Shift Reduce Parser (Bottom up Parser)

Historie

Grundbegriffe

Tabellengesteuerter LR(1) Parser

Konstruktion der Elementmengen

Tabellenkonstruktion

Historie

Die ersten Compiler entstanden in den 50ern. Zunächst gab es keine Theorie!
Fortran wurde so definiert, dass der Compiler funktionierte.
Erste Ansätze bestanden darin, die innerste Klammerebene zu finden (O(n2))

Rekursiver Abstieg wurde nicht entdeckt, weil die Programmiersprachen keine Rekursion erlaubten.

LR-Parser entstanden in den 60ern (zunächst Operator Präzedenz)
In den 70ern wurden tabellengesteuerte LR-Parser und auch Werkzeuge entwickelt (yacc).

Grundbegriffe / Beispiel (inverse Rechtsableitung)
:

Regel 1: E → E + T
Regel 2: E → T
Regel 3: T → T * z
Regel 4: T → z

Schiebe so lange Token auf den Stack, bis dort eine brauchbare rechte Seite steht (handle).

Reduziere, die rechte Seite zur linken Seite.

Wenn Eingabe verarbeitet und Stack = Startsymbol: fertig!

handle = rechte Seite, die in einer Ableitung vorkommen kann.

Eingabe	Stack	Ableitung	Bemerkung
z+z*z$		z+z*z	shift
+z*z$	z	z+z*z	Regel 4
+z*z$	T	T+z*z	Regel 2
+z*z$	E	E+z*z	shift
z*z$	E+	E+z*z	shift
*z$	E+z	E+z*z	Regel 4
*z$	E+T	E+T*z	shift (!)
z$	E+T*	E+T*z	shift
$	E+T*z	E+T*z	Regel 3
$	E+T	E+T	Regel 1
$	E	E	Startsymbol

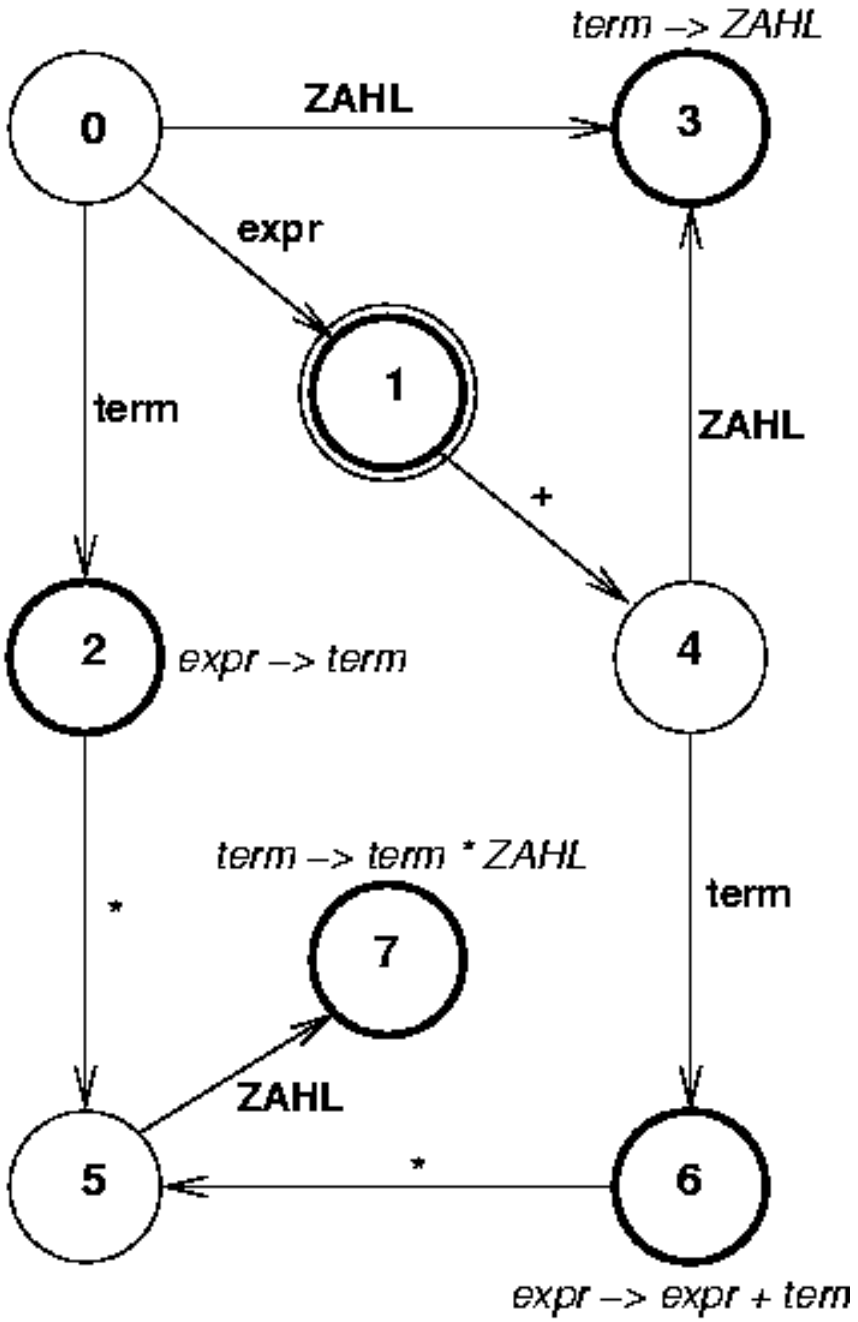
Zustandsdiagramm

Das Zustandsdiagramm macht deutlich, dass man Symbolfolgen durch Zustände ausdrücken kann.

Die Zustandsübergänge sind Shift- bzw. Goto-Aktionen.

Umrandete Zustände enthalten Reduce-Aktionen.

Der doppelt umrandeter Zustand ist ein Endzustand.



Die erkannte Symbolfolge, entspricht einem bestimmten Zustand.

Die Parsertabelle, steuert in Abhängigkeit von Lookahead-Symbol und aktuellem Zustand den Ablauf.

shift: überspringe das Eingabe-symbol und schiebe den Folge-zustand auf den Stack.

reduce: entferne so viele Zustände wie auf rechter Seite der Regel, Schiebe den Folgezustand auf den Stack (ergibt sich durch Non-terminalsymbol und Zustand aus Goto-Tabelle)

Eingabe	Stack	Bemerkung
z+z*z$	0	shift 3
+z*z$	3 0	reduce 4
+z*z$	2 0	reduce 2
+z*z$	1 0	shift 4
z*z$	4 1 0	shift 3
*z$	3 4 1 0	reduce 4
*z$	6 4 1 0	shift 5
z$	5 6 4 1 0	shift 7
$	7 5 6 4 1 0	reduce 3
$	6 4 1 0	reduce 1
$	1 0	acc

Aktionstabelle

	z	+	*	$		E	T
0	sh 3	-	-	-		1	2
1	-	sh 4	-	acc			
2	-	red 2	sh 5	red 2			
3	-	red 4	red 4	red 4			
4	sh 3	-	-	-			6
5	sh 7	-	-	-			
6	-	red 1	sh 5	red 1			
7	-	red 3	red 3	red 3			

sh = shift <Zustand>
red = reduce <Regel>
acc = accept
= error

Goto-Tabelle

LR – Parser (Pseudocode)

Eingabe:

Eingabe-String w$

LR-Tabellen *aktion* und *goto*

Methode:

Anfangs hat der Parser den Startzustand S_0 auf dem Stack und w$ im Eingabepuffer.

```
ip := 0; push(0/S0)
loop
   s := top(); a := eingabe[ip]
   if aktion[s,a]=shift s' then
       push(s'/a)
       ip++
   else if aktion[s,a]=reduce A → β then
       hole |β| Zustände vom Stack
       s' := top()
       push(goto[s',A]/A)
       gib A→ β aus // oder führe die Übersetzungsaktion
                       der Regel aus
   else if aktion[s,a]=accept then
       return
   else
       error()
end-loop
```

Konstruktion der LR-Elementmengen

I0 = {A→ #E, E→ #E+T, E→ #T, T→ #T*z, T→ #z}
Sp(I0, E) =
I1 = {A→ E#, E→ E#+T}
Sp(I0, T) =
I2 = {E→T#, T→ T#*z}
Sp(I0, z) =
I3 = {T→ z#}
Sp(I1, +) =
I4 = {E→ E+#T, T→ #T*z, T→ #z}
Sp(I2, *) =
I5 = {T→ T*#z}
Sp(I4,T) =
I6 = {E→ E+T#, T→ T#*z}
Sp(I4, z) = I3
Sp(I5, z) =
I7 = {T→ T*z#}
Sp(I6, *) = I5

Follow(E) = {$, +}
Follow(T) = {$, +, *}

A == Startsymbol

LR(0)-Mengen - 1

1. Erweiterung der Grammatik G → G'

füge zu G das neue Startsymbol S_0 und die Produktion $S_0 \rightarrow S$ hinzu.

2. Berechnung von Hülle

function Hülle(**M**)
begin
  repeat
    for Element **A** → **α** **#** **B** **β** in M
      Produktion **B** → **γ** aus G
        füge **B** → **#** **γ** **zu M** hinzu
    until keine Veränderung mehr möglich
    return M
end

3. Berechnung von Sprung

```
function Sprung(M,X)
begin
   As= {}
   for              A→α # X β    in M
      füge Hülle(A→α X # β)   zu As hinzu
   return J
end
```

4. Berechnung der LR(0) Menge

```
C := Hülle({ S0→ S })
J = { C }
repeat
   for E aus C
        jedes Symbol X aus E
            füge Sprung(E,X) zu J hinzu
   until keine Veränderung mehr möglich
```

5. Konstruktion der SLR(1)-Tabelle

Eingabe: die erweiterte Grammatik G'
Ausgabe: die SLR-Tabellen Aktion und Sprung

1. Konstruiere C = {I_0, I_1, .., I_n} die Menge der LR(0) Elemente aus G'
2. Konstruiere Zustand i aus der Menge I_i:
   a) Wenn $A \rightarrow \alpha.a\beta$ in I_i und I_j=Sprung(I_i,a), setze Aktion[i,a| auf "*shift j*".
   b) wenn S_0->S# in I_i dann setze Aktion[i,$] auf "*accept*"
   c) wenn $A \rightarrow \alpha\#$ in I_i ist, dann setze Aktion[i,a| für alle a in Follow(A) auf "*reduce* $A \rightarrow \alpha$"
   d) Wenn Sprung(I_i,A) =I_j, setze Goto[i,A] = "*goto j*".
   e) setze alle freien Einträge auf "*error*"
3. Der Anfangszustand des Parsers ist der, der von der Menge konstruiert wird, die $S_0 \rightarrow$ #S enthält.

Varianten der Tabellenkonstruktion

Einfachster Fall: Unabhängig vom Lookahead. Dies nennt man auch **LR(0)**-Tabelle. Problem: Konflikte!!!

Notwendige Bedingung für reduce ist, dass das Lookaheadsymbol in der Followmenge des erkannten Nichterminalsymbols enthalten ist. **SLR(1)**

Allgemeinste Methode ergibt einen kanonischen **LR(1)** Parser.

Der Nachteil des kanonischen LR(1) Parsers besteht darin, dass neue Zustände erzeugt werden. Man kann aber zeigen, dass die neuen Zustände keine verbesserte Konfliktvermeidung sondern nur frühere Fehlererkennung liefern.

Kompromiss: LALR(1) Parser: die möglichen Lookahead-Symbole werden aus dem Kontext bestimmt. Zustände mit gleichen Elementen (aber unterschiedlichen Folgemengen) werden verschmolzen.

Compilerbauwerkzeuge basieren meist auf LALR(1).

Beispiel für Kontext

Die Grammatik wird um geklammerte Ausdrücke erweitert.

Jetzt gibt es Ausdrück innerhalb von Klammern mit den möglichen Folgesymbolen
{), + } und Ausdrücke außerhalb von Klammern mit den Folgesymbolen { $, + }

Regel 1: $E \rightarrow E + T$
Regel 2: $E \rightarrow T$
Regel 3: $T \rightarrow T * F$
Regel 4: $T \rightarrow F$
Regel 5: $F \rightarrow (E)$
Regel 6: $F \rightarrow z$

Beispiel zu LR und LALR

In {} steht die Menge der Lookahead-Symbole

```
I0 = {A→ #E{$}, E→ #E+T{$,+}, E→ #T{$,+}, T→ #T*F{$,+,*},
     T→ #F{$,+,*},F→ #z{$,+,*}, F→ #(E){$,+,*}}
Sp(I0, E) = I1 = {A→ E#{$}, E→ E#+T{$,+}}
Sp(I0, T) = I2 = {E→ T#{$,+}, T→ T#*F{$,+,*}}
Sp(I0, F) = I3 = {T→ F#{$,+,*}}
Sp(I1, +) = I4 = {E→ E+#T{$,+}, T→ #T*F{$,+,*}, T→ #F{$,+,*}}
Sp(I2, *) = I5 = {T→ T*#F{$,+,*}}
Sp(I4, T) = I6 = {E→ E+T#{$,+}, T→ T#*F{$,+,*}}
Sp(I4, F) = I3
Sp(I5, F) = I7 = {T→ T*F#{$,+,*}}
Sp(I6, *) = I5

Sp(I0, z) = I8 = {F→ z#{$,+,*}}
Sp(I0, () = I9 = {F→ (#E){$,+,*}, E→ #E+T{),+}, E→ #T{),+},
     T→ #T*F{),+,*}, T→ #F{),+,*},F→ #z{),+,*}, F→ #(E){),+,*}}
Sp(I9, E) = I10= {F→ (E#){),+,*}, E→ E#+T{),+}}
Sp(I10,)) = I11 = {F→ (E)#{),+,*}}
Sp(I9, T) = ??? = {E→ T#{),+}, T→ T#*F{),+,*}}    ??? = I2  / I12
```

Kanonisches LR(1): Folgesymbole kennzeichnen Zustand mit.
LALR(1): nur die Elemente kennzeichnen den Zustand.
In diesem Beispiel ist LALR(1) gleich SLR(1).

Präzdenz und Assoziativität

Regel 1: E → E + E %left +
Regel 2: E → E * E %left *
Regel 3: E → z

```
I0 = {A→ #E, E→ #E+E, E→ #E*E, E→ #z}
Sp(I0, E) = I1 = {A→ E#, E→ E#+E, E→ E#*E}
Sp(I0, z) = I2 = {E→ z#}
Sp(I1, +) = I3 = {E→ E+#E, E→ #E+E, E→ #E*E, E→ #z}
Sp(I1, *) = I4 = {E→ E*#E, E→ #E+E, E→ #E*E, E→ #z}
Sp(I3, E) = I5 = {E→ E+E#, E→ E#+E, E→ E#*E}
Sp(I3, z) = I2 = {E→ z#}
Sp(I4, E) = I6 = {E→ E*E#, E→ E#+E, E→ E#*E}
Sp(I4, z) = I2 = {E→ z#}
Sp(I5, +) = I3, Sp(I5, *) = I4
Sp(I6, +) = I3, Sp(I6, *) = I4
```

Zustand 5: reduce(E→ E+E) oder shift(+), shift(*)
Zustand 6: reduce(E→ E*E) oder shift(+), shift(*)