} while (Zustand hat neue Items bekommen);
```

Beispiel

```
S = A . B c    / #
B = . b        / c
B = . b b a    / c
```

Kern

Hülle

```
S = A B c
A = a
A = a a b
B = b
B = b b a
```

Beispiel: Berechnung der Hülle

Grammatik

```
0   S' = S #
1   S = A B
2   S = S A B
3   A = a
4   A = a a b
5   B = b
6   B = b b a
```

Kern

```
S' = . S #
```

Füge alle *S*-Produktionen hinzu

```
S' = . S #
S = . A B     / #
S = . S A B   / #
```

Füge alle *A*-Produktionen hinzu

```
S' = . S #
S = . A B     / #
S = . S A B   / #
A = . a       / b
A = . a a b   / b
```

Füge alle *S*-Produktionen hinzu (wegen S = . S A B / #)
Nachfolger von *S* ist hier *a*

```
S' = . S #
S = . A B     / #a
S = . S A B   / #a
A = . a       / b
A = . a a b   / b
```

} Hülle

Folgezustand

Succ(state, sym)

Zustand, in dem man aus *state* mit *shift sym* kommt (*sym* = Steuersymbol)

Beispiel

state *i*:

```
S = A . B c    / #
B = . b        / c
B = . b b a    / c
```

Succ(*i*, *b*):

```
B = b .        / c
B = b . b a    / c
```

- enthält alle Items von *i* mit Steuersymbol *b*
- Punkt befindet sich hinter *b*
- diese Produktionen werden parallel weiterverfolgt

Succ(*i*, *B*):

```
S = A B . c    / #
```

Succ(state, sym) ist nur der Kern des neuen Zustands, der erst zur Hülle erweitert werden muss.

LALR(1)-Tabellenerzeugung


```
Erweitere Grammatik um Pseudoproduktion S' = S #
Erzeuge Zustand 0 mit Kern S' = . S #  // Ausnahme: einziger Kern mit Punkt am Anfang

while (noch nicht alle Zustände betrachtet) {
   s = nächster noch nicht betrachteter Zustand;
   Bilde Hülle von s;
   for (alle Items von s) {
      switch (Item-Art) {
         case X = S . # :        erzeuge accept; break;
         case X = α . y β / γ :  erzeuge neuen Zustand s1 = Succ(s, y) falls noch nicht vorhanden
                                 erzeuge shift y, s1; break;
         case X = α . / γ :      erzeuge reduce γ, (X = α); break;
         default:                erzeuge error; break;
      }
   }
}
```

Bei der Prüfung, ob ein Zustand schon vorhanden ist, werden nur die Zustandskerne ohne Nachfolgersymbole überprüft, z.B.:

vorhanden		neu		
E = v . / #+	+	E = v . /)+	⇒	E = v . /)#+

LALR(1)-Tabellenerzeugung

Grammatik

0	S' = S #
1	S = A B
2	S = S A B
3	A = a
4	A = a a b
5	B = b
6	B = b b a

0	S' = . S #		shift	a	1
	S = . A B	/ #a	shift	A	2
	S = . S A B	/ #a	shift	S	3
	A = . a	/ b			
	A = . a a b	/ b			
1	A = a .	/ b	red	b	3
	A = a . a b	/ b	shift	a	4
2	S = A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	6
	B = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . A B	/ #a	shift	a	1 (!)
	A = . a	/ b	shift	A	7
	A = . a a b	/ b			
4	A = a a . b	/ b	shift	b	8
5	B = b .	/ #a	red	#,a	5
	B = b . b a	/ #a	shift	b	9
6	S = A B .	/ #a	red	#,a	1
7	S = S A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	10
	B = . b b a	/ #a			
8	A = a a b .	/ b	red	b	4
9	B = b b . a	/ #a	shift	a	11
10	S = S A B .	/ #a	red	#,a	2
11	B = b b a .	/ #a	red	#,a	6

Parser-Tabelle

0	S' = . S #		shift	a	1
	S = . A B	/ #a	shift	A	2
	S = . S A B	/ #a	shift	S	3
	A = . a	/ b			
	A = . a a b	/ b			
1	A = a .	/ b	red	b	3
	A = a . a b	/ b	shift	a	4
2	S = A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	6
	B = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . A B	/ #a	shift	a	1
	A = . a	/ b	shift	A	7
	A = . a a b	/ b			
4	A = a a . b	/ b	shift	b	8
5	B = b .	/ #a	red	#,a	5
	B = b . b a	/ #a	shift	b	9
6	S = A B .	/ #a	red	#,a	1
7	S = S A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	10
	B = . b b a	/ #a			
8	A = a a b .	/ b	red	b	4
9	B = b b . a	/ #a	shift	a	11
10	S = S A B .	/ #a	red	#,a	2
11	B = b b a .	/ #a	red	#,a	6

Zustandsübergangstabelle

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Parser-Tabelle in Listenform

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Fehlereinträge bei NT-Aktionen
können eigentlich nie vorkommen

	TS-Aktionen		*NTS-Aktionen*	
0	a	s1	S	s3
	*	error	A	s2
1	a	s4		
	*	r3		
2	b	s5	B	s6
	*	error		
3	a	s1	A	s7
	#	acc		
	*	error		
4	b	s8		
	*	error		
5	b	s9		
	*	r5		
6	*	r1		
7	b	s5	B	s10
	*	error		
8	*	r4		
9	a	s11		
	*	error		
10	*	r2		
11	*	r6		

- Aktionen in jedem Zustand werden seq. geprüft
- letzte T-Aktion jedes Zustands ist * *error* oder * *ri*
  (falls fälschlicherweise reduziert wird, wird der Fehler beim nächsten shift bemerkt)
- kompakter aber langsamer als mit einer Tabelle

LALR(1)- versus LR(1)-Tabellen

LR(1) ist etwas mächtiger als LALR(1), hat aber etwa 10 x größere Tabellen

LR(1)-Tabellenerzeugung

- Zustände werden nie verschmolzen
- Grammatik ist LR(1), wenn in keinem Zustand einer der folgenden Konflikte auftritt:

shift-reduce-Konflikt	shift a ... red a ...	Parser kann sich mit 1 Symbol Vorgriff nicht entscheiden, ob er lesen oder reduzieren soll
reduce-reduce-Konflikt	red a n red a m	Parser kann sich mit 1 Symbol Vorgriff nicht entscheiden, zu welchem NTS er reduzieren soll

LALR(1)-Tabellenerzeugung

- Zustände dürfen verschmolzen werden, wenn Kerne bis auf Vorgriffssymbole gleich sind

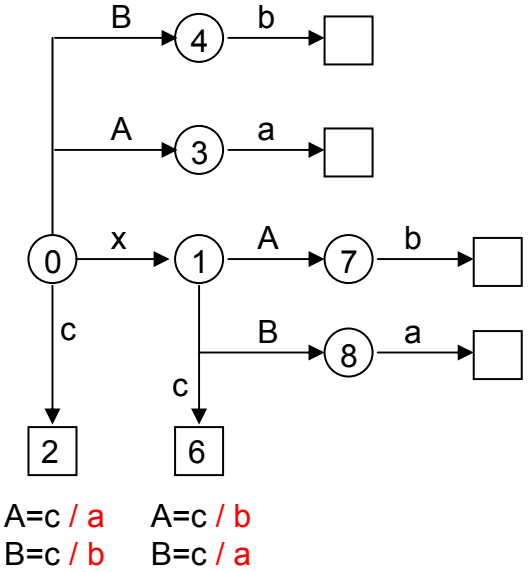
E = v . / #+ + E = v . /)+ ⇒ E = v . /)#+

- Grammatik ist LALR(1), wenn sich nach dem Verschmelzen kein Parse-Konflikt ergibt (kann höchstens ein reduce-reduce-Konflikt sein; warum?)

Beispiel: LR(1)-Grammatik, die nicht LALR(1) ist

Grammatik

S' = S #
S = x A b
S = x B a
S = A a
S = B b
A = c
B = c



0	S' = . S #		shift	x	1
	S = . x A b	/ #	shift	c	2
	S = . x B a	/ #	shift	A	3
	S = . A a	/ #	shift	B	4
	S = . B b	/ #	shift	S	5
	A = . c	/ a			
	B = . c	/ b			
1	S = x . A b	/ #	shift	c	6
	S = x . B a	/ #	shift	A	7
	A = . c	/ b	shift	B	8
	B = . c	/ a			
2	A = c .	/ a	red	a	(A = c)
	B = c .	/ b	red	b	(B = c)
...	...				
6	A = c .	/ b	red	b	(A = c)
	B = c .	/ a	red	a	(B = c)
...	...				

Verschmelzen von 2 und 6 würde zu folgendem Zustand führen

2	A = c .	/ ab	red	a,b	(A = c)
	B = c .	/ ab	red	a,b	(B = c)

reduce-reduce-Konflikt!

SLR(1)-Tabellen (Simple LR(1))

- SLR(1) ist weniger mächtig als LALR(1)
+ SLR(1)-Tabellen sind einfacher zu erzeugen als LALR(1)-Tabellen

SLR(1)-Items

Shift-Items	$X = \alpha \,.\, \beta$	⟵	es werden keine Nachfolger gespeichert
Reduce-Items	$X = \alpha \,.\, / \gamma$	⟵	γ = Follow(*X*), d.h. alle Nachfolger von *X* in jedem beliebigen Kontext

Beispiel: LALR(1)-Grammatik, die nicht SLR(1) ist

Grammatik

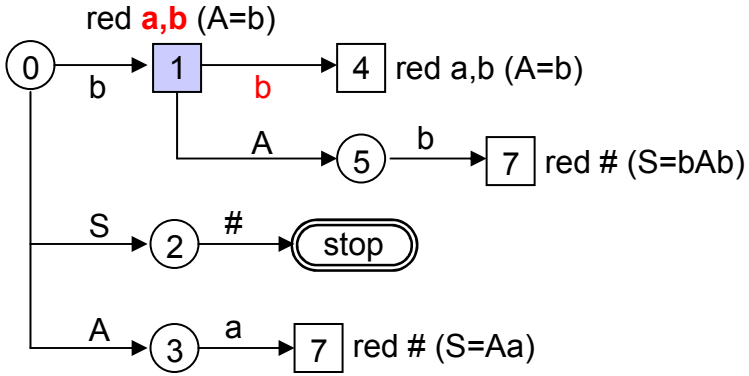
S' = S #
S = b A b
S = A a
A = b

0	S' = . S #	shift	b	1
	S = . b A b	shift	S	2
	S = . A a	shift	A	3
	A = . b			
1	S = b . A b	shift	b	4
	A = b .	red	a,b	(A = b)
	A = . b	shift	A	5

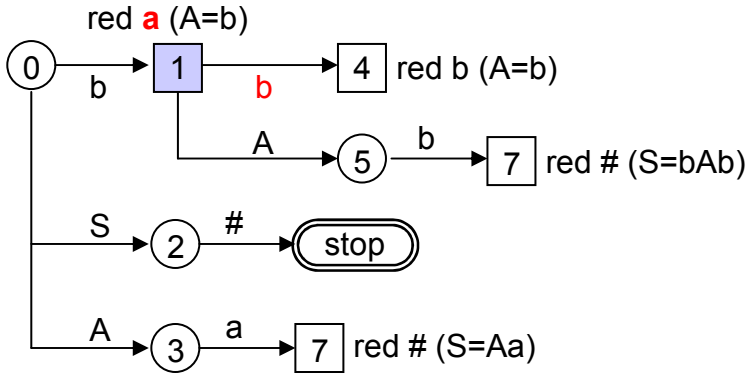
shift-reduce-Konflikt, der in LALR(1)-Tabellen nicht aufgetreten wäre

Follow(S) = {#}
Follow(A) = {a, b}

SLR(1)



LALR(1)



7. Bottomup-Syntaxanalyse

Größenabschätzung

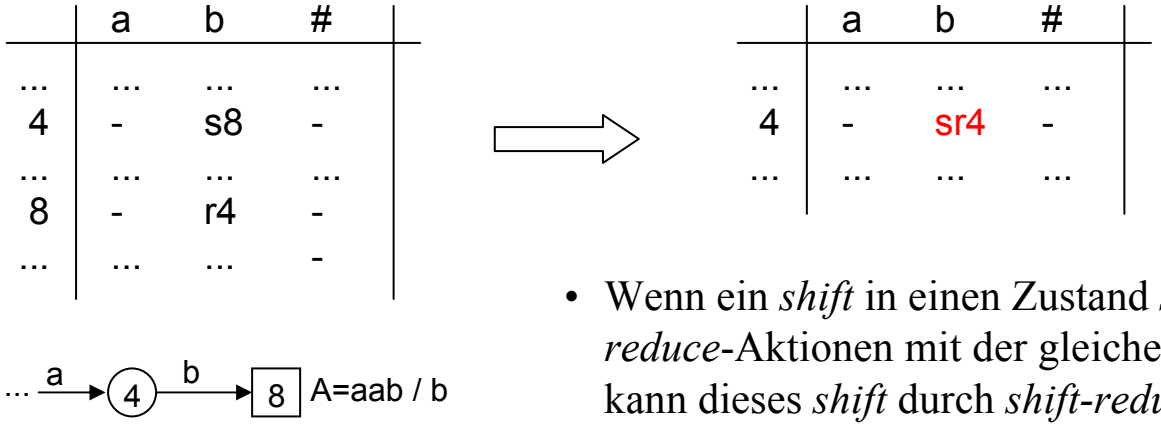
Annahme (z.B. C#)

80 Terminalsymbole
200 Nonterminalsymbole
2500 Zustände ⇒ 280 x 2.500 = 700.000 Elemente

4 Byte pro Element ⇒ 700.000 x 4 = 2.800.000 Byte = 2,8 MByte

Tabellengröße kann um ca. 90% verkleinert werden

Zusammenfassen von shift und reduce

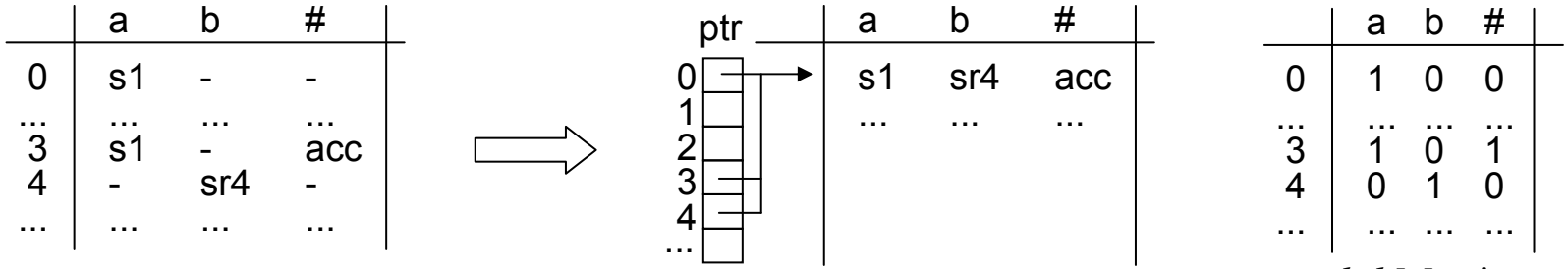


- Wenn ein *shift* in einen Zustand *s* führt, in dem nur *reduce*-Aktionen mit der gleichen Produktion vorkommen, kann dieses *shift* durch *shift-reduce* ersetzt werden.
- Zustand *s* kann dann entfallen

Änderung im Analysealgorithmus

```
...
switch (op) {
   case shift:   // shift n
      state = n; sym = Next(); break;
   case shiftred:  // shiftred n
      for (int i = 0; i < length[n]-1; i++) Pop();
      a = action[Top(), leftSide[n]]; n = a % 256;  // shift n
      state = n;
      break;
   ...
```

Zeilenverschmelzung



valid-Matrix

- Zeilen, deren Aktionen beim Verschmelzen nicht in Konflikt stehen, können verschmolzen werden.
- Benötigt Zeigerarray, um indirekt auf die richtige Zeile zuzugreifen.
- Benötigt Bitmatrix, die sagt, welche Aktionen in den jeweiligen Zuständen gültig sind.

- T- und NT-Aktionen sollten unabhängig voneinander verschmolzen werden.
- Gleiche Technik kann auch für Spaltenverschmelzung angewendet werden.
- Weitere Kompression möglich, wird aber immer teurer.

Änderung im Analysealgorithmus

```
BitArray[] valid;
short[] ptr;
...
a = action[ptr[state], sym];
op = a / 256; n = a % 256;
if (!valid[state][sym]) op = error;
switch (op) {
   ...
```

Beispiel

Originaltabelle (6 Spalten x 12 Zeilen x 2 Bytes = **144** Bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Zusammenfassen von shift und reduce (6 Spalten x 8 Zeilen x 2 Bytes = **96** Bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Beispiel (Fortsetzung)

Nach Zusammenfassen von shift und reduce (**96** Bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Zeilenverschmelzung (3 Spalten * 6 Zeilen * 2 Bytes
+ 2 * 8 Zeilen * 2 Bytes + 8 * 1 Byte = 36 + 32 + 8 = **76** Bytes)

T-Aktionen

	a	b	#
0,3,4	s1	sr4	acc
1	s4	r3	-
2,7,9	sr6	s5	-
5	r5	s9	r5

NT-Aktionen

	S	A	B
0,2	s3	s2	sr1
3,7	-	s7	sr2

valid

	a	b	#
0	1	0	0
1	1	1	0
2	0	1	0
3	1	0	1
4	0	1	0
5	1	1	1
7	0	1	0
9	1	0	0

7. Bottomup-Syntaxanalyse

Semantische Aktionen

Sind nur am Ende von Produktionen möglich (d.h. beim Reduzieren)

Semantische Aktionen inmitten von Produktionen

```
X = a (. ... .) b.
```

müssten folgendermaßen implementiert werden:

```
X = a Y b.
Y = (. ... .).
```
← leere Produktion, bei deren Reduktion die sem. Aktion ausgeführt wird

Problem: das kann die LALR(1)-Eigenschaft zerstören

```
X = a b c.
X = a (. ... .) b d.
```

KFG wäre LALR(1) ⇒

```
X = a b c.
X = a Y b d.
Y = (. ... .).
```

Tabellenerzeugung

i	X = a . b c / #	shift	b	i+1
	X = a . Y b d / #	red	b	(Y=)
	Y = . / b	shift	Y	i+2

shift-reduce-Konflikt

Grund: Parser kann nicht mehr beide Produktionen parallel verfolgen, sondern muss sich entscheiden, in welcher Produktion er ist (d.h. ob er die sem. Aktion ausführen soll oder nicht).

Attribute

- Jedes Symbol hat *ein* Ausgangsattribut (u.U. ein Objekt mit mehreren Feldern)
- Nach Erkennen eines Symbols wird sein Attribut auf einen *Semantikkeller* gelegt

Beispiel

```
Expr↑x  = Term↑x .
Expr↑x  = Expr↑x "+" Term↑y      (. Push(Pop() + Pop(); .).    // x = x + y;
Term↑x  = Factor↑x .
Term↑x  = Term↑x "*" Factor↑y    (. Push(Pop() * Pop(); .).    // x = x * y;
Factor↑x  = const↑t              (. t = Pop(); Push(t.val); .).
Factor↑x  = "(" Expr↑x ")".
```

Änderungen im Parser

Produktionentabelle

	leftSide	length	sem
0	Expr	1	0
1	Expr	3	1
2	Term	1	0
3	Term	3	2
4	Factor	1	3
5	Factor	3	0

Semantikauswerter

```
void SemAction (int n) {
   switch (n) {
      case 1: Push(Pop() + Pop()); break;
      case 2: Push(Pop() * Pop()); break;
      case 3: Token t = Pop(); Push(t.val); break;
   }
}
```

Variablen, die in mehreren sem. Aktionen vorkommen, müssen global sein (Probleme bei rekursiven NTS).

Parser

```
...
switch(op) {
   ...
   case reduce:  // red n
      if (sem[n] != 0) SemAction(sem[n]);
      for (int i = 0; i < length[n]; i++) Pop();
      ...
}
```

Eingangsattribute

Implementierbar als semantische Aktionen mit Attributzuweisungen:

```
X = a Y<↓v> b.
Y<↓w>  = ... .
```

kann dargestellt werden als

```
X = a (. w = v; .) Y b.
Y  = ... .
```

kann aber wieder die LALR(1)-Eigenschaft zerstören

Daher

- In der Praxis bauen LALR(1)-Parser einfach einen Syntaxbaum auf
  (kann mit Ausgangsattributen und sem. Aktionen am Regelende implementiert werden)
- Semantikauswertung findet dann am Syntaxbaum statt

7. Bottomup-Syntaxanalyse

Idee

Situation beim Auftreten eines Fehlers

$s_0 \ldots s_n \;.\; t_0 \ldots t_m \,\#$

Stack Input

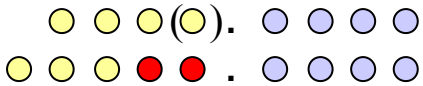
Ziel: Stack und Input so synchronisieren, dass gilt:

$s_0 \ldots s_i \;.\; t_j \ldots t_m \,\#$

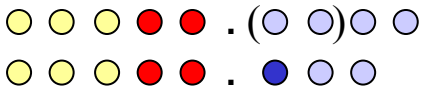
und t_j in s_i akzeptiert wird

Vorgehensweise

- Zustände ein- und/oder auskellern



- Token einfügen und/oder löschen



Algorithmus

1. Suche Fluchtweg

- Ersetze t_0 .. t_m durch eine virtuelle Eingabe v_0 .. v_k, die den Parser vom Fehlerzustand s_n möglichst schnell in den Endzustand steuert.
- Sammle bei der Analyse von v_0 .. v_k alle Token, die in durchlaufenen Zuständen gültig sind $\Rightarrow$ Anker

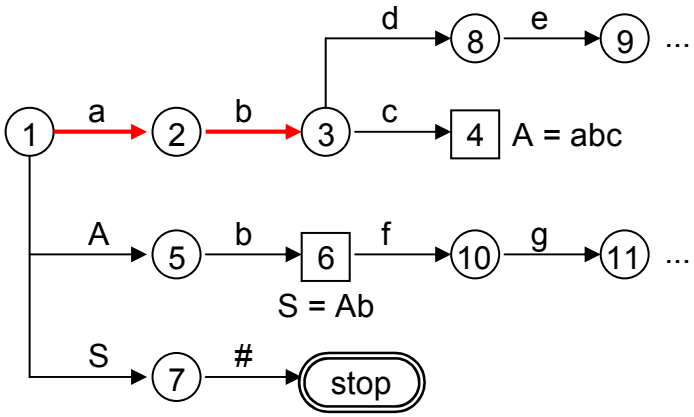
2. Lösche fehlerhafte Token

- Überlies Token aus der Eingabe t_0 .. t_m, bis ein Token t_j auftritt, das ein Anker ist.

3. Füge fehlende Token ein

- Steuere die Analyse von s_n weg mit v_0 .. v_k, bis ein Zustand s_i erreicht wird, in dem t_j gültig ist.
- Füge alle "gelesenen" virtuellen Token von v_0 .. v_k in den Quelltext vor t_j ein.

Beispiel



Eingabe: a b b #

Suche Fluchtweg

virtuelle Eingabe: c b #

Anker:

(3)	c, d
(5)	b
(6)	f
(7)	#

Füge fehlende Token ein

c einfügen, um nach [4] und (5) zu kommen.

In (5) geht es mit *b* wieder weiter.

Lösche fehlerhafte Token

Es wird nichts überlesen, weil *b* bereits ein Anker ist

Korrigierte Eingabe

a b c b #

Fehlermeldungen

Geben an, was eingefügt oder gelöscht wurde

- Wenn Token *a*, *b*, *c* aus der restlichen Eingabe gelöscht wurden

  line ... col ... : "a b c" deleted

- Wenn Token *x, y* vor der restlichen Eingabe eingefügt wurden

  line ... col ... : "x y" inserted

- Wenn Token *a*, *b*, *c* aus der restlichen Eingabe gelöscht und *x*, *y* eingefügt wurden

  line ... col ... : "a b c" replaced by "x y"

Nochmals Beispiel (mit Simulation des Parsers)

Grammatik

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

Fehlerhafte Eingabe

a a a b #

Jeder Zustand hat ein "Wegweisersymbol". Wie man dazu kommt, siehe später.

Beginn der Analyse

Stack	*Input*	*Aktion*
0	a a a b #	s1
0 1	a a b #	s4
0 1 4	a b #	-- error!

Fluchtweg suchen und Anker sammeln

Stack	*guide*	*Aktion*	*Anker*
0 1 4	b	s8	b
0 1 4 8	b	r4, s2	b
0 2	b	s5	b
0 2 5	a	r5, s6	a, b, #
0 2 6	a	r1, s3	a, #
0 3	#	acc	a, #

Anker = {*a*, *b*, #}

Beispiel (Fortsetzung)

Grammatik

0	S' = S #
1	S = A B
2	S = S A B
3	A = a
4	A = a a b
5	B = b
6	B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

restliche Eingabe

a b #

Anker

{a, b, #}

Input überlesen

Man braucht nichts zu überlesen, weil nächstes Token *a* bereits in der Ankermenge {*a*, *b*, #} ist

Fehlende Token einfügen

Stack	*guide*	*Aktion*	*eingefügt*
0 1 4	b	s8	b
0 1 4 8	b	r4, s2	
0 2	b	s5	b
0 2 5			

← Nur *shift* führt zu eingefügten Token, nicht *reduce*

Hier kann mit *a* fortgesetzt werden

Beispiel (Fortsetzung)

Grammatik

0	S' = S #
1	S = A B
2	S = S A B
3	A = a
4	A = a a b
5	B = b
6	B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

restliche Eingabe

a b #

Fortsetzen der Analyse

Stack	*Input*	*Aktion*
0 2 5	a b #	r5, s6
0 2 6	a b #	r1, s3
0 3	a b #	s1
0 3 1	b #	r3, s7
0 3 7	b #	s5
0 3 7 5	#	r5, s10
0 3 7 10	#	r2, s3
0 3	#	acc

Korrigierte Eingabe

a a b b a b #

Fehlermeldung

line ... col ...: "b b" inserted

Wie findet man die Wegweiser?

1. Produktionen so ordnen, dass die erste Produktion jedes NTS nicht rekursiv und möglichst kurz ist

```
S = A B.
S = S A B.   ← rekursive Produktion als zweite
A = a.
A = a a b.   ← längere Produktion als zweite
B = b.
B = b b a.   ← längere Produktion als zweite
```

2. Normale LALR(1)-Tabellenerzeugung.
   Die erste erzeugte T-Aktion gibt den Wegweiser an

```
0   S' = . S #              shift   a   1  ← Wegweiser ist a
    S = . A B     / #a      shift   A   2
    S = . S A B   / #a      shift   S   3
    A = . a       / b
    A = . a a b   / b
```

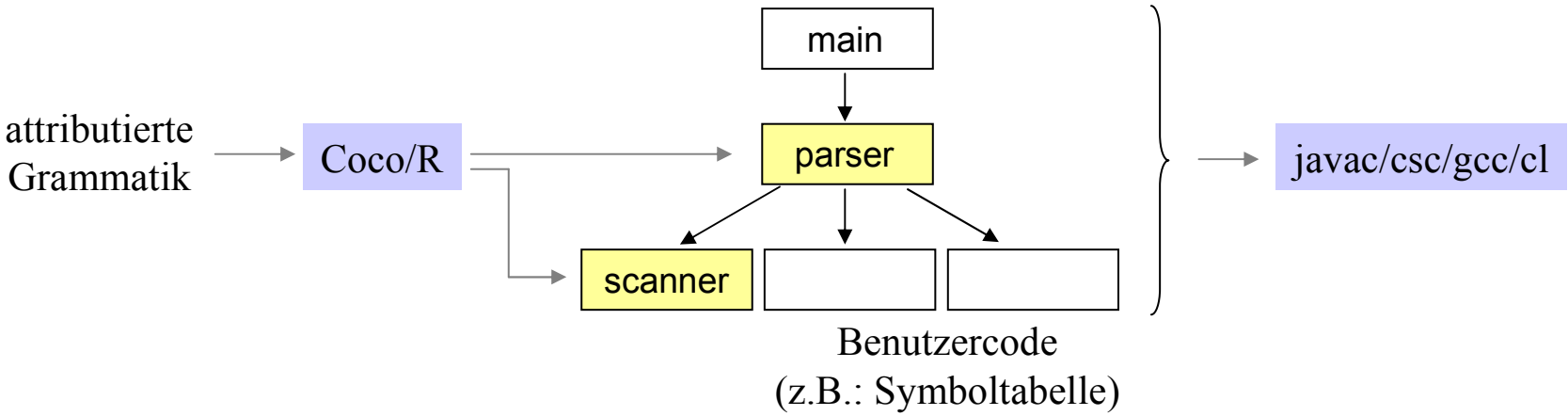
Beurteilung dieser Fehlerbehandlungstechnik

- Behindert fehlerfreie Analyse nicht
- Terminiert immer (# ist immer ein Anker)
- Meldet die Fehler nicht nur, sondern "korrigiert" sie auch.
  Die Korrektur ist allerdings nicht immer die, die man will.
- Ermöglicht gute Fehlermeldungen

8. Erzeugen von Generatoren mit Coco/R

Coco/R - Compiler Compiler / Recursive Descent

Erzeugt einen Scanner und einen Parser aus einer ATG



Scanner	DFA
Parser	Rekursiver Abstieg
Erste Version	1980, Universität Linz
Aktuelle Versionen	**Java**, **C#**, **C++**, Delphi, Modula-2, Visual Basic, Oberon, Python, ...
Open source	http://ssw.jku.at/Coco/
Vergleichbare Tools	Lex+Yacc, JavaCC, ANTLR, ...

Beispiel: Compiler für Arithmetische Ausdrücke

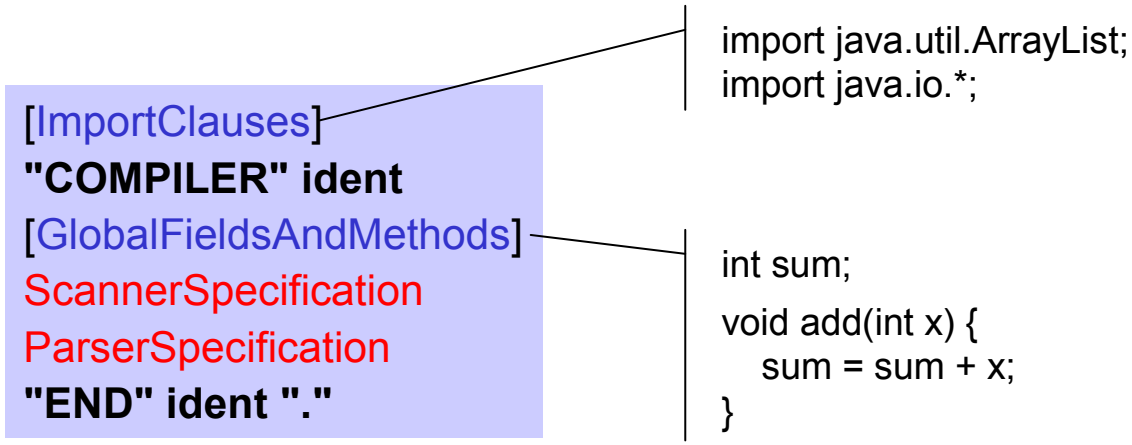

```
COMPILER Calc
CHARACTERS
   digit = '0' .. '9'.
   tab = '\t'. cr = '\r'. lf = '\n'.
TOKENS
   number = digit {digit}.
COMMENTS
   FROM "//" TO cr lf
   FROM "/*" TO "*/" NESTED
IGNORE tab cr lf
```

Scanner-Spezifikation

```
PRODUCTIONS
   Calc                          (. int x; .)
   = "CALC" Expr<out x>          (. System.out.println(x); .) .

   Expr <out int x>              (. int y; .)
   =  Term<out x>
      { '+' Term<out y>          (. x = x + y; .)
      }.

   Term <out int x>              (. int y; .)
   =  Factor<out x>
      { '*' Factor<out y>        (. x = x * y; .)
      }.

   Factor <out int x>
   =  number                     (. x = Integer.parseInt(t.val); .)
   |  '(' Expr<out x> ')'.
END Calc.
```

Parser-Spezifikation

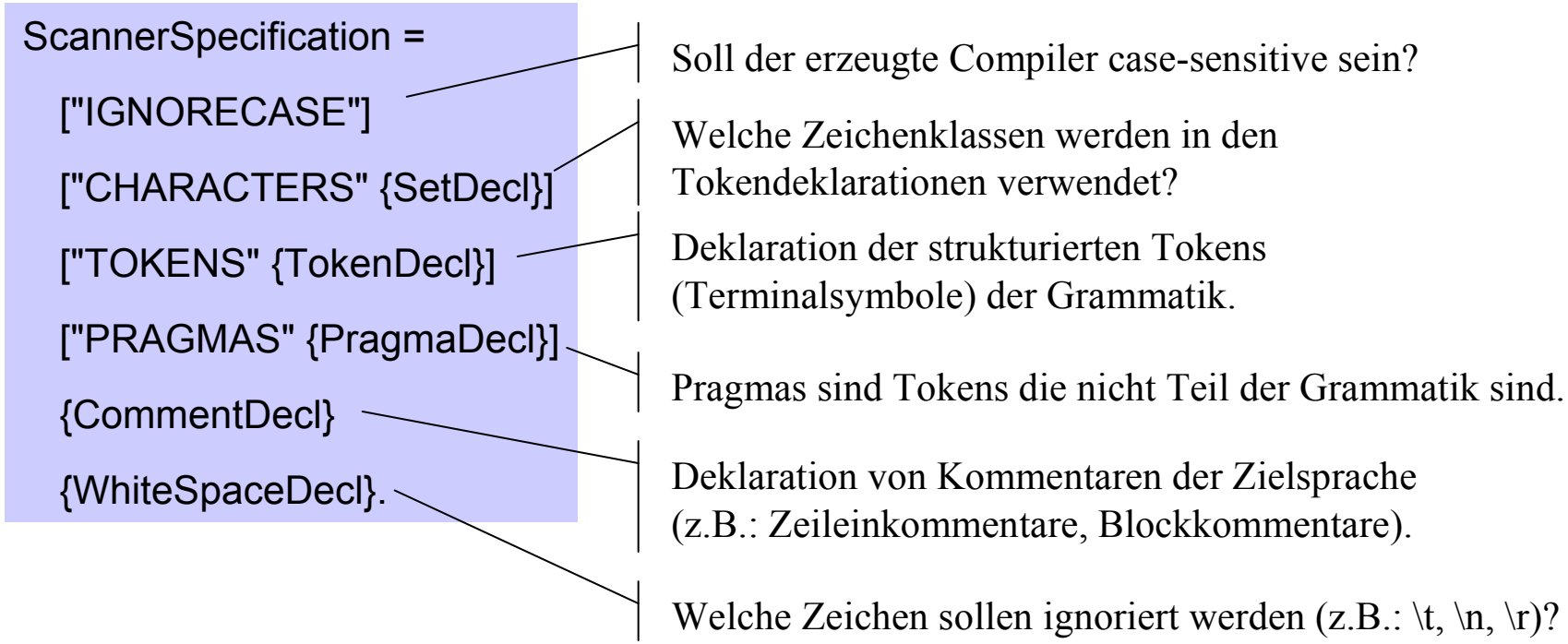
Aufbau einer Compiler Beschreibung



ident bezeichnet das Startsymbol (oberstes Nonterminal) der Grammatik

8. Erzeugen von Generatoren mit Coco/R

Aufbau der Scanner-Spezifikation



Zeichenklassen

Beispiel

```
CHARACTERS
   digit    = "0123456789".
   hexDigit = digit + "ABCDEF".
   letter   = 'A' .. 'Z'.
   eol      = '\n'.
   noDigit  = ANY - digit.
```

die Menge aller Ziffern
die Menge der Hexadezimalziffern
die Menge aller Großbuchstaben
das Zeilenende
alle Zeichen ausgenommen Ziffern

Gültige Escapesequenzen in Zeichenkonstanten und Strings

\\	backslash	\r	carriage return	\f	form feed
\'	apostrophe	\n	new line	\a	bell
\"	quote	\t	horizontal tab	\b	backspace
\0	null character	\v	vertical tab	\uxxxx	hex character value

Coco/R unterstützt Unicode (UTF-8)

Token-Deklarationen

Definieren der Struktor von *Tokenklassen* (z.B.: ident, number, ...)
Literale wie "while" oder ">=" müssen nicht deklariert werden.

Beispiel

```
TOKENS
   ident     =  letter {letter | digit | '_'}.
   number    =  digit {digit}
              |  "0x" hexDigit hexDigit hexDigit hexDigit.
   float     =  digit {digit} '.' digit {digit} ['E' ['+' | '-'] digit {digit}].
```

kein Problem wenn Tokens
mit dem selben Zeichen beginnen

- Die Rechte Seite muss ein regulärer EBNF-Ausdruck sein.
- Namen auf der rechten Seite bezeichnen Zeichenklassen.

Pragmas

Spezielle Tokens (z.B.: Compileroptionen)

- können überall im Quellcode vorkommen
- sind nicht Teil der Grammatik
- müssen semantisch verarbeitet werden.

Beispiel

```
PRAGMAS
   option = '$' {letter}.  (.  for (int i = 0; i < t.val.length(); i++) {
                                  char ch = t.val.charAt(i);
                                  if (ch == 'A') ...
                                  else if (ch == 'B') ...
                                  ...
                               } .)
```

wann immer eine *Option* (z.B.: $ABC) im Quellcode vorkommt wird diese semantische Aktion ausgeführt.

Typische Anwendungen

- Compileroptionen
- Preprozessor (z.B.: #ifdef)
- Kommentar-Verarbeitung
- Verarbeitung von Zeilenenden

Kommentare

Gesondert deklariert weil

- Geschachtelte Kommentare können nicht mit regulären Grammatiken beschrieben werden
- Müssen vom Parser ignoriert werden

Beispiel

```
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO "\r\n"
```

White Space und Case Sensitivity

White space

```
IGNORE   '\t' + '\r' + '\n'
```

Zeichenmenge

Leerzeichen werden standardmäßig ignoriert

Case sensitivity

Mit Coco/R erzeugte Compiler sind standardmäßig case-sensitiv

Können aber mit einem Schüsselwort case-insensiv gemacht werden `IGNORECASE`

```
COMPILER Sample
IGNORECASE
CHARACTERS
   hexDigit = digit + 'a'..'f'.
   ...
TOKENS
   number = "0x" hexDigit hexDigit hexDigit hexDigit.
   ...
PRODUCTIONS
   WhileStat = "while" '(' Expr ')' Stat.
   ...
END Sample.
```

Erkennt

- 0x00ff, 0X00ff, 0X00FF als *number*
- while, While, WHILE als keyword

Der Tokenwert der zum Parser geliefert wird behält die Originalschreibweise.

Schnittstelle des erzeugten Scanners


```
public class Scanner {
   public Buffer buffer;
   public Scanner (String fileName);
   public Scanner (InputStream s);
   public Token       Scan();
   public Token       Peek();
   public void        ResetPeek();
}
```

Hauptmethode, liefert einen Token pro Aufruf

liest von der aktuellen Scannerposition an weiter ohne die Tokens vom Eingabestrom zu entfernen.

setzt die Peek-Position an die aktuelle Scan-Position

```
public class Token {
   public int    kind;   // token kind (i.e. token number)
   public int    pos;    // token position in the source text (starting at 0)
   public int    col;    // token column (starting at 1)
   public int    line;   // token line (starting at 1)
   public String val;    // token value
}
```

8. Erzeugen von Generatoren mit Coco/R

Produktionen

- Können in beliebiger Reihenfolge vorkommen.
- Es muss genau eine Produktion für jedes Nonterminal geben.
- Es muss eine Produktion für das Startsymbol (Name der Grammatik) geben.

Beispiel

```
COMPILER Expr
   ...
PRODUCTIONS
   Expr      =  SimExpr [RelOp SimExpr].
   SimExpr   =  Term {AddOp Term}.
   Term      =  Factor {Mulop Factor}.
   Factor    =  ident | number | "-" Factor | "true" | "false".
   RelOp     =  "==" | "<" | ">".
   AddOp     =  "+" | "-".
   MulOp     =  "*" | "/".
END Expr.
```

Beliebige kontextfreie Grammatik in EBNF

Semantische Aktionen

Beliebiger Java/C#/… Code zwischen (. und .)

```
IdentList       (. int n; .)
= ident         (. n = 1; .)
  { ',' ident   (. n++; .)
  }             (. System.out.println(n); .)
  .
```

lokale semantische Deklaration

semantische Aktion

Semantische Aktionen werden von Coco/R ohne Prüfung in den Parser übernommen.

Globale semantische Aktionen

```
import java.io.*;
COMPILER Sample
   FileWriter w;
   void Open(string path) {
      w = new FileWriter(path);
      ...
   }
...
PRODUCTIONS
   Sample = ...      (. Open("in.txt"); .)
   ...
END Sample.
```

Import von Klassen und Paketen

Globale semantische Deklarationen (werden Felder und Methoden im Parser)

Semantische Aktionen können auf globale Deklarationen als auch importierte Klassen zugreifen.

Attribute

Von Terminalsymbolen

- Terminalsymbole haben keine expliziten Attribute
- Ihre Werte können in semantischen Aktionen mit den folgenden Variablen abgefragt werden

  Token **t**; der zuletzt erkannte Token

  Token **la**; der Lookahead-Token (noch nicht erkannt)

Beispiel

```
Factor <out int x> = number    (. x = Integer.parseInt(t.val); .)
```

```
class Token {
   int kind;       // token code
   String val;     // token value
   int pos;        // token position in the source text (starting at 0)
   int line;       // token line (starting at 1)
   int col;        // token column (starting at 1)
}
```

Von Nonterminalsymbolen

- NTS können beliebige Attribute haben

  Formal-Attr.: `A <int x, char c> = ... .` Aktual-Attr.: `... A <y, 'a'> ...`

- NTS können genau ein (Java Coco/R Limit) Ausgabe-Attribut haben (muss das erste in der Liste sein)

  `B <out int x, int y> = ... .` `... B <out z, 3> ...`

Produktionen in Parsmethoden übersetzen

Produktion

```
Expr<out int n>          (. int n1; .)
=  Term<out n>
   {  '+'
      Term<out n1>       (. n = n + n1; .)
   }.
```

Zugehörige Parsmethode

```
int Expr() {
   int n;
   int n1;
   n = Term();
   while (la.kind == 3) {
      Get();
      n1 = Term();
      n = n + n1;
   }
   return n;
}
```

Attribute => Parameter oder Rückgabewerte
Semantische Aktionen => Im Parscode eingebettet

Das Symbol ANY

Bezeichnet jeden Token der nicht in Konkurenz zu diesem ANY steht.

Beispiel: zählen wie oft *int* vorkommt

```
Type
=  "int"          (. intCounter++; .)
|  ANY.
```

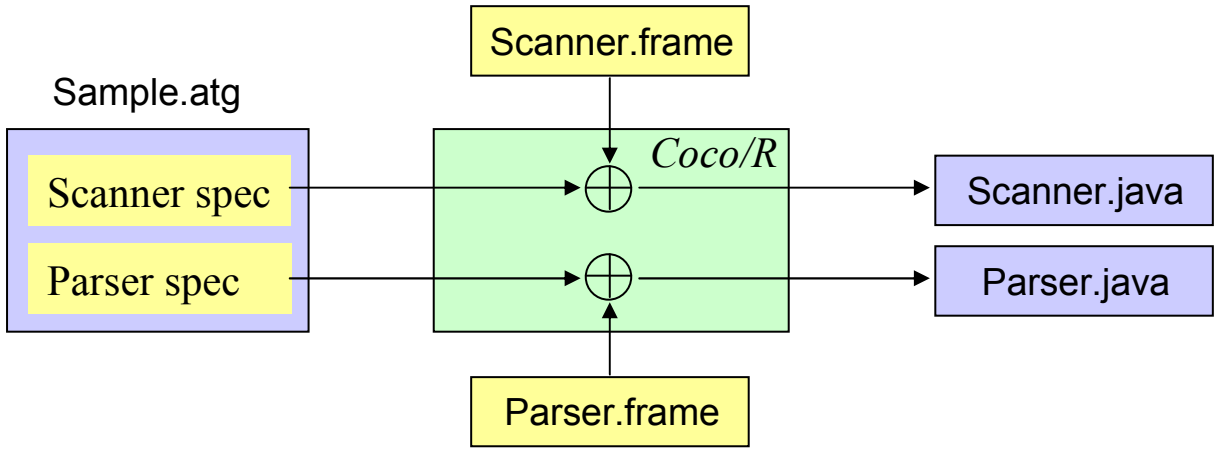
Jeder Token außer "int"

Beispiel: berechnen der Länge von semantischen Aktionen

```
SemAction<out int len>
=  "(."          (. int beg = t.pos + 2; .)
   { ANY }
   ".)"          (. len = t.pos - beg; .) .
```

Jeder Token außer ".)"

Framedatein



Scanner.frame Ausschnitt

```
public class Scanner {
   static final char EOL = '\n';
   static final int eofSym = 0;
-->declarations
   ...
   public Scanner (InputStream s) {
      buffer = new Buffer(s);
      Init();
   }
   void Init () {
      pos = -1; line = 1; …
-->initialization
   ...
}
```

- Coco/R fügt generierte Teile an mit "-->..." gekennzeichneten Positionen ein.
- Der Benutzer kann die Framedatein editieren, um den erzeugten Scanner und Parser anzupassen.
- Framedatein werden in dem Verzeichnis erwartet in dem sich die Compilerspezifikation (e.g. *Sample.atg*) befindet.

Schnittstelle des erzeugten Parsers


```
public class Parser {
   public Scanner scanner;   // the scanner of this parser
   public Errors  errors;    // the error message stream
   public Token t;           // most recently recognized token
   public Token la;          // lookahead token
   public Parser (Scanner scanner);
   public void Parse ();
   public void SemErr (String msg);
}
```

Parseraufruf im Hauptprogramm

```
public class MyCompiler {

   public static void main(String[] arg) {
      Scanner scanner = new Scanner(arg[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      Console.WriteLine(parser.errors.count + " errors detected");
   }
}
```

8. Erzeugen von Generatoren mit Coco/R

Syntax Fehlerbehandlung

Syntax Fehlermeldungen werden automatisch erzeugt

Für ungültige Terminalsymbole

Produktion	S = a b c.
Eingabe	a x c
Fehlermeldung	-- line ... col ...: b expected

Für ungültige Alternativen

Produktion	S = a (b | c | d) e.
Eingabe	a x e
Fehlermeldung	-- line ... col ...: invalid S

Fehlermeldungen können durch umschreiben der Grammatik verbessert werden

Produktionen	S = a T e. T = b | c | d.
Eingabe	a x e
Fehlermeldung	-- line ... col ...: invalid T

Syntax Fehlerbehandlung Wiederaufsatz

Der Benutzer muss Synchronisationspunkte zum Wiederaufsatz angeben.

Synchronisationspunkte

```
Statement
= SYNC
   (   Designator "=" Expr SYNC ';'
   |   "if" '(' Expression ')' Statement ["else" Statement]
   |   "while" '(' Expression ')' Statement
   |   '{' {Statement} '}'
   |   ...
   }.
```

Was passiert im Fehlerfall?

- Der Parser meldet einen Fehler.
- Der Parser fährt bis zum nächsten Synchronisationspunk fort.
- Der Parser übergeht die Eingabesymbole bis er ein passendes zu Synchronisation findet.

```
while (la.kind is not accepted here) {
   la = scanner.Scan();
}
```

Was sind gute Synchronisationspunkte?

Stellen in der Grammatik an denen besonders "sichere" Tokens erwartet werden.

- Anfang eines Statements: if, while, do, ...
- Anfang einer Deklaration: public, static, void, ...
- Vor einem Strichpunkt

Semantische Fehlerbehandlung

Muss in semantischen Aktionen geschehen

```
Expr<out Type type>          (.  Type type1; .)
=  Term<out type>
   {  '+' Term<out type1>    (.  if (type != type1) SemErr("incompatible types"); .)
   } .
```

Methode *SemErr* im Parser

```
void SemErr (String msg) {
   ...
   errors.SemErr(t.line, t.col, msg);
   ...
}
```

Errors Klasse

Coco/R erzeugt eine Klasse für die Fehlerausgabe

```
public class Errors {

   public int count = 0;                                    // number of errors detected
   public PrintStream errorStream = System.out;             // error message stream
   public String errMsgFormat = "-- line {0} col {1}: {2}"; // 0=line, 1=column, 2=text

   // called by the programmer (via Parser.SemErr) to report semantic errors
   public void SemErr (int line, int col, String msg) {
      printMsg(line, col, msg);
      count++;
   }

   // called automatically by the parser to report syntax errors
   public void SynErr (int line, int col, int n) {
      String msg;
      switch (n) {
         case 0: msg = "..."; break;
         case 1: msg = "..."; break;
         ...
      }
      printMsg(line, col, msg);
      count++;
   }
   ...

}
```

syntax error messages generated by Coco/R

8. Erzeugen von Generatoren mit Coco/R

Coco/R findet LL(1) Konflikte automatisch

Beispiel

```
...
PRODUCTIONS
   Sample     =  {Statement}.
   Statement  =  Qualident '=' number ';'
                 |  Call
                 |  "if" '(' ident ')' Statement ["else" Statement].
   Call       =  ident '(' ')' ';'.
   Qualident  =  [ident '.'] ident.
   ...
```

Coco/R erzeugt folgende Warnungen

```
>coco Sample.atg
Coco/R (Sep 19, 2006)
checking
   Sample deletable
   LL1 warning in Statement: ident is start of several alternatives
   LL1 warning in Statement: "else" is start & successor of deletable structure
   LL1 warning in Qualident: ident is start & successor of deletable structure
parser + scanner generated
0 errors detected
```

Konfliktlösung durch Multi-Symbol Lookahead


```
A = ident (. x = 1; .) {',' ident (. x++; .) } ':'
  | ident (. Foo(); .) {',' ident (. Bar(); .) } ';'.
```

LL(1) Konflikt

Lösung

```
A = IF (FollowedByColon())
      ident (. x = 1; .) {',' ident (. x++; .) } ':'
  | ident (. Foo(); .) {',' ident (. Bar(); .) } ';'.
```

Methode zum Lösen des Konflikts

```
bool FollowedByColon() {
   Token x = la;
   while (x.kind == _ident || x.kind == _comma) {
      x = scanner.Peek();
   }
   return x.kind == _colon;
}
```

```
TOKENS
   ident = letter {letter | digit} .
   comma = ','.
   ...
```

```
static final int
   _ident = 17,
   _comma = 18,
   ...
```

Konfliktlösung durch Semantische Information


```
Factor = '(' ident ')' Factor      /* type cast */
       | '(' Expr ')'              /* nested expression */
       | ident | number.
```

LL(1) Konflikt

Lösung

```
Factor = IF (IsCast())
           '(' ident ')' Factor    /* type cast */
       | '(' Expr ')'              /* nested expression */
       | ident | number.
```

Methode zum Lösen des Konflikts

```
bool IsCast() {
   Token next = scanner.Peek();
   if (la.kind == _lpar && next.kind == _ident) {
      Obj obj = SymTab.Find(next.val);
      return obj != null && obj.kind == TYPE;
   } else return false;
}
```

Liefert *true* wenn auf '(' ein deklarierter Name folgt.

8. Erzeugen von Generatoren mit Coco/R

Beispiel: Query Form Generator

Eingabe: Domain-spezifische Sprache zur Beschreibung von Formularen

```
RADIO "How did you like this course?"
("very much", "much", "somewhat", "not so much", "not at all")
CHECKBOX "What is the field of your study?"
("Computer Science", "Mathematics", "Physics")
TEXTBOX "What should be improved?"
...
```

Ausgabe: HTML-Formular

How did you like this course?
- ○ very much
- ○ much
- ○ somewhat
- ○ not so much
- ○ not at all

What is the field of your study?
- ☐ Computer Science
- ☐ Mathematics
- ☐ Physics

What should be improved?

To do

1. Eingabe durch Grammatik beschreiben
2. Definieren der Attribute
3. Definieren der semantischen Methoden
4. Schreiben der ATG

Eingabe Grammatik


```
QueryForm   = {Query}.
Query       = "RADIO" Caption Values
            | "CHECKBOX" Caption Values
            | "TEXTBOX" Caption.
Values      = '(' string {',' string} ')'.
Caption     = string.
```

```
RADIO "How did you like this course?"
("very much", "much", "somewhat",
 "not so much", "not at all")

CHECKBOX "What is the field of your study?"
("Computer Science", "Mathematics", "Physics")

TEXTBOX "What should be improved?"
```

Attribute

- Caption liefert einen String — Caption<out String s>
- Values eine Liste von Strings — Values<out ArrayList list>

Semantische Methoden

- printPrologue()
- printEpilogue()
- printRadio(caption, values)
- printCheckbox(caption, values)
- printTextbox(caption)

} implementiert in der Klasse HtmlGenerator

Scanner-Spezifikation


```
COMPILER QueryForm

CHARACTERS
   noQuote = ANY - "".
   tab = '\t'.
   cr = '\r'.
   lf = '\n'.

TOKENS
   string = "" {noQuote} "".

COMMENTS
   FROM "//" TO cr lf

IGNORE tab+cr+lf

...

END QueryFormGenerator.
```

Parser-Spezifikation


```
import java.util.ArrayList;
COMPILER QueryFormGenerator
   HtmlGenerator html;
   ...
PRODUCTIONS
QueryFormGenerator =              (. html.printPrologue(); .)
   { Query }                      (. html.printEpilogue(); .) .
//-----------------------------------------------------------------------------
Query                             (. String caption; ArrayList values; .)
= "RADIO" Caption<out caption> Values<out values>
                                  (. html.printRadio(caption, values); .)
|  "CHECKBOX" Caption<out caption> Values<out values>
                                  (. html.printCheckbox(caption, values); .)
|  "TEXTBOX" Caption<out caption>
                                  (. html.printTextbox(caption); .) .
//-----------------------------------------------------------------------------
Caption<out String s> = StringVal<out s>.
//-----------------------------------------------------------------------------
Values<out ArrayList values>      (. String s; .)
= '(' StringVal<out s>            (. values = new ArrayList(); values.add(s); .)
   { ',' StringVal<out s>         (. values.add(s); .)
   }
   ')'.
//-----------------------------------------------------------------------------
StringVal<out String s>
= string                          (. s = t.val.substring(1, t.val.length()-1); .) .
END QueryFormGenerator.
```

Klasse HtmlGenerator (1/2)


```
import java.io.*;
import java.util.ArrayList;

class HtmlGenerator {
   PrintStream s;
   int itemNo = 0;

   public HtmlGenerator(String fileName) throws FileNotFoundException {
      s = new PrintStream(fileName);
   }

   public void printPrologue() {
      s.println("<html>");
      s.println("<head><title>Query Form</title></head>");
      s.println("<body>");
      s.println("  <form>");
   }

   public void printEpilogue() {
      s.println("  </form>");
      s.println("</body>");
      s.println("</html>");
      s.close();
   }
   ...
```

Klasse HtmlGenerator (2/2)


```
    public void printRadio(String caption, ArrayList values) {
        s.println(caption + "<br>");
        for (int i = 0; i < values.size(); i++) {
            s.print("<input type='radio' name='Q" + itemNo + "' ");
            s.print("value='" + values.get(i) + "'>" + values.get(i) + "<br>");
            s.println();
        }
        itemNo++; s.println("<br>");
    }

    public void printCheckbox(String caption, ArrayList values) {
        s.println(caption + "<br>");
        for (int i = 0; i < values.size(); i++) {
            s.print("<input type='checkbox' name='Q" + itemNo + "' ");
            s.print("value='" + values.get(i) + "'>" + values.get(i) + "<br>");
            s.println();
        }
        itemNo++; s.println("<br>");
    }

    public void printTextbox(String caption) {
        s.println(caption + "<br>");
        s.println("<textarea name='Q" + itemNo + "' cols='50' rows='3'></textarea><br>");
        itemNo++; s.println("<br>");
    }
}
```

```
<input type='radio' name='Q0'
    value='very much'>very much<br>
```

```
<input type='checkbox' name='Q1'
    value='Mathematics'>Mathematics<br>
```

```
<textarea name='Q2' cols='50' rows='3'>
</textarea><br>
```

Hauptprogramm

Aufgaben

- Lesen der Kommandozeilen-Argumente
- Erzeugen und initialisieren eines Scanners und eines Parsers
- Starten des Parsers

```
import java.io.*;
class QueryFormGenerator {
   public static void main(String[] args) {
      String inFileName = args[0];
      String outFileName = args[1];
      Scanner scanner = new Scanner(inFileName);
      Parser parser = new Parser(scanner);
      try {
         parser.html = new HtmlGenerator(outFileName);
         parser.Parse();
      } catch (FileNotFoundException e) {
         System.out.println("-- file " + outFileName + " not found");
      }
   }
}
```

Alles zusammen

Ausführen von Coco/R

```
java -jar Coco.jar QueryFormGenerator.ATG
```

Alles kompilieren

```
javac Scanner.java Parser.java HtmlGenerator.java QueryFormGenerator.java
```

Ausführen des Query-Form-Generators

```
java QueryFormGenerator input.txt output.html
```

Zusammenfassung

Compilergenerator-Werkzeuge wie Coco/R können angewandt werden wenn

- Eingabedaten gelesen/transformiert werden müssen
- Die Eingabe strukturiert ist

Typische Anwendungen

- Statische Quellcode-Analysatoren
- Quellcode-Metrik-Tools
- Quellcode Instrumentierung
- Domainspezifische Sprachen
- Log-Datei-Analysatoren
- Datenstrom-Verarbeitung
- ...