

PBLexerGen / PBParserGen - Tutorial

Vorwort:

Ich habe mich dazu entschlossen statt einer Beschreibung dieses Tutorial zu schreiben, das anhand eines einfachen Beispiels die einzelnen Möglichkeiten und Besonderheiten der beiden Generatoren beleuchtet. Es beschreibt die Entwicklung eines einfachen Programms, welches mathematische Ausdrücke syntaktisch analysiert, und dabei berechnet. Das fertige Projekt liegt im Unterordner „Rechner“.

Dieses Tutorial soll vor allem Leute ansprechen, die noch keine Erfahrung mit Parser- und Lexergeneratoren haben. Die einzelnen Schritte in diesem Tutorial sollen ein Gefühl vermitteln, wie man selbst sehr einfach und schnell auch komplexere Parser und Lexer schreiben kann.

Für diejenigen, die bereits mit Generatoren gearbeitet haben, sei gesagt, dass die Programme „bison“ und „flex“ als Vorbild dienten bei der Entwicklung von PBParserGen und PBLexerGen. Somit ist die Funktionsweise und Syntax sehr ähnlich. Unterschiede, vor allem in der Syntax (wie z.b. „\$x“ statt „\$x“), können entweder in diesem Tutorial oder in dem zugrundeliegenden Beispiel eingesehen werden.

In diesem Tutorial wird zusätzlich das Programm PBMakeTool benutzt. Eine Beschreibung von PBMakeTool ist in dessen Ordner zu finden.

Erste Schritte:

Als erstes erstellen wir einen Ordner namens "Rechner" und in diesem die folgenden Dateien:

- Rechner.pb: Die ausführende Datei
- Parser.pb: Grammatikbeschreibung für den PBParserGen
- Lexer.pb: RegEx-Regeln für den PBLexerGen
- PBMake.txt: Anweisungen für das PBMakeTool

PBMake.txt:

Das PBMakeTool ist ein Ersatz für das Programm make aus Linux für Windowsnutzer.

In der PBMake.txt können wir dem PBMakeTool sagen, was er vor und nach dem Kompilieren von PureBasic machen soll. In unserem Fall ist es nur wichtig, dass die Generatoren vor dem Kompilieren aufgerufen werden. Dies kann man mit diesen Zeilen tun:

```
1. BEFORE_COMP, BEFORE_EXE:
2.   PBLexerGen  Lexer.pb  -o Lexer_out.pb
3.   PBParserGen Parser.pb -o Parser_out.pb
4. ;
```

PBMake.txt

Die beiden Schlüsselwörter `BEFORE_COMP` und `BEFORE_EXE` zeigen PBMakeTool an, dass die folgenden Befehle dann ausgeführt werden sollen, wenn eine Datei aus diesem Ordner kompiliert, oder ein Executable erstellt wird. In unserem Fall ist das, wenn wir die Datei Rechner.pb kompilieren und starten wollen.

Wenn dies der Fall ist, generiert der Lexergenerator mit Eingabe Lexer.pb einen Lexer in der Datei Lexer_out.pb, und der Parsergenerator mit Eingabe Parser.pb einen Parser in der Datei Parser_out.pb.

Rechner.pb:

Nun befassen wir uns mit unserer ausführenden Datei "Rechner.pb". In der müssen wir die erstellten Output-Dateien includen. Zu beachten ist dabei, dass der Parser stets zuerst includet wird. Außerdem werden wir schon mal eine kleine Funktion schreiben, indem wir den Parser aufrufen:

```
1. IncludeFile "Parser_out.pb"
2. IncludeFile "Lexer_out.pb"
3.
4. Procedure EvalString(string.s)
5.   *yyin = @string
6.   Debug yyparser()
7. EndProcedure
8.
9. EvalString("2 + 3 * 4")
```

Rechner.pb

In der Funktion "EvalString" setzen wir zunächst die Variable "*yyin". Diese globale Variable zeigt dem Parser und Lexer an, an welcher Speicherstelle nun gelesen werden soll. In unserem Fall ist das die Speicheradresse des übergebenen Strings. Dann rufen wir nur noch yyparser() auf, und dieser erledigt dann die syntaktische Analyse. Die Rückgabe von yyparser() ist 1, wenn der String syntaktisch korrekt ist, ansonsten 0.

Lexer.pb:

Zur Zeit würde die Funktion yyparser() aber noch gar nichts machen, da wir den Lexer und Parser noch gar nicht definiert haben. Dies holen wir sofort nach. Doch zunächst die Frage: was ist überhaupt ein Lexer? Ein Lexer trennt eine Eingabe, in unserem Fall z. B. "2 + 3 * 4" in die einzelnen Bestandteile "2", "+", "3", "*" und "4". In der Regel können diese Teile (genannt Token) auch mehrere Zeichen lang sein (z.B. Variablennamen oder Zahlen mit mehreren Ziffern). Wie der Lexer nun vorgehen soll, beschreiben wir mit Regulären Ausdrücken (RegEx), die in unserer Lexer-Datei definieren werden.

Im Allgemeinen wird eine Lexer-Datei in drei Teile geteilt, jeweils getrennt durch "%%":

```
1. ;
2. ; Individueller Code vor dem Lexer und
3. ; Schlüsselwörter für PBLexerGen
4. ;
5.
6. %%
7.
8. ;
9. ; RegEx-Regeln
10. ;
11.
12. %%
13.
14. ;
15. ; Individueller Code nach dem Lexer
16. ;
```

Im ersten und im dritten Teil kann beliebiger Code stehen, der unverändert übernommen wird, bis auf einige wenige Schlüsselwörter im ersten Teil:

- "%nocase": alle RegEx-Regeln werden ohne Prüfung der Groß- und Kleinschreibung interpretiert.
- "%userdata": dies wird noch später in diesem Tutorial angesprochen

Der zweite Teil ist der weitaus interessantere. Hier werden nun die einzelnen RegEx-Regeln definiert, und Code geschrieben, der aufgerufen wird, wenn sie gefunden werden. Der Code kann sich über mehrere Zeilen erstrecken und wird durch "{" und "}" eingeschlossen. So sieht eine Regel aus:

```
1. RegEx { ... code ...
2.     ... weiterer code ...
3. }
```

In der generierten Output-Datei wird dieser Code in einer Prozedur stehen. "Protected"- und "ProcedureReturn"- Aufrufe sind hier also möglich.

Nun erstellen wir uns einen Lexer für unseren Rechner:

```
1. %nocase
2.
3. %%
4.
5. <<EOF>> { ProcedureReturn $<EOF> }
6. [ \t]    { } ; ignore char
7.
8. [0-9]+   { ProcedureReturn $NUMBER }
9. [+\\-*/] { ProcedureReturn Asc(yytext) } ; operators
10.
11. .       { yyerror("Invalid char: "+yytext) }
12.
13. %%
```

Lexer.pb

Die erste Zeile "%nocase" ist zwar noch nicht nötig, da wir bisher keine Regel mit Buchstaben haben, zeigt aber, wie "%nocase" im ersten Teil verwendet wird.

Die Zeilen 5 und 6 sollten in jedem Lexer vorhanden sein. Zeile 5 ist dazu da, dem Parser zu sagen, dass die Eingabe vollständig gelesen wurde. Ohne diese Regel, kann es passieren, dass der Parser sich nicht beendet.

Die Regel in Zeile 6 zeigt an, welche Zeichen ignoriert werden können. In diesem Fall sind das das Leer- und das Tabulatorzeichen "\t". Durch die Rückgabe von -1 wird dies dem Parser mitgeteilt. "ProcedureReturn -1" kann hier jedoch weggelassen werden, da dies in jeder Regel am Ende hinzugefügt wird.

In Zeile 8 definieren wir nun die Regel für das Token "NUMBER". Es besteht aus beliebig vielen Ziffern 0-9, aber mindestens einem. Der zurückgegebene Wert "\$NUMBER" wird durch den Lexergenerator in eine Konstante umgeformt. Diese Konstante wird durch den PBParserGen generiert (dazu später mehr).

In Zeile 9 definieren wir die Regel für die Operatoren. Zu beachten ist hierbei, dass das Minuszeichen "-" in eckigen Klammern ein Steuerzeichen ist (siehe hierzu Regel in Zeile 8). Deshalb muss davor ein "\" stehen, damit das Minuszeichen auch als Minuszeichen interpretiert wird. Die Rückgabe ist nun keine "\$"-Konstante, sondern einfach der Ascii-Wert des Zeichens. Dies ist die zweite Möglichkeit dem Parser zu sagen, welches Token gerade gelesen wurde.

Hinweis: In der globalen Variable "yytext" ist nach einem erfolgreichem Matchen eines RegEx, der Text des Tokens gespeichert. Hier also nur das Zeichen des Operators.

Hinweis: Eine andere Möglichkeit die Regel zu definieren, wäre z.B. noch diese: "+|\"|-|\"*|\"/\". Welche man bevorzugt, ist Sache des Entwicklers ;).

Der Punkt in Zeile 11 zeigt an, dass hier jedes Zeichen erlaubt ist. Da diese Regel allerdings am Ende steht, wird diese nur dann aktiv, wenn keine andere Regel angewendet werden kann. Falls wir also ein falsches Zeichen in der Eingabe haben, signalisieren wir das, indem wir die yyerror-Funktion aufrufen.

Parser.pb:

Nun kommen wir zum Parser. Ein Parser prüft, ob eine Eingabe syntaktisch korrekt ist. Dies macht er anhand einer Grammatik die wir nun definieren müssen.

Parser-Dateien sind ähnlich aufgebaut, wie Lexer-Dateien, besitzen aber mehr Schlüsselwörter im ersten Teil. So können bzw. müssen mit "%token" die Token definiert werden, die der Lexer zurückgeben kann. Mit "%type" werden die Nichtterminalsymbole der Grammatik definiert, und mit "%start" das Nichtterminalsymbol, mit dem die Analyse einer Eingabe begonnen wird. Nichtterminalsymbole sind Hilfssymbole, die in der Regel aus mehreren Terminalsymbolen (Token) und Nichtterminalsymbolen bestehen können. Im zweiten Teil wird dann die Grammatik beschrieben, die der Analyse zugrunde liegt.

Hier nun ein einfaches Beispiel für unseren Rechner:

```
1. %token NUMBER
2.
3. %type eval exp
4. %start eval
5.
6. %%
7.
8. exp: ;[
9.     NUMBER
10.    | exp '+' exp
11.    | exp '-' exp
12.    | exp '*' exp
13.    | exp '/' exp
14.    ;]
15.
16. eval: exp
17.
18. %%
```

Parser.pb

In Zeile 1 definieren wir unser einziges Token "NUMBER". Token müssen mit einem Großbuchstaben beginnen, und sollten auch sonst nur aus Großbuchstaben und "_" bestehen. In dieser Zeile wird auch die Konstante "\$NUMBER" definiert, die wir im Lexer verwendet haben.

In Zeile 3 definieren wir nun die Nichtterminalsymbole, die wir für unsere Grammatik benötigen. Nichtterminalsymbole müssen mit einem Kleinbuchstaben beginnen.

Hinweis: Sowohl bei "%token" und "%type", als auch bei allen anderen Schlüsselwörtern werden mehrere Symbole mit Leerzeichen getrennt (wie in Zeile 3).

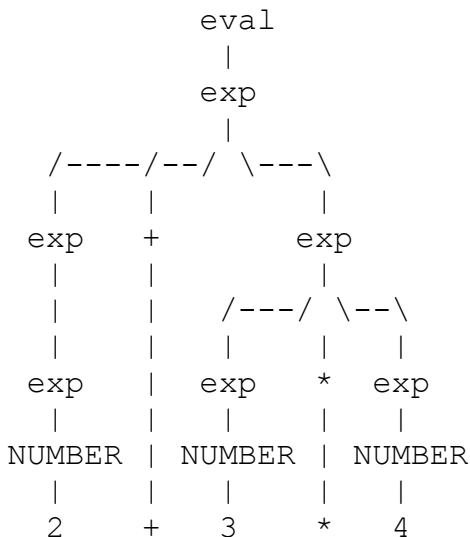
Mit "%start" (Zeile 4) wird hier das Nichtterminalsymbol "eval" als Startsymbol definiert. Dies bedeutet, dass jede Eingabe zu diesem Symbol reduziert werden muss, damit eine Eingabe syntaktisch korrekt ist.

In Zeile 16 definieren wir die Regel für dieses Symbol "eval". Das Zeichen ":" kann wie ein "besteht aus" gelesen werden. Ein Symbol "eval" besteht also aus dem Symbol "exp".

In den Zeilen 8-14 sind nun alle Reduzierungsmöglichkeiten für exp aufgelistet. Die Möglichkeiten werden mit "|" getrennt ("|" kann wie "oder" gelesen werden). Ein Expression "exp" besteht also aus einer Zahl oder aus zwei Expressions getrennt durch einem Operator.

Hinweis: Die beiden Kommentare "[:]" und "[:]" gehören nicht zur Syntax des PBParserGen. Sie dienen lediglich der Einrückungsautomatik der PB-IDE.

Mit diesen Regeln kann bereits geprüft werden, ob beispielsweise die Eingabe "2 + 3 * 4" syntaktisch korrekt ist:



Dies ist hier der Fall, da die Eingabe bis zum Startsymbol "eval" reduziert werden kann.

Erster Test:

Zeit für unseren ersten Test. Dazu starten wir nun die "Rechner.pb". PBLexerGen wird als erstes aufgerufen, und läuft durch. PBParserGen bleibt allerdings stehen, mit einer Warnung:

```

16 shift/reduce-conflicts
0 reduce/reduce-conflicts
  
```

Darunter werden diese Konflikte nun aufgelistet. Aber was hat dies nun zu bedeuten?

Mehrdeutige Grammatiken:

Das Problem unserer Grammatik ist, dass diese leider mehrdeutig ist. Sehen wir uns nochmal unser Beispiel an "2 + 3 * 4". Dies kann auf zwei Arten reduziert werden:

```

2 + 3 * 4 -> (exp + exp) * exp -> exp * exp -> exp -> eval
2 + 3 * 4 -> exp + (exp * exp) -> exp + exp -> exp -> eval
  
```

Wenn wir uns die Grammatik nochmal anschauen, werden wir merken, dass der Parsergenerator nichts von Prioritäten der Operatoren oder dessen Assoziativität unserer Grammatik kennt.

Um diese zu definieren gibt es dafür zwei weitere Schlüsselwörter im ersten Teil des Parsers "%left" und "%right". "%left" bedeutet dabei, dass das Token (also der Operator) links-assoziativ, "%right" rechts-assoziativ ist.

```
1 + 2 + 3 -> (exp + exp) + exp -> ...    mit %left '+'
1 + 2 + 3 -> exp + (exp + exp) -> ...    mit %right '+'
```

Die Reihenfolge, in der die Token angegeben werden, bestimmt die Priorität. Wir erweitern die "Parser.pb"-Datei:

```
1. %token NUMBER
2.
3. %left '+' '-'
4. %left '*' '/'
5.
6. %type eval exp
7. %start eval
8.
9. %%
10.
11. exp: ;[
12.     NUMBER
13.     | exp '+' exp
14.     | exp '-' exp
15.     | exp '*' exp
16.     | exp '/' exp
17. ;]
18.
19. eval: exp
20.
21. %%
```

Parser.pb

Die beiden Zeilen 3-4 haben nun folgende Bedeutungen:

1. Alle vier Operatoren sind links-assoziativ ("%left")
2. Die Operatoren '+' und '-' haben die gleiche Priorität (gleiche Zeile)
3. Die Operatoren '*' und '/' haben die gleiche Priorität (gleiche Zeile)
4. Die Operatoren '*' und '/' haben eine höhere Priorität als '+' und '-' (Reihenfolge)

Zweiter Test:

Die Grammatik ist nun eindeutig. Nun können wir es nochmal probieren. Starten wir also erneut "Rechner.pb". Diesmal durchlaufen beide Generatoren und unser Rechner wird gestartet. Die Ausgabe ist 1. Dies bedeutet, dass die Eingabe "2 + 3 * 4" syntaktisch korrekt ist. Bei andere Eingaben wie z.b. "2 + " wird yyparser() 0 zurückgeben.

Nun kann also unser Rechner eine Eingabe auf Syntax-Fehler prüfen. Allerdings fehlt noch der semantische Teil, also die Berechnung einer Eingabe.

Berechnung:

Der erste Schritt ist, den Lexer so zu verändern, dass er dem Parser mitteilt, welche Zahl er gerade gelesen hat. Bisher wurde mit der Rückgabe von "\$NUMBER" nur signalisiert, dass das gerade gelesene Token eine Zahl ist, aber nicht welche. Dies machen wir mit Hilfe der globalen strukturierten Variable `yylval`:

```
1. %nocase
2.
3. %%
4.
5. <<EOF>> { ProcedureReturn $<EOF> }
6. [ \t]    { } ; ignore char
7.
8. [0-9]+    { ;[
9.     yylval\int = Val(yytext)
10.    ProcedureReturn $NUMBER
11. ];]
12.
13. [+\\-*/] { ProcedureReturn Asc(yytext) } ; operators
14.
15. .        { yyerror("Invalid char: "+yytext) }
16.
17. %%
```

Lexer.pb

In Zeile 9 setzen wir den Struktureintrag "int" von "yylval" auf den Wert der gelesenen Zahl. Damit die Struktur von "yylval" auch diesen Eintrag besitzt, definieren wir nun diese Struktur im Parser mit Hilfe des Schlüsselwortes "%structure".

```
1. %Structure
2.   int.i
3. %EndStructure
4.
5. [...]
```

Da die Variable "yylval" nur einmal existiert, wird im Parser auf den Wert eines Tokens nicht mit dieser Variable zugegriffen, sondern mit den Variablen `$x`, wobei `x` die Position in der Regel anzeigt (beginnend mit 1). Hier mal ein kleines Beispiel:

```
exp: NUMBER { $$ = $1 }
```

Hier wurde noch eine weitere Variable "\$\$" verwendet, um den Wert eines Nicht-terminalsymbols auf der linken Seite zu setzen.

Beim Verwenden dieser Variablen muss dem Parser allerdings noch angegeben werden, welchen Struktureintrag er bei diesem Token bzw. dem Nichtterminalsymbol benutzen muss. Dies wird mit Hilfe von "%token" und "%type" erreicht:

```
%token.int NUMBER
%type.int exp
```


Mit diesen Möglichkeiten, können wir den Rechner nun zum Rechnen bringen:

```
1. %Structure
2.   int.i
3. %EndStructure
4.
5. %token.int NUMBER
6.
7. %left '+' '-'
8. %left '*' '/'
9.
10. %type.int eval exp
11. %start eval
12.
13. %%
14.
15. exp: ;[
16.   NUMBER          { $$ = $1 }
17.   | exp '+' exp { $$ = $1 + $3 }
18.   | exp '-' exp { $$ = $1 - $3 }
19.   | exp '*' exp { $$ = $1 * $3 }
20.   | exp '/' exp { $$ = $1 / $3 }
21. ;]
22.
23. eval: exp { Debug $1 }
24.
25. %%
```

Parser.pb

In den Zeilen 16-20 wird der Wert eines Expressions gesetzt, wenn eine dieser Regeln beim Parsen angewendet wird. In Zeile 16 übernimmt es einfach den Wert von NUMBER. Dieser Wert wurde im Lexer durch "yyval.int" gesetzt.

In den Zeilen 17-20 wird aus den beiden anderen Expressions auf der rechten Seite der neue Wert berechnet.

Wenn der Parser eine syntaktisch korrekte Eingabe analysiert hat, wird der Code in Zeile 23 ausgeführt. Hier wird der Wert des Expressions im Debugger ausgegeben.

Starten wir nun nochmal den Rechner. Als Ausgabe steht da nun 14. Mit ein wenig Kopfrechnen kommen wir bei der Eingabe "2 + 3 * 4" auch auf dieses Ergebnis.

Der Parser funktioniert nun soweit. Allerdings wäre es gut, wenn man nun diesen Wert auch außerhalb des Parsers benutzen kann.

UserData:

Unsere Prozedur "EvalString" aus der Datei "Rechner.pb" soll nun den berechneten Wert zurückgeben. Dies könnte man beispielsweise mit einer globalen Variable bewerkstelligen. Dies ist aber nicht immer erwünscht. Eine andere Möglichkeit stellt der Parser zur Verfügung. Der Funktion yyparser() kann man einen Wert übergeben. Auf diesen Wert kann in jedem Codestück, eingeschlossen durch "{" und "}", mit der Variable "*userData" zugegriffen werden.

```
1. eval: exp { Debug *userData } ; -> 42424242
2.
3. [...]
4.
5. yyparser(42424242)
```

Mit dem Schlüsselwort "%userdata" kann der Name der Variable auch geändert werden, um damit beispielsweise eine Struktur hinzuzufügen.

In unserem Fall werden wir die Adresse einer Integer-Variable übergeben, und in diese dann das Ergebnis speichern:

```
1. %userdata *result.Integer
2.
3. %Structure
4.   int.i
5. %EndStructure
6.
7. %token.int NUMBER
8.
9. %left '+' '-'
10. %left '*' '/'
11.
12. %type.int eval exp
13. %start eval
14.
15. %%
16.
17. exp: ;[
18.     NUMBER          { $$ = $1 }
19.     | exp '+' exp { $$ = $1 + $3 }
20.     | exp '-' exp { $$ = $1 - $3 }
21.     | exp '*' exp { $$ = $1 * $3 }
22.     | exp '/' exp { $$ = $1 / $3 }
23. ;]
24.
25. eval: exp { *result\i = $1 }
26.
27. %%
```

Parser.pb

Damit wir den Wert direkt setzen können, ändern wir den Namen der UserData-Variable in Zeile 1 in "*result.Integer". In Zeile 25 speichern wir das Ergebnis der Rechnung in dieser Variablen.

```

1. IncludeFile "Parser_out.pb"
2. IncludeFile "Lexer_out.pb"
3.
4. Procedure EvalString(string.s)
5.   Protected result.Integer
6.
7.   *yyin = @string
8.   If yyparser(@result)
9.     ProcedureReturn result\i
10.  EndIf
11. EndProcedure
12.
13. Debug EvalString("2 + 3 * 4")

```

Rechner.pb

In Zeile 8 übergeben wir hier die lokale Variable "result" dem Parser, in diesem wird der Wert dann gesetzt. Mit "result\i" kann danach darauf zugegriffen werden.

Unäre Minus:

So nun wollen wir eine weitere Möglichkeit hinzufügen. Das Parsen des unären Minus, wie z.b. in dieser Eingabe "2 + -3 * 4". Die erste Idee wäre es vielleicht, dies direkt im Lexer einzubauen, die Regel für Zahlen so zu erweitern, dass auch negative Zahlen gelext werden. Wir wollen allerdings auch das Parsen einer solchen Eingabe "-----3" erlauben, sowie später, wenn wir Klammerausdrücke zulassen, auch solcher Eingaben "-(2 + 3)". Dies geht nur, indem wir im Parser eine weitere Regel hinzufügen:

```
exp: '-' exp { $$ = - $2 }
```

Das Problem an der Sache ist allerdings die Priorität dieses Operators. Das unäre Minus soll bei unserem Rechner (so wie es bei vielen anderen Programmiersprachen auch ist) eine höhere Priorität besitzen als "*" und "/". Das Zeichen "-" wird allerdings schon verwendet für das binäre Minus (a – b), welches eine niedrigere Priorität besitzt. Um dies zu umgehen, gibt es einen einfachen Trick:

```

1. %userdata *result.Integer
2.
3. %Structure
4.   int.i
5. %EndStructure
6.
7. %token.int NUMBER
8.
9. %left '+' '-'
10. %left '*' '/'
11. %right NEGATIVE
12.
13. %type.int eval exp
14. %start eval
15.
16. %%
17.
18. exp: ;[
19.     NUMBER          { $$ = $1 }
20.   | exp '+' exp { $$ = $1 + $3 }
21.   | exp '-' exp { $$ = $1 - $3 }
22.   | exp '*' exp { $$ = $1 * $3 }
23.   | exp '/' exp { $$ = $1 / $3 }
24.   | '-' exp %prec NEGATIVE { $$ = - $2 }
25. ;]
26.
27. eval: exp { *result\i = $1 }
28.
29. %%

```

Parser.pb

Wir setzen in Zeile 11 für ein neues Token "NEGATIVE" eine höhere Priorität als "*" und "/". Im Verbindung mit „%prec“ wird dieses Token dann in Zeile 24 dazu verwendet, dem Zeichen "-", temporär nur für diese Regel, eine andere Priorität, nämlich die von "NEGATIVE", zu zuweisen.

Dieses Token "NEGATIVE" wird dabei sonst nirgendwo verwendet, es wird also auch nicht vom Lexer zurückgegeben. Es dient lediglich dazu, die Priorität eines anderen Tokens zu ändern.

Das unäre Minus wird hier mit "%right" als rechts-assoziativ definiert, da es so in den meisten Programmiersprachen auch angegeben ist. Da der Operator nur einen Operanden besitzt, spielt es praktisch keine Rolle. "%left" würde genauso funktionieren.

Variablen und Klammern:

Wir wollen das Programm nun weiter ausbauen. Damit Klammerausdrücke und Variablenzuweisungen funktionieren. Dazu sind einige wenige Änderungen nötig.

```

1. %nocase
2.
3. %%
4.
5. <<EOF>>      { ProcedureReturn $<EOF> }
6. [ \t]        { } ; ignore char
7.
8. [0-9]+       { ;[
9.     yylval\int = Val(yytext)
10.    ProcedureReturn $NUMBER
11. ];]
12.
13. [A-Z]+       { ;[
14.     yylval\str = yytext
15.    ProcedureReturn $VARIABLE
16. ];]
17.
18. " := "       { ProcedureReturn $ASSIGN }
19. [+ \- * / ( )] { ProcedureReturn Asc(yytext) }
20.
21. .            { yyerror("Invalid char: "+yytext) }
22.
23. %%

```

Lexer.pb

Die neue Regel in Zeile 13 bis 16 erlaubt das Lexen einer Variable. Hier benutzen wir den Struktureintrag "str" der globalen Variable "yylval" um den Namen der Variable zu speichern. Diesen müssen wir gleich noch im Parser hinzufügen.

Zeile 18 ist für den Zuweisungsoperator zuständig. Wichtig ist hierbei, dass dieser aus zwei Zeichen besteht. Die Identifizierung vom Parser funktioniert also nur mit einer Konstante "\$ASSIGN", nicht mit dem Zeichen selbst (wie in Zeile 19).

In der Gruppe der Ein-Zeichen-Token in Zeile 19 fügen wir noch die Klammern "(" und ")" hinzu.

Im Parser sind nun weitere Änderungen nötig. Zuerst überlegen wir uns, wie wir den Wert einer zugewiesenen Variable speichern. Dazu verwenden wir eine Map. In Zeile 3 definieren wir diese.

In Zeile 7 fügen wir den angesprochenen Struktureintrag "str" hinzu, und in Zeile 12 typisieren wir das Token "VARIABLE". Das neue Token "ASSIGN" dagegen braucht keinen Typ, muss aber mittels Zeile 10 dennoch definiert werden.

Nun kommen wir zur Grammatik: Zuerst sei auf die Zeilen 46 bis 50 verwiesen. Hier definieren wir, dass die Eingabe nicht nur ein Ausdruck sein kann, sondern auch eine Zuweisung einer Variablen. Falls diese stattfindet, fügen wir der Map ein neues Element hinzu mit dem Namen der Variablen "\$1". Dieser Wert wurde im Lexer in Zeile 14 gesetzt. Den Wert dieses Elements setzen wir auf das Expression ("§3").

Durch die Regeln in den Zeilen 26 bis 33 kann nun ein Ausdruck auch eine Variable sein. Falls dies der Fall ist, wird erst geprüft, ob eine Variable mit dem Namen "\$1" bereits zugewiesen wurde. Falls nicht, rufen wir yyerror() auf und beenden das Parsen durch die Rückgabe von 0. Ansonsten setzen wir den Wert des Expressions auf den Wert der Variablen in Zeile 28.

Zeile 35 fügt nun noch eine Regel für Klammerausdrücke hinzu.

```

1. %userdata *result.Integer
2.
3. Global NewMap varValue.i()
4.
5. %Structure
6.     int.i
7.     str.s
8. %EndStructure
9.
10. %token ASSIGN
11. %token.int NUMBER
12. %token.str VARIABLE
13.
14. %left '+' '-'
15. %left '*' '/'
16. %right NEGATIVE
17.
18. %type.int eval exp
19. %start eval
20.
21. %%
22.
23. exp: ;[
24.     NUMBER          { $$ = $1 }
25.
26.     | VARIABLE {;[
27.         If FindMapElement(varValue(), $1)
28.             $$ = varValue()
29.         Else
30.             yyerror("Can't find the variable " + $1)
31.             ProcedureReturn 0
32.         EndIf
33.     ];]
34.
35.     | '(' exp ')' { $$ = $2 }
36.     | exp '+' exp { $$ = $1 + $3 }
37.     | exp '-' exp { $$ = $1 - $3 }
38.     | exp '*' exp { $$ = $1 * $3 }
39.     | exp '/' exp { $$ = $1 / $3 }
40.     | '-' exp %prec NEGATIVE { $$ = - $2 }
41. ;]
42.
43. eval: ;[
44.     exp { *result\i = $1 }
45.
46.     | VARIABLE ASSIGN exp {;[
47.         AddMapElement(varValue(), $1)
48.         varValue() = $3
49.         *result\i = $3
50.     ];]
51. ;]
52. %%

```