VISUAL COMPILER COMPILER

Sprachübersetzer anhand eines Beispiels

Inhalt

Zielstellung

- Für einen in der Programmiersprache PL/0 geschrieben Programm-Quelltext soll eine lexikalische und syntaktische Analyse durchgeführt werden.
- Der „**Scanner**“ übernimmt die lexikalische Analyse und gibt sein Ergebnis an den „**Parser**“ weiter, welcher die syntaktische Analyse übernimmt.
- Als Ergebnis soll eine entsprechende Ausgabe erzeugt werden, ob es sich um einen gültigen Quelltext in der Sprache PL/0 handelt.

Vorüberlegungen

- Ein Satz der deutschen Sprache besteht aus mehreren Teilen, die nur in einer bestimmten Reihenfolge verwendet werden dürfen.
  Ein Beispiel:
  - Artikel Subjekt Verb Adjektiv Satzzeichen
    Der Hund bellt laut. ✓ ein gültiger Satz
    Hund laut der . bellt × kein gültiger Satz
- Um entscheiden zu können, ob der zu betrachtende Quelltext Fehler enthält, müssen wir die Sprache PL/0 und ihren Aufbau betrachten.

Die Sprache PL/0

- **PL/0** ist eine vereinfachte Programmiersprache.
- Sie dient als Muster, um im Buch Compilerbau von *Niklaus Wirth* zu zeigen, wie man einen Compiler herstellt.
- Die Sprache kann nur mit Zahlenwerten umgehen und ist nicht dazu gedacht, wirklich eingesetzt zu werden.
- Ist Pascal sehr ähnlich.

Syntaxregeln für PL/0


```
program    → block .
block      → const1  var1  proc1 statement
const1     → CONST ident = number const2 ;  | ε
const2     → , ident = number const2  | ε
var1       → VAR ident var2 ;  | ε
var2       → , ident var2  | ε
proc1      → proc2 proc1  | ε
proc2      → PROCEDURE ident ; block ;
statement  → ident := expression
           | CALL ident
           | ? ident
           | ! expression
           | BEGIN statement statements END
           | IF condition THEN statement
           | WHILE condition DO statement
           | ε
statements → ; statement statements | ε
```

```
condition  → ODD expression
           | expression = expression
           | expression # expression
           | expression < expression
           | expression <= expression
           | expression >= expression
           | expression > expression
expression → + term expblock
           | - term expblock
           | term expblock
expblock   → + term expblock
           | - term expblock
           | ε
term       → factor termblock
termblock  → * factor termblock
           | / factor termblock
           | ε
factor     →  ident
           | number
           | ( expression )
```

- Nichterminale der Grammatik
- Sprachelemente die in PL/0 Quelltexten verwendet werden (*Terminale*)
- ident und number sind hier nicht weiter zerlegt, werden aber später mit regulären Ausdrücken beschrieben.

Der „**Scanner**“

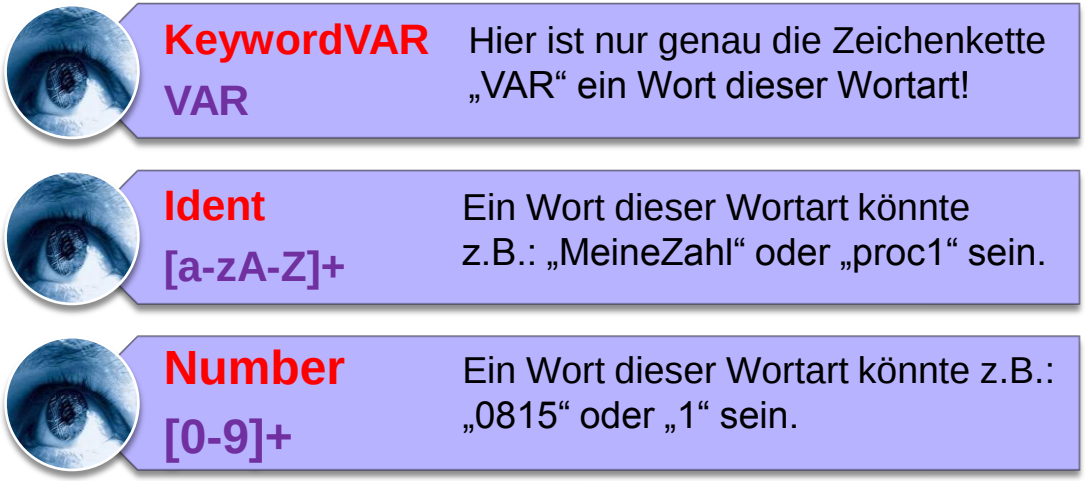
- Der Scanner übernimmt das Einlesen unseres Quelltextes und das Bestimmen der „*Wortarten*“.
- In der deutschen Sprache können wir sagen:
  der, die, das sind **Artikel** oder
  gehen, laufen, springen,… sind **Verben**
- Bei der Betrachtung der Syntaxregeln für PL/0 haben wir festgestellt, dass es eine Reihe von Schlüsselworten gibt. (blau gekennzeichneten Teile)
- Wie können auch hier mehrere dieser Wörter zu einer Wortart zusammenfassen:
  "**+**" und "**-**" sind **Strichrechnungszeichen** und
  "*****" und "**/**" sind **Punktrechnungszeichen**

Der „**Scanner**“

- Der Scanner liest den gesamten Quelltext ein und bestimmt für alle gefundenen Wörter die entsprechende Wortart.
- Dabei legt er eine Liste mit Paaren an. Ein Paar enthält jeweils das gefundene Wort und die entsprechende Wortart.
  Beispiel: […, ("**+**", Strichrechnungszeichen), …]
- Ein solches Paar aus Wort und Wortart nennen wir ab jetzt „Token“. Die Liste aller Paare heißt „Tokenliste“.

Arbeitsweise eines Scanners

- Die Wortarten werden mit sogenannten „Pattern“ beschrieben.
- Pattern sind meist reguläre Ausdrücke und alle Wörter die mit diesem Ausdrück beschrieben werden können, gehören zur Wortart.
- Eine Wortart besitzt jeweils einen Wortartnamen (Tokenname) und ein Pattern.



Arbeitsweise eines Scanners

- Wie bestimmt man nun die Wortart eines Wortes und was ist eigentlich ein Wort?
- Ein Wort ≙ eine zusammenhängende Zeichenfolge
- Zunächst ein Beispiel:
  - Ich esse gerade.
  - Ich habe mich vermessen.
    → zweimal die gleiche Zeichenfolge
- Worte werden durch Leerzeichen (oder andere spezielle Zeichen wie „. , !“) voneinander getrennt!
- In der Welt des Computers kann auch ein Zeilenumbruch oder Tabulator ein Trennzeichen sein.

Arbeitsweise eines Scanners

- Ein Stück Quelltext in PL/0:
  - …
    PROCEDURE proc1;
    VAR VARIABLE1; ← Was ist was?
    BEGIN
     VARIABLE1 := 10;
    END;
    …
  - Hier ist das unterstrichene Schlüsselwort VAR auch ein Teil eines anderen Wortes VARIABLE1.
  - Scanner könnte zwei mögliche Lösungen für die 2. Zeile ausgeben:

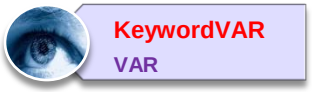
    ("**VAR**", KeywordVAR), ("**VARIABLE1**", Ident) oder

    ("**VAR**", KeywordVAR), ("**VAR**", KeywordVAR), ("**IABLE1**", Ident)

Arbeitsweise eines Scanners

- Wie liest ein Scanner den Eingabetext (Quelltext)?
- Eingabeband (wie bei Endlichen Automaten):

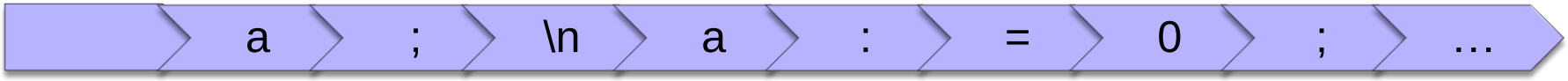
V	A	R		a	;	\n	a	:	=	0	;	…

- Alle Wortart-Pattern werden auf den Anfang das Bandes angewendet und es wird geprüft, welche passen.
- Das Pattern „VAR“ für KeywordVAR würde die ersten 3 Zeichen treffen → Pattern 

passt!
- Wenn mehrere Pattern passen, müssen Entscheidungsregeln angewendet werden → nur eine Wortart ist möglich.
- Passt kein Pattern, liegt ein lexikalischer Fehler vor!

Arbeitsweise eines Scanners

- Angenommen, wir haben uns für die Wortart KeywordVAR entschieden.
- Eingabeband verändert sich (erkannter Teil wird vorn abgeschnitten):

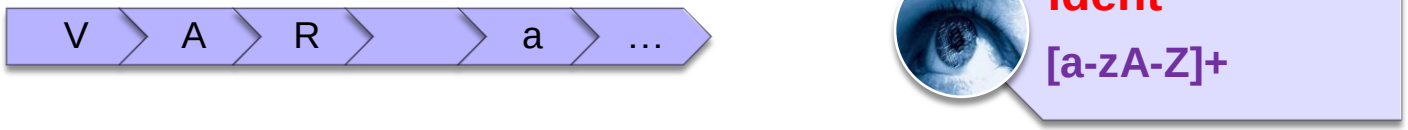


- An die Tokenliste des Scanners wird ein Token der Form ("VAR", KeywordVAR) angehängt.
- Wieder werden alle Pattern auf den verbleibenden Eingabestrom angewendet → solange bis das Band vollständig abgearbeitet ist!

Arbeitsweise eines Scanners

Was wenn mehrere Pattern passen?

- In unserem Beispiel hätte auch „Ident“ auf den Anfang des Bandes gepasst und würde ebenfalls die ersten 3 Zeichen treffen:



- Regeln bei mehreren passenden Pattern:
  - Wenn zwei oder mehr Pattern passen, wird für die Wortart entschieden, bei der die meisten Eingabezeichen getroffen werden.
  - Wenn zwei oder mehr Pattern passen und alle die gleiche Anzahl von Zeichen treffen, wird für die Wortart entschieden, die in der Wortartliste am weitesten oben steht.

Arbeitsweise eines Scanners

Besonderheit von irrelevanten Zeichen:

- Einige Zeichen sind in Programmiersprachen typischerweise nicht wichtig, sondern dienen nur der besseren Lesbarkeit:

BEGIN **!5;** **END.**	**BEGIN!5;END.**

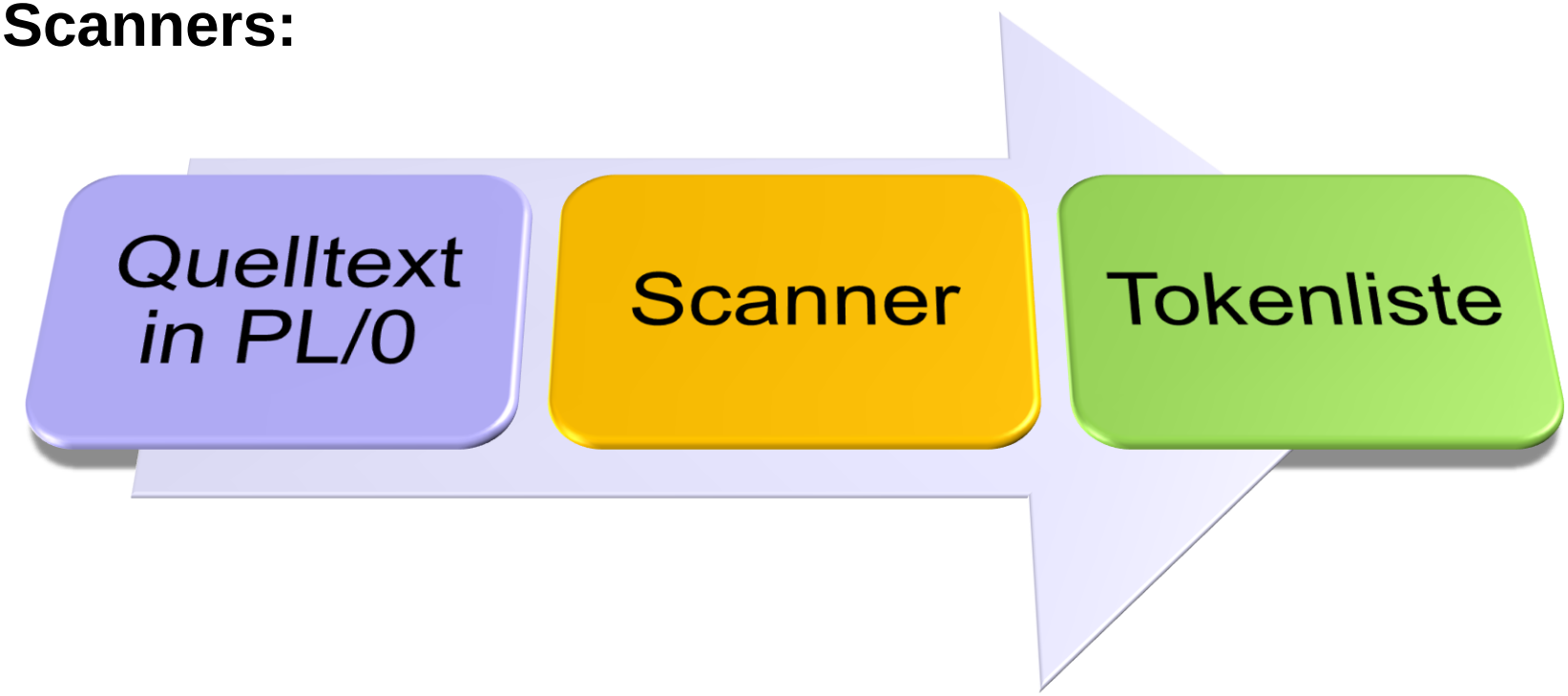
Beide Quellcodes sind gültige PL/0 Programme.

- Leerzeichen, Zeilenumbrüche, Tabs sind nicht Bestandteil der Sprache PL/0!

→ Scanner filtert diese Zeichen heraus und schreibt sie nicht mit in die Tokenliste

Arbeitsweise eines Scanners

Ein- und Ausgabe des Scanners:

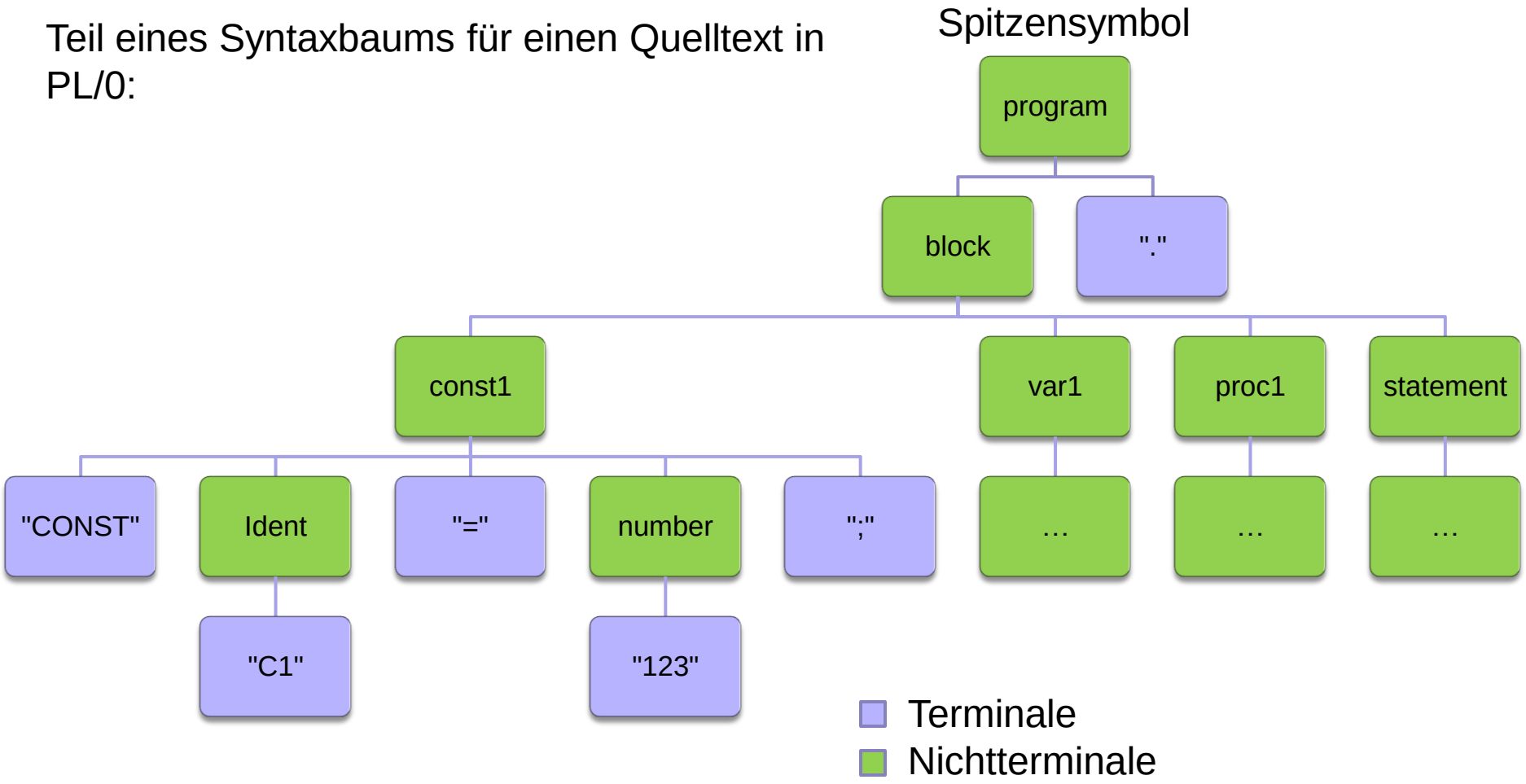


Arbeitsweise des Parsers

- Der Parser verarbeitet die Tokenliste des Scanners.
- Die Syntaxregeln der Sprache werden im Parser verwendet, um zu entscheiden, ob die Eingabe syntaktisch korrekt ist.
- Dabei wird versucht, die Nichtterminale der Grammatik vom Spitzensymbol bis hin zu den jeweiligen Terminalen abzuleiten.
- Die angewendeten Syntaxregeln für eine konkrete Tokenliste lassen sich in Form eines Baumes darstellen.

Arbeitsweise des Parsers

Teil eines Syntaxbaums für einen Quelltext in PL/0:



Arbeitsweise des Parsers

Wie funktioniert die Ableitung:

- Gegeben sei das Programm:
  BEGIN ! 5; END.
- Der Scanner produziert uns eine Tokenliste der Form:
  [("**BEGIN**", KeywordBEGIN), ("**!**", Output),
  ("**5**", Number), ("**;**", Semikolon),
  ("**END**", KeywordEND), ("**.**", EndPoint)]

program	→ block "."
block	→ const1 var1 proc1 statement
const1	→ "CONST" ident "=" number const2 ";" | ε
const2	→ "," ident "=" number const2 | ε
var1	→ "VAR" ident var2 ";" | ε
var2	→ "," ident var2 | ε
…	

Arbeitsweise des Parsers

Wie funktioniert die Ableitung:

- Der Parser beginnt mit die Syntaxregeln am Spitzensymbol program anzuwenden:
- program wird zu block .
  block . wird zu const1 var1 proc1 statement .
  Nun wird versucht const1 abzuleiten mit der ersten Möglichkeit:
  const1 var1 proc1 statement .
  wird zu **CONST** ident **=** number var1 …
  Jetzt steht ein Terminal ganz links und wird mit der Tokenliste verglichen. Da wir aber ein "**BEGIN**" und kein "**CONST**" in der Liste haben, kann diese Ableitung nicht funktionieren. →

Alternativen versuchen!

program	→ block **.**
block	→ const1 var1 proc1 statement
const1	→ **CONST** ident **=** number const2 **;** | ε
…	

Arbeitsweise des Parsers

- program wird zu block .
  block . wird zu const1 var1 proc1 statement .

  Nun versuchen wir const1 mit der zweiten Möglichkeit abzuleiten:
  const1 var1 proc1 statement . wird zu ε var1 proc1 statement .

  Mit Epsilon haben wir keine Probleme, da wir erstmal kein Terminal ganz links stehen haben, was nicht zur Tokenliste passen würde. Wir entscheiden uns also für diese Ableitung (dies könnten wir wieder im Syntaxbaum darstellen).

- Als nächstes könnten wir var1 auf die gleiche Art und Weise ableiten.

program	→ block .
block	→ const1 var1 proc1 statement
const1	→ **CONST** ident **=** number const2 **;** | ε
const2	→ **,** ident **=** number const2 | ε
var1	→ **VAR** ident var2 **;** | ε
…	

Arbeitsweise des Parsers

- Tokenliste zur Erinnerung:
  [("**BEGIN**", KeywordBEGIN), ("**!**", Output),
  ("**5**", Number), ("**;**", Semikolon),
  ("**END**", KeywordEND), ("**.**", EndPoint)]
- Da var1 und proc1 auch nur zu ε abgeleitet werden können (da weder „**VAR**“ noch „**PROCEDURE**“ am Anfang unserer Tokenliste stehen), bleibt nur noch statement als Nichtterminal übrig:
- ε var1 proc1 statement **.** wird zu ε ε ε statement **.**

program	→ block **.**
block	→ const1 var1 proc1 statement
const1	→ **CONST** ident **=** number const2 **;** | ε
const2	→ **,** ident **=** number const2 | ε
var1	→ **VAR** ident var2 **;** | ε
var2	→ **,** ident var2 | ε
proc1	→ proc2 proc1 | ε
proc2	→ **PROCEDURE** ident **;** block **;**
…	

Arbeitsweise des Parsers

- Tokenliste zur Erinnerung:
  [("**BEGIN**", KeywordBEGIN), ("**!**", Output), ("**5**", Number),
  (";", Semikolon), ("**END**", KeywordEND), (".", EndPoint)]
- program wird zu block **.**
  wird zu const1 var1 proc1 statement **.**
  wird zu statement **.** (ε ε ε weggestrichen)
- Wir versuchen nun statement abzuleiten. Erst bei der 5. Alternative laden wir einen Treffer:
  statement **.** wird zu **BEGIN** statement statements **END .**
- Jetzt haben wir ein Terminal welches zum Token in der Token-liste passt. "**BEGIN**" wird in der Ableitung gestrichen und das Token aus der Tokenliste entfernt.

statement → ident **:=** expression
| **CALL** ident
| **?** ident
| **!** expression
| **BEGIN** statement statements **END**
| **IF** condition **THEN** statement
| **WHILE** condition **DO** statement
| ε

Arbeitsweise des Parsers

- Tokenliste zur Erinnerung:
  [("**BEGIN**", KeywordBEGIN), ("**!**", Output), ("**5**", Number),
  (";", Semikolon), ("**END**", KeywordEND), (".", EndPoint)]
- program wird zu block **.**
  wird zu const1 var1 proc1 statement **.**
  wird zu statement **.** (ε ε ε weggestrichen)
- Wir versuchen nun statement abzuleiten. Erst bei der 5. Alternative laden wir einen Treffer:
  statement **.** wird zu **BEGIN** statement statements **END .**
- Jetzt haben wir ein Terminal welches zum Token in der Tokenliste passt. "**BEGIN**" wird in der Ableitung gestrichen und das Token aus der Tokenliste entfernt.

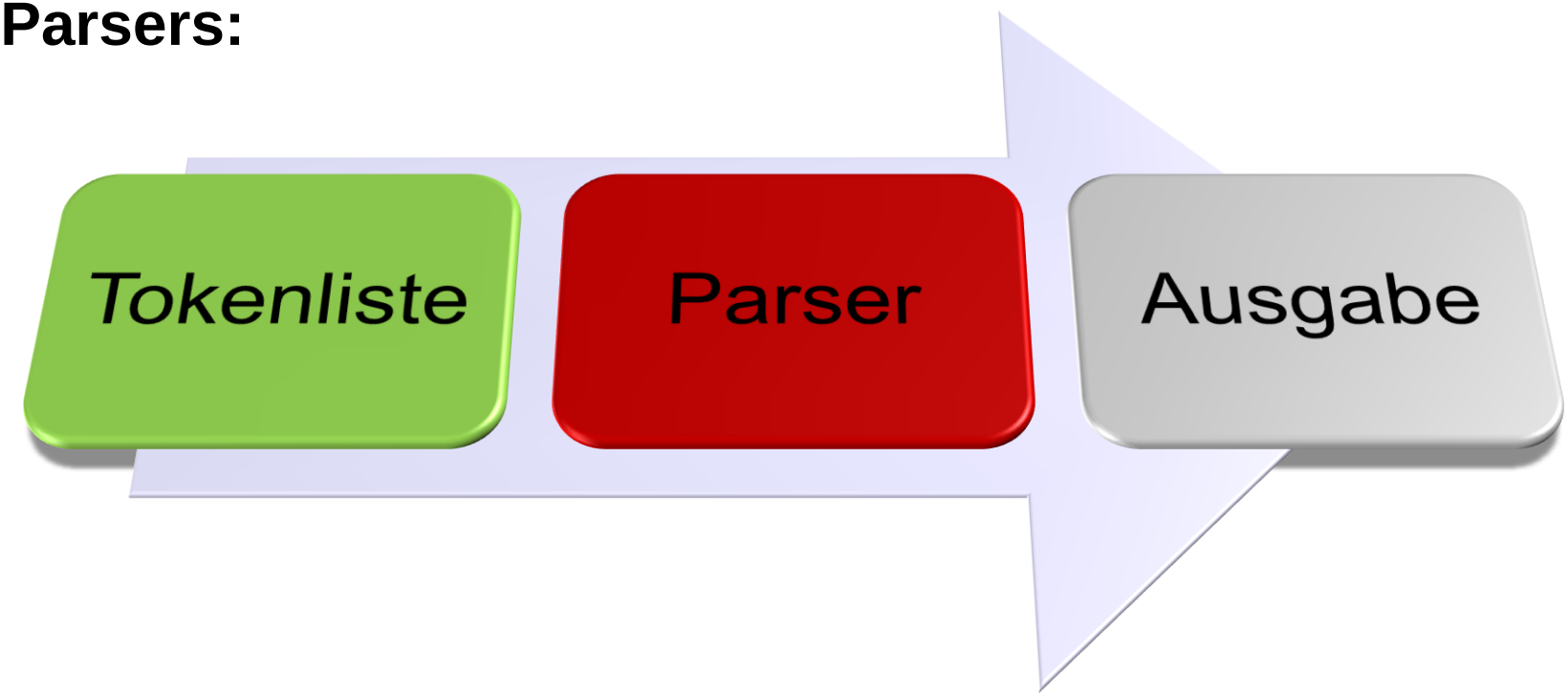
statement → ident **:=** expression
| **CALL** ident
| **?** ident
| **!** expression
| **BEGIN** statement statements **END**
| **IF** condition **THEN** statement
| **WHILE** condition **DO** statement
| ε

Arbeitsweise des Parsers

- Diese Ableitung wird so lange weitergeführt bist das Spitzensymbol vollständig in Terminale aufgelöst ist und die Tokenliste leer ist.
- Sollte es in einem Schritt keine Regel geben, die mit der Tokenliste zusammenpasst, liegt ein syntaktischer Fehler vor!

Arbeitsweise des Parsers

Ein- und Ausgabe des Parsers:

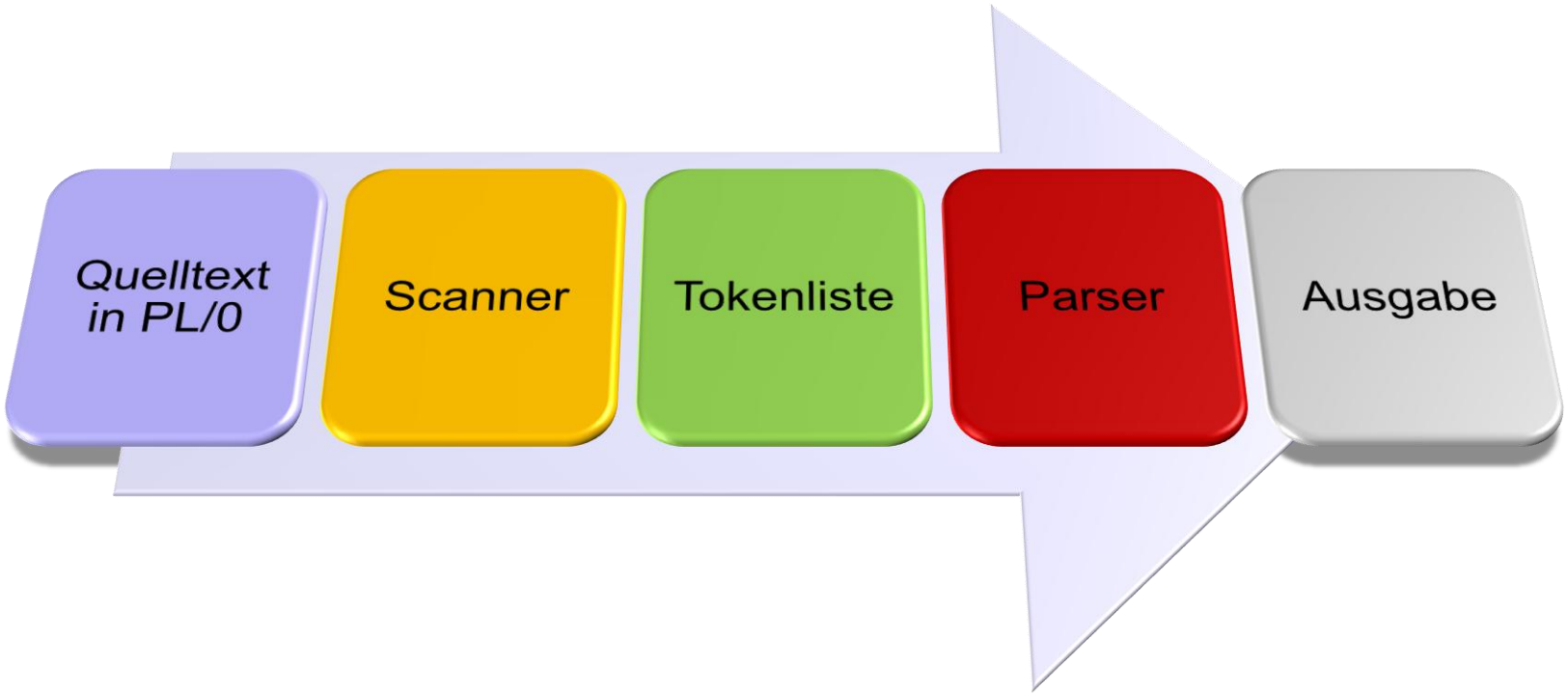


Der Compiler

- Ein Compiler stellt nun einen Verbund aus einem Scanner und einem Parser dar.
- Bei einem Compiler gibt es zusätzlich einen **Zielcodegenerator**.
- Ein Compiler soll ja nicht nur „true“ (für gültig) oder „false“ (für ungültig) ausgeben wenn ihm ein Quelltext vorgelegt wird, sondern soll einen neuen Quelltext in einer anderen Sprache liefern (z.B.: Java-Compiler erzeugt aus Javaquelltext eine Java-Bytecode-Datei)

Der Compiler

- Gesamtprozess:



Der Compiler

- Unsere Zielstellung war:
  - Als Ergebnis soll eine entsprechende Ausgabe erzeugt werden, ob es sich um einen gültigen Quelltext in der Sprache PL/0 handelt.
- Wir definieren also eine einfache Zielsprache „S_{out}“ für unseren Compiler welche genau einen Satz als Terminal kennt:
  S → **Es handelt sich um ein gültiges PL/0 Programm**
- Warum gibt es keinen Satz „Es handelt sich nicht um ein…“ in unser Zielsprache?

Sobald ein syntaktischer Fehler auftritt, wird der Compilervorgang gestoppt und es ist keine Zielcodeausgabe mehr möglich.

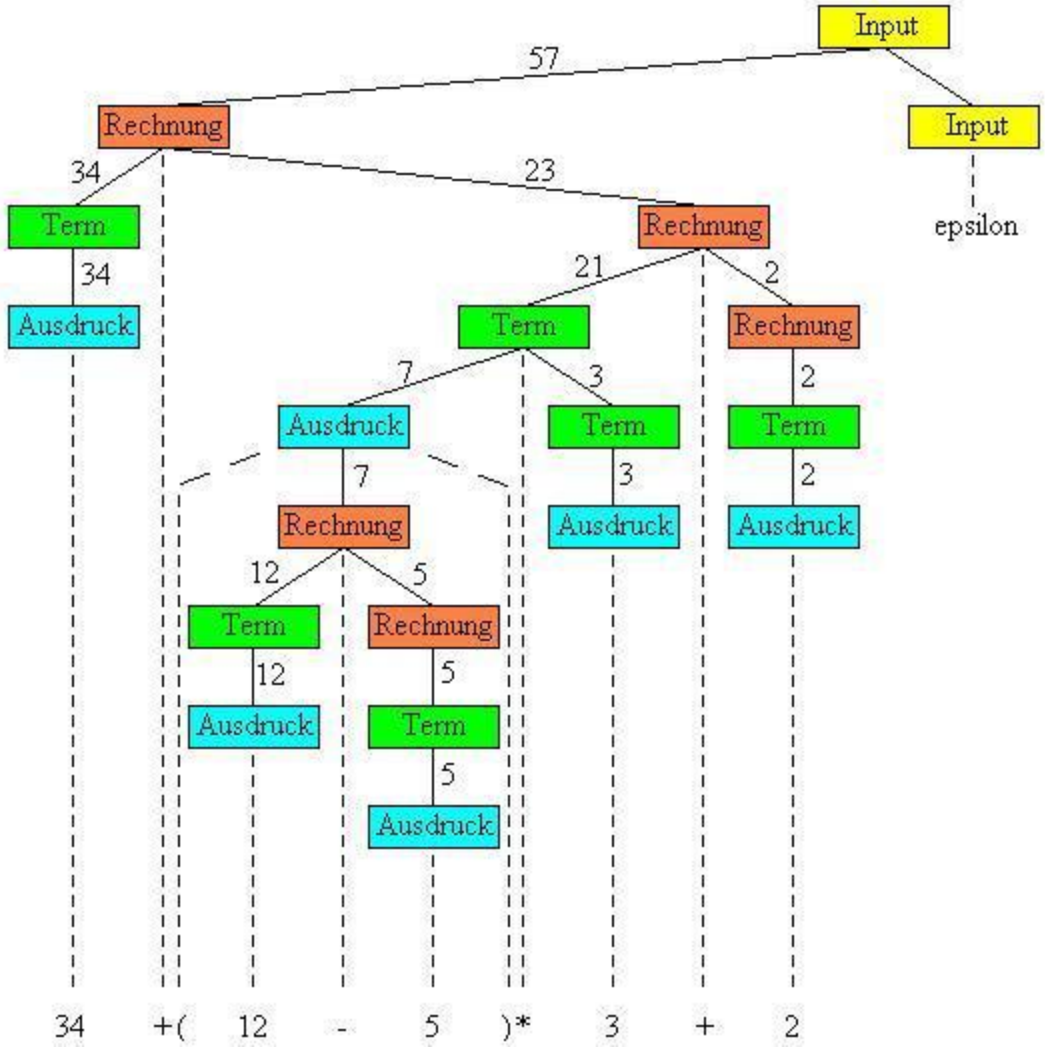
Der Compiler

- In vielen Compiler Generatoren wird der **Zielcodegenerator** mit in den Parser aufgenommen.
- Man verwendet dabei eine attributierte Grammatik um für jede angewendete Regel (bei der Ableitung) eine bestimmte Ausgabe zu erzeugen.
- Die einzelnen Ausgaben werden dabei im Syntaxbaum von unten nach oben hochgereicht und können dann auf höherer Ebene wieder durch Konkatenation verbunden werden.

Der Compiler

- Ein Ableitungsbaum eines arithmetischen Ausdrucks: 34+(12-5)*3+2

- Hinweis:
  Die verwendete Grammatik soll hier nicht betrachtet werden.
  Das „Hinaufreichen“ der Ausgaben ist zu erkennen.



Der Compiler

- Für unsere PL/0 Sprache reicht es also aus, wenn wir dem Spitzensymbol „program“ ein solches Ausgabe-Attribut geben.
- program soll "Es handelt sich um ein gültiges PL/0 Programm" ausgeben .

- Nachdem wir nun Scanner, Parser und Compiler für PL/0 → „S_{out}“ betrachtet haben, können wir diesen im Visual Compiler Compiler implementieren. ☺

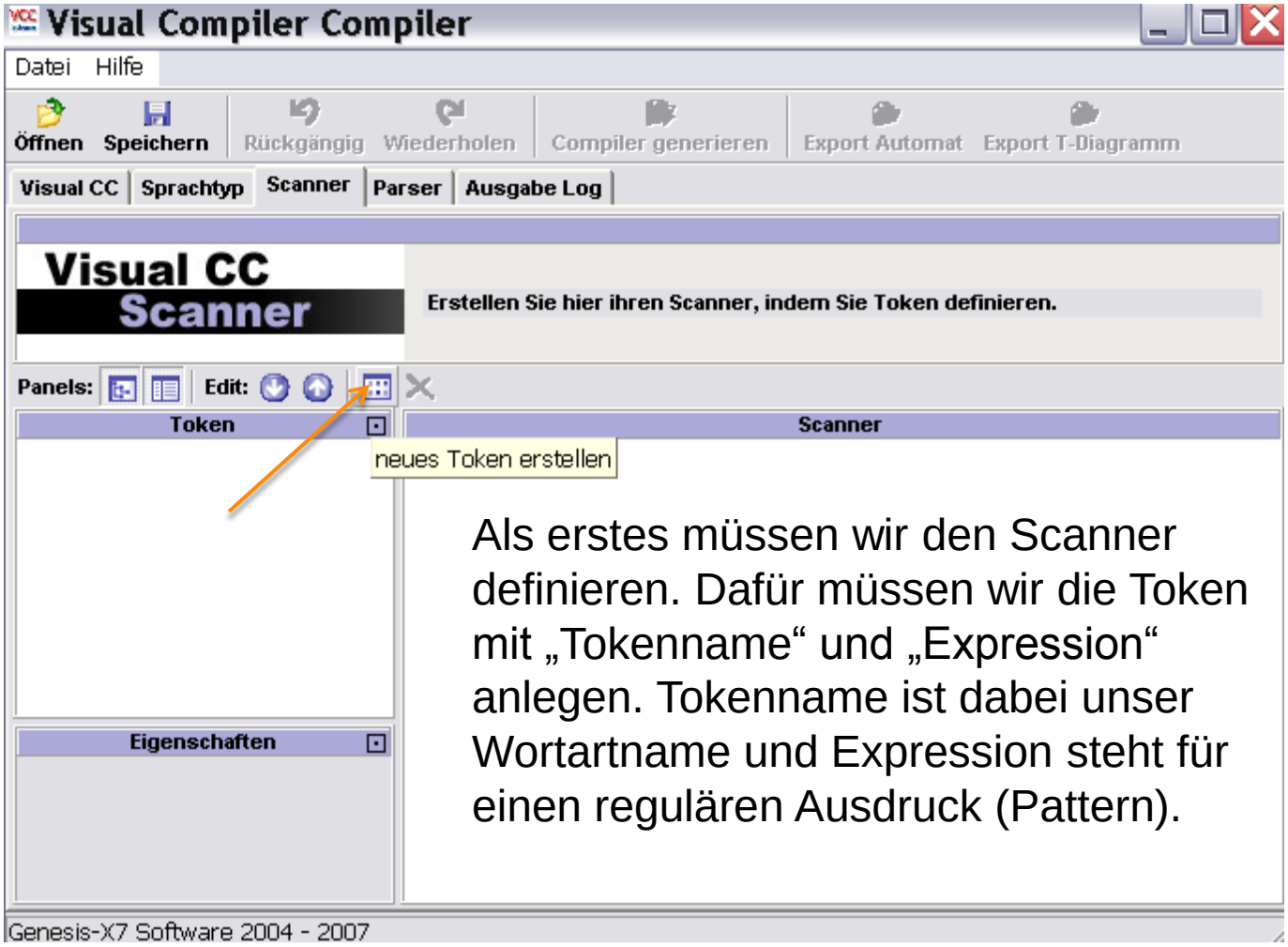
Implementierung mit VCC

- Wir öffnen VCC und wählen „Neue VCC-Datei“:

Implementierung mit VCC

- Anschließend wählen wir eine Programmiersprache.
- Alle Ausgabe-Attribute der Grammatik müssen später in dieser Sprache verfasst werden.

Implementierung mit VCC - Scanner



Als erstes müssen wir den Scanner definieren. Dafür müssen wir die Token mit „Tokenname“ und „Expression“ anlegen. Tokenname ist dabei unser Wortartname und Expression steht für einen regulären Ausdruck (Pattern).

Implementierung mit VCC - Scanner


```
program    → block .
block      → const1  var1  proc1 statement
const1     → CONST ident = number const2 ;  | ε
const2     → , ident = number const2  | ε
var1       → VAR ident var2 ;  | ε
var2       → , ident var2  | ε
proc1      → proc2 proc1  | ε
proc2      → PROCEDURE ident ; block ;
statement  → ident := expression
           | CALL ident
           | ? ident
           | ! expression
           | BEGIN statement statements END
           | IF condition THEN statement
           | WHILE condition DO statement
           | ε
statements → ; statement statements | ε
```

```
condition  → ODD expression
           | expression = expression
           | expression # expression
           | expression < expression
           | expression <= expression
           | expression >= expression
           | expression > expression
expression → + term expblock
           | - term expblock
           | term expblock
expblock   → + term expblock
           | - term expblock
           | ε
term       → factor termblock
termblock  → * factor termblock
           | / factor termblock
           | ε
factor     →  ident
           | number
           | ( expression )
```

Streichen wir alle Nichtterminale um zu sehen welche Terminale für unseren Scanner verbleiben.

Implementierung mit VCC - Scanner

.	\.
CONST	CONST
ident	[a-zA-Z]+
number	[0-9]+
;	;
,	,
VAR	VAR
PROCEDURE	PROCEDURE
:=	:=
CALL	CALL
?	\?
!	!
BEGIN	BEGIN
END	END
IF	IF
THEN	THEN

WHILE	WHILE
DO	DO
ODD	ODD
=	=
#	#|<|>|<=|>=
<	
<=	
>=	
>	
+	\+|\-
-	
*	*|/
/	
(	\(
)	\)

In rot sind nun die regulären Ausdrücke für die Terminale angegeben. Zu beachten ist das jegliche Metazeichen mit einem vorangestellen „\“ zu escapen sind! (mehr dazu hier)

Implementierung mit VCC - Scanner

ProgramEnde	**\.**
Cost	**CONST**
ident	**[a-zA-Z]+**
number	**[0-9]+**
Semicolon	**;**
Komma	**,**
Var	**VAR**
Procedure	**PROCEDURE**
Zuweisung	**:=**
Call	**CALL**
Eingabe	**\?**
Ausgabe	**!**
Begin	**BEGIN**
End	**END**
If	**IF**
Then	**THEN**

While	**WHILE**
Do	**DO**
Ungerade	**ODD**
Gleich	**=**
Relation	**#|<|>|<=|>=**
Strichzeichen	**\+|\-**
Punktzeichen	***|/**
KlammerAuf	**\(**
KlammerZu	**\)**

Nun vergeben wir zu jedem Ausdruck einen Tokennamen um die Wortarten zu definieren.

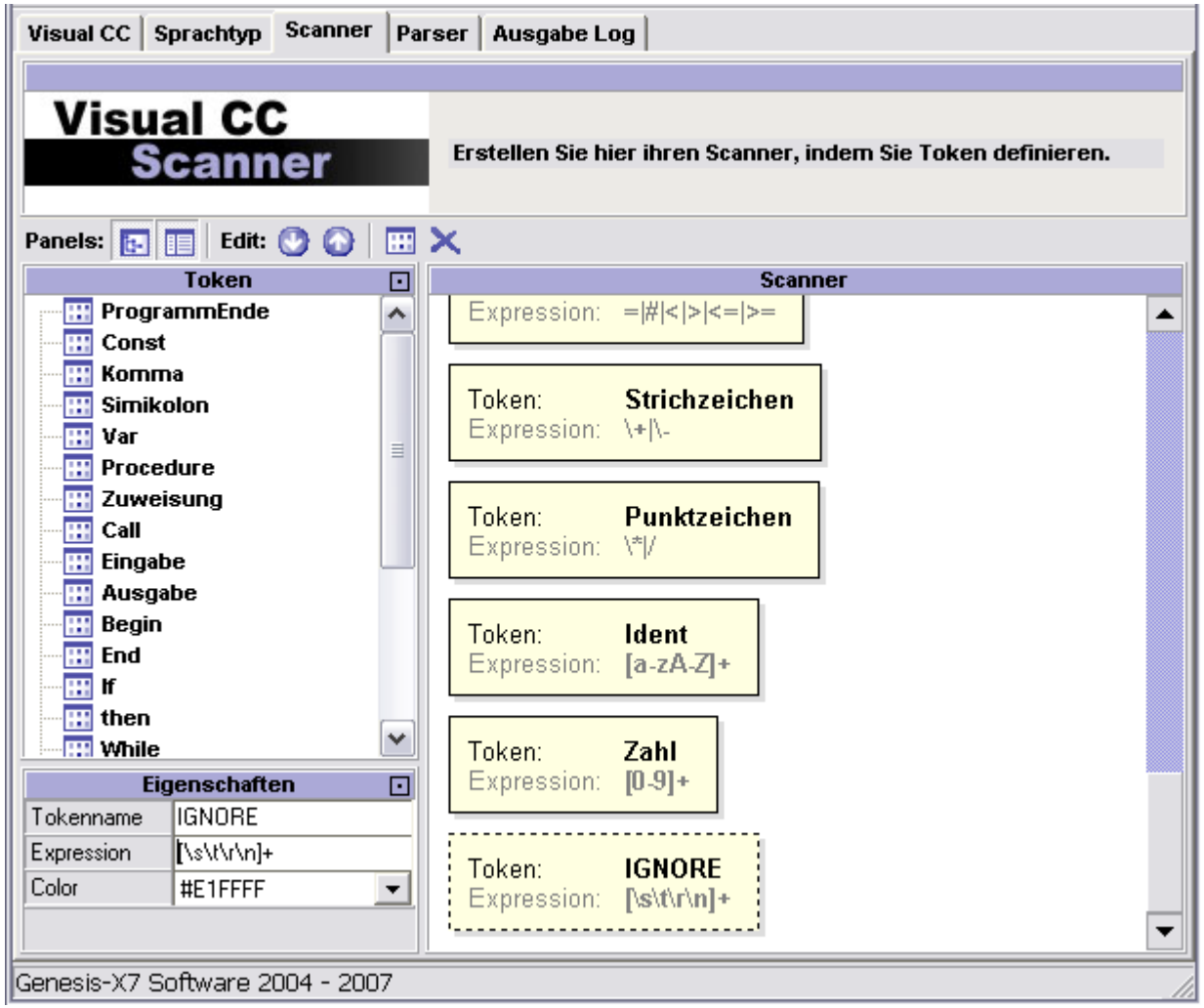
Implementierung mit VCC - Scanner

- Nun müssen wir uns noch für die richtige Reihenfolge bei den Wortarten entscheiden, die sich in ihrer Schreibweise überschneiden.
- Die Wortart „Ident“ überlappt mit allen anderen Schlüsselworten wie „**BEGIN**“ oder „**DO**“
- → Ident muss ganz unten in der Wortartliste stehen um die anderen Schlüsselwörter beim scannen zu favorisieren!
- Alle anderen Wortarten überlappen in unser PL/0 Sprache nicht.

Implementierung mit VCC - Scanner

- Zusätzlich geben wir noch ein spezielles Token an:

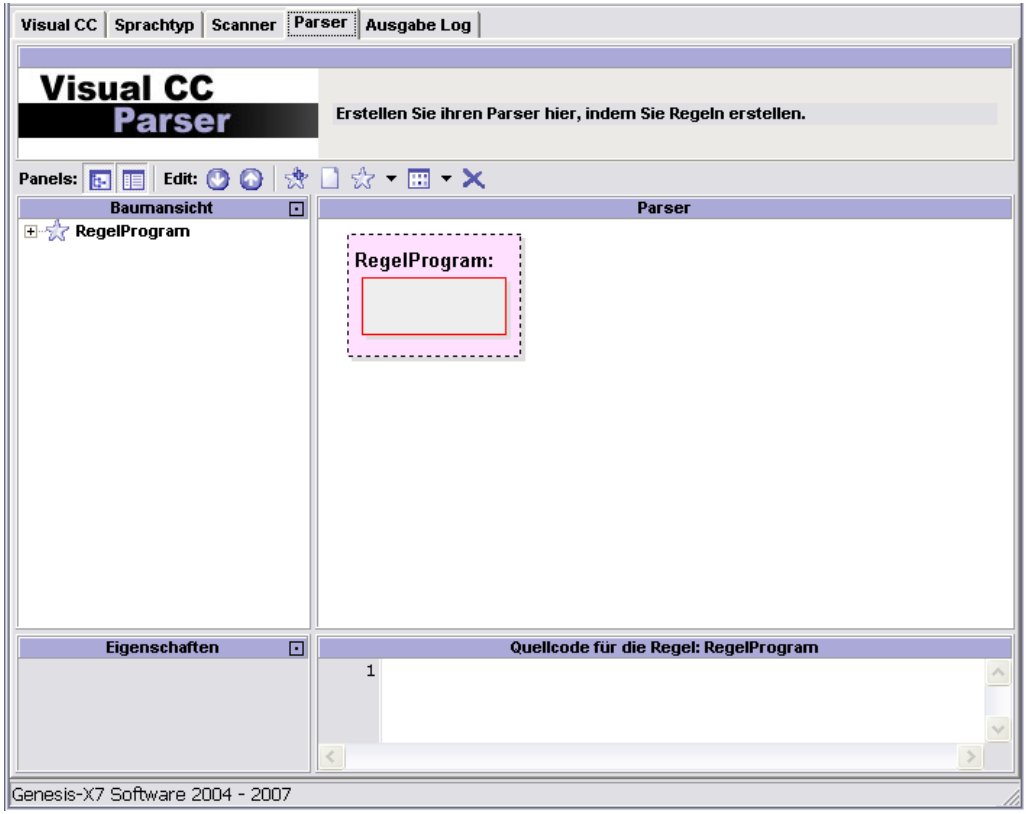
  IGNORE
  filtert alle Zeichen heraus, die nicht zur Sprache gehören sollen



Implementierung mit VCC - Scanner

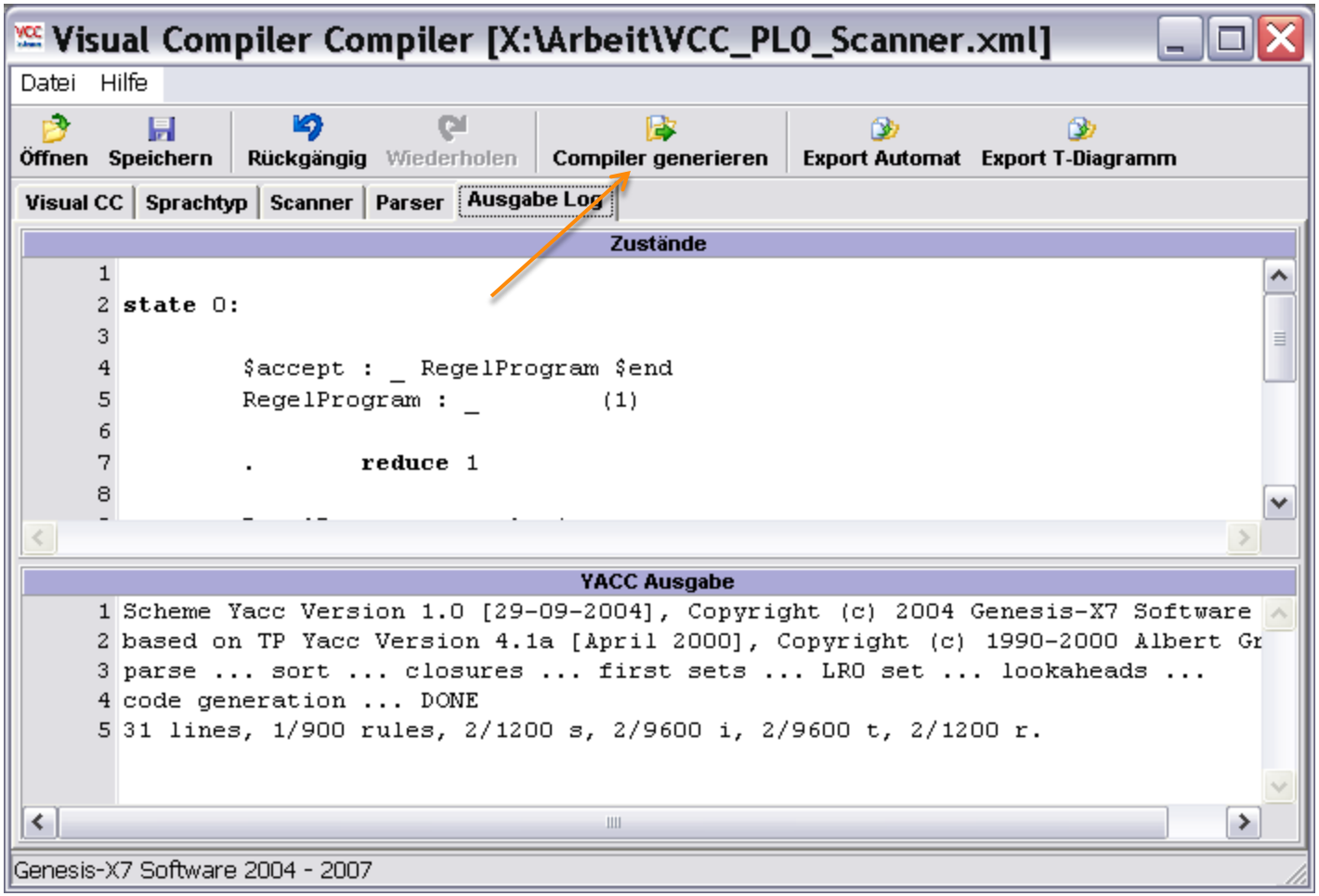
- Nun legen wir eine erste Regel im Parser an, damit wir zunächst den Scanner an einem PL/0 Quelltext testen können.

- Hinweis:
  Regelnamen dürfen nie wie Tokennamen heißen!
  Verwende keine Leerzeichen oder Sonderzeichen für Tokennnamen oder Regelnamen!



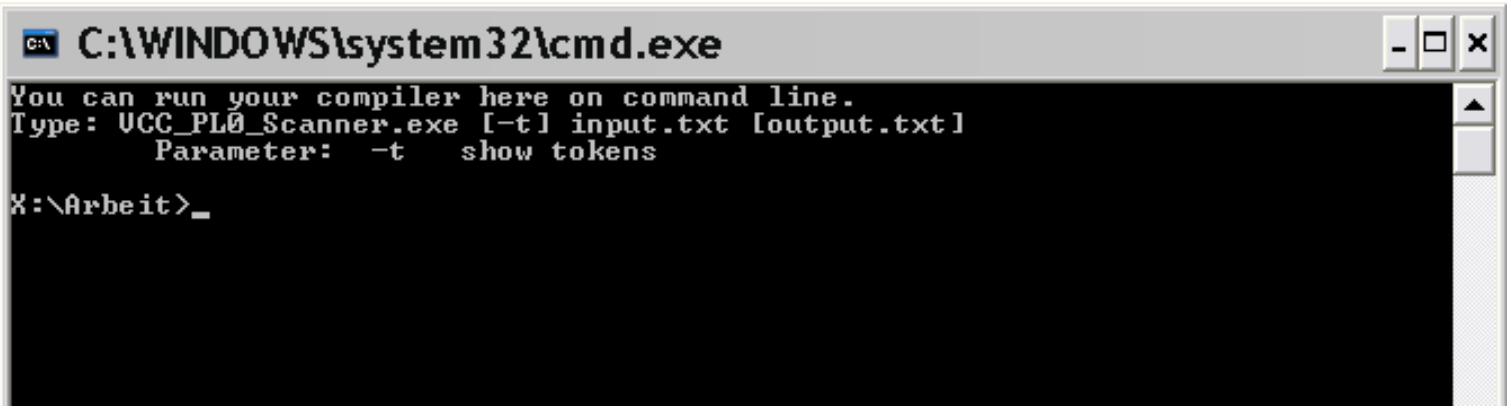
Implementierung mit VCC - Scanner

- Wir erstellen den Compiler über den entsprechenden Button:



Implementierung mit VCC - Scanner

- Je nach gewählter Programmiersprache (C#, Delphi, Java oder Scheme) erhalten wir eine Eingabekonsole.



- Wir wollen nun einen Quelltext von unserem Compiler verarbeiten lassen. Zur Anzeige der Token geben wir auch den Parameter -t an.
- In obigen Beispiel: „`VCC_PL0_Scanner -t in.txt`“

Implementierung mit VCC - Scanner

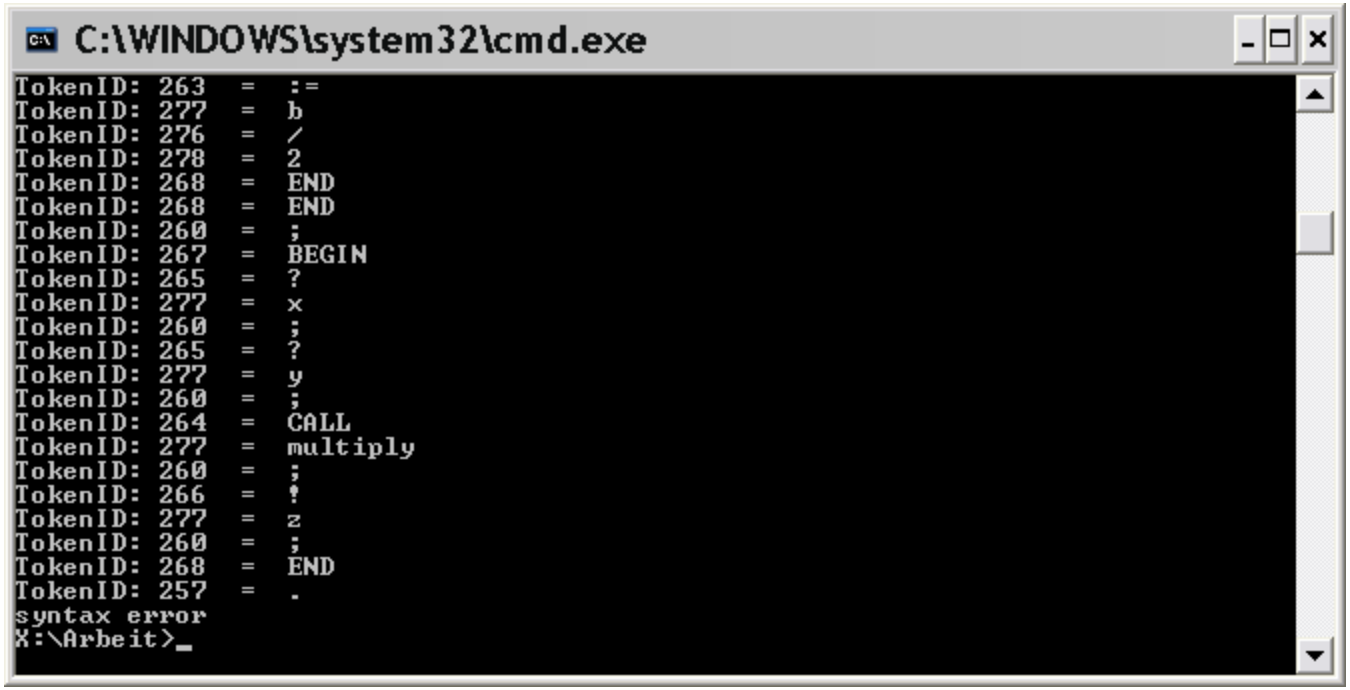
Als Eingabe verwenden wir einen PL/0 Quelltext:

```
VAR x,y,z;

PROCEDURE multiply;
  VAR a, b;
  BEGIN
    a := x; b := y; z := 0;
    WHILE b > 0 DO
      BEGIN
        IF ODD z THEN z := z + a;
        a := 2 * a;
        b:= b/2
     END
END;
BEGIN
  ?x; ?y; CALL multiply; !z;
END.
```

Implementierung mit VCC - Scanner

- Die Ausgabe zeigt uns welche Token erkannt wurden. Dabei sind die Tokennamen durch ID‘s repräsentiert:



Am Ende wird ein „syntax error“ ausgeben. Dies ist aber nicht verwunderlich, da wir unseren Parser ja noch nicht definiert haben.

Implementierung mit VCC - Parser

- Nun können wir die Regeln für den Parser anlegen. Hierfür nehmen wir uns die Syntaxregeln der Grammatik und übertragen diese in VCC.
- Dabei ersetzen wir die Terminale der Grammatik mit den Tokennamen die wir eben im Scanner vergeben haben.
- Einige Regeln der Grammatik können wir nun bereits zusammenfassen.
  (z.B.: Expression **<=** Expression und
  Expression **>=** Expression sind zusammen
  Expression Relation Expression)

Implementierung mit VCC - Parser

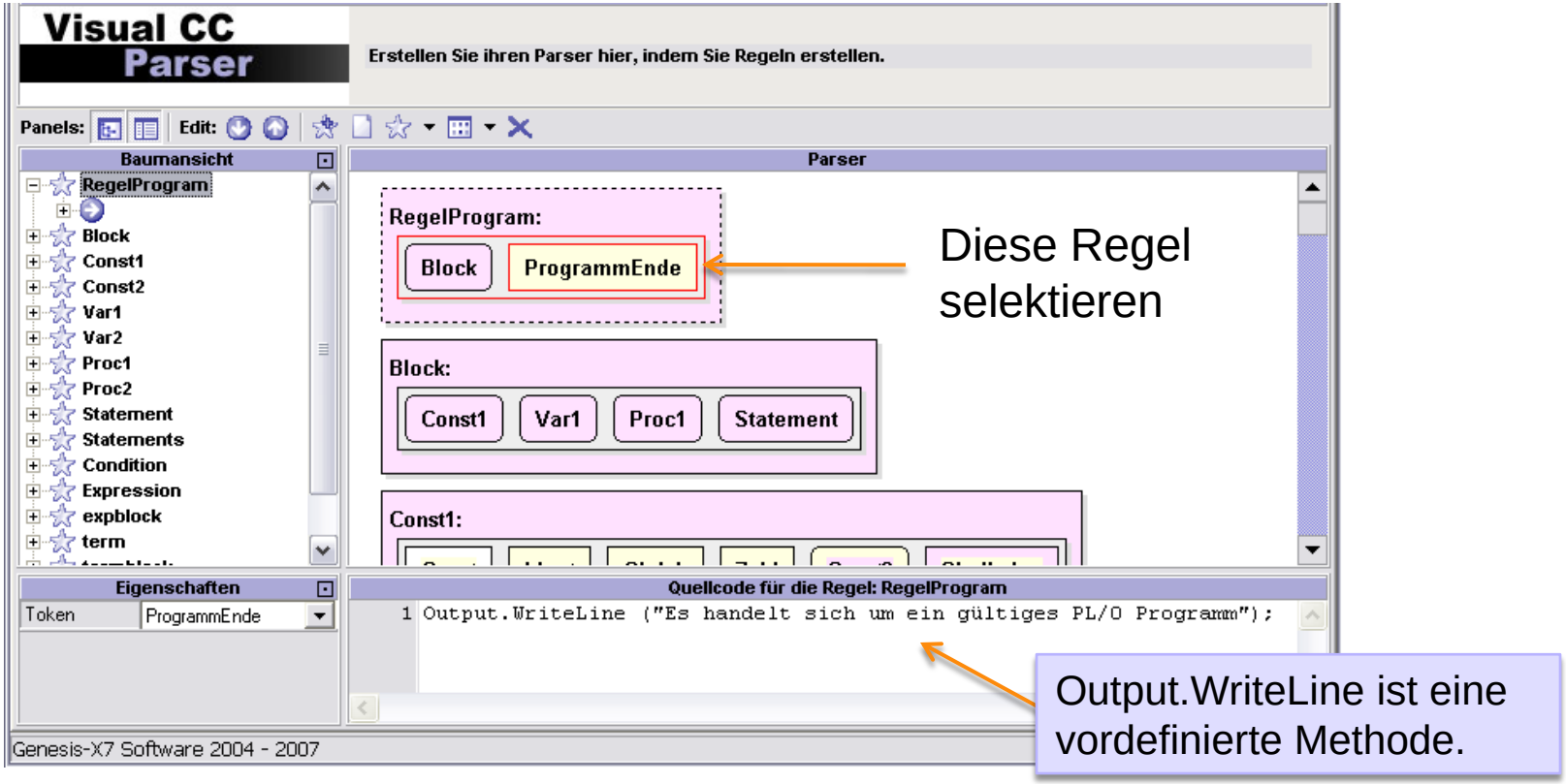

```
program    → block ProgramEnde
block      → const1  var1  proc1 statement
const1     → Cost ident Gleich Zahl const2 Semicolon | ε
const2     → Komma ident Gleich Zahl const2  | ε
var1       → Var ident var2 Semicolon | ε
var2       → Komma ident var2  | ε
proc1      → proc2 proc1  | ε
proc2      → Procedure ident Semicolon block Semicolon
statement  → ident Zuweisung expression
           | Call ident
           | Eingabe ident
           | Ausgabe expression
           | Begin statement statements End
           | If condition Then statement
           | While condition Do statement
           | ε
statements → Semicolon statement statements | ε
```

```
condition  → Ungerade expression
           | expression Relation expression
           | expression Gleich expression
expression → Strichzeichen term expblock
           | term expblock
expblock   → Strichzeichen term expblock
           | ε
term       → factor termblock
termblock  → Punktzeichen factor termblock
           | ε
factor     →  ident
           | Zahl
           | KlammerAuf expression KlammerZu
```

- Tokennamen aus dem Scanner
- Regelnamen im Parser

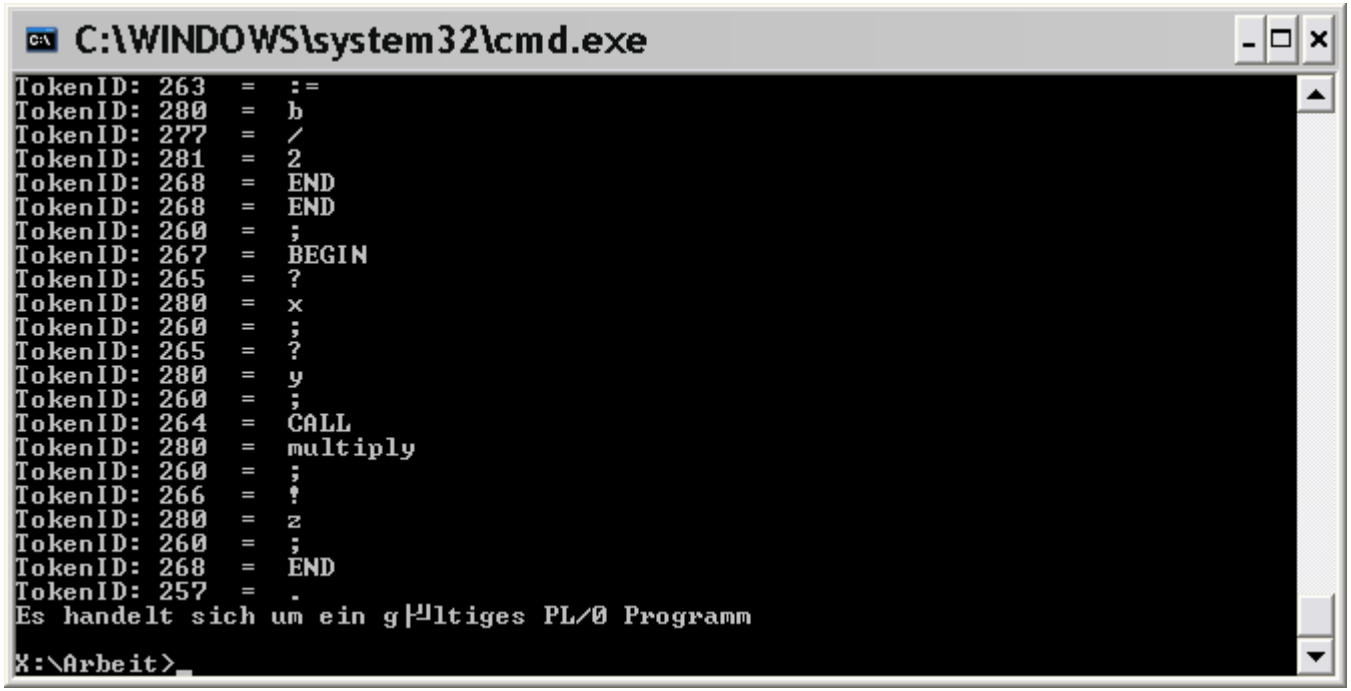
Implementierung mit VCC - Parser

- Nachdem alle Regeln übertragen wurden, legen wir ein Attribut für die oberste Regel fest:



Implementierung mit VCC - Parser

- Erneut lassen wir den Compiler generieren und testen ihn anschließend mit unserem Quelltext:



Implementierung mit VCC - Parser

- Nun wird kein „syntax error“ mehr ausgegeben sondern unser Zielcode der Sprache „S_{out}“.
- Wenn wir nun den Quelltext (in.txt) verändern und bewusst Fehler einbauen, bekommen wir wieder entsprechende Syntaxfehler angezeigt.
- Verwenden wir ein Zeichen wie „~“, welches durch kein Pattern im Scanner beschrieben werden kann:

Zusammenfassung

- Wir haben somit einen vollständigen Compiler für die Sprache PL/0 entwickelt.
- Die Zielcodegenerierung haben wir hier jedoch nur minimal betrachtet.
- Um sich mehr mit Zielcodegenerierung zu beschäftigen, schauen Sie sich die Beispiele im VCC Sample Verzeichnis an (Calc oder ChartCompiler).

Viel Erfolg mit VCC