LL(1)-Stackautomat

- Komponenten
  - Eingabestring (Tokens)
  - Vorhersagestack (Grammatiksymbole)
  - Transitionstabelle (Produktionen)
    - Zeilen ≙ Nichtterminale
    - Spalten ≙ Token
- Warum reicht nicht ein endlicher Automat?
  - Eintrag ist eine Symbol*folge* (ein Symbol statt Folge ⇒ endl. Automat)
  - diese wird zur Erinnerung auf den Stack geschoben
  - höchstens eine Symbolfolge ⇒ deterministischer Kellerautomat
- Parseroperationen (Moves)
  - Vorhersage (in LR: Shift)
  - Match (in LR: Reduce)
  - Termination
  - Fehler

Beispielgrammatik

```
input → expression EOF
expression → term rest_expression
term → IDENTIFIER | parenthesized_expression
parenthesized_expression → '(' expression ')'
rest_expression → '+' expression | ε
```

Figure 2.53 A simple grammar for demonstrating top-down parsing.

Rule	FIRST set	FOLLOW set
input	{ IDENTIFIER '(' }	{ }
expression	{ IDENTIFIER '(' }	{ EOF ')' }
term	{ IDENTIFIER '(' }	{ '+' EOF ')' }
parenthesized_expression	{ '(' }	{ '+' EOF ')' }
rest_expression	{ '+' ε }	{ EOF ')' }

Parsetabelle für Beispiel

Rule	FIRST set	FOLLOW set
input	{ IDENTIFIER '(' }	{ }
expression	{ IDENTIFIER '(' }	{ EOF ')' }
term	{ IDENTIFIER '(' }	{ '+' EOF ')' }
parenthesized_expression	{ '(' }	{ '+' EOF ')' }
rest_expression	{ '+' ε }	{ EOF ')' }

Top of stack/state:	Look-ahead token				
	IDENTIFIER	+	(	)	EOF
input	expression		expression		
expression	term rest_- expression		term rest_- expression		
term	IDENTIFIER		parenthesized_- expression		
parenthesized_- expression			(expression)		
rest_expression		+ expression		ε	ε

Figure 2.65 Transition table for an LL(1) parser for the grammar of Figure 2.53.

Beispiel
Vorhersagestack und Input

```
Prediction stack:   expression ')' rest_expression EOF
Present input:      IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF
```

Figure 2.66 Prediction stack and present input in a push-down automaton.

Parsing mit Tabelle (NT auf Stack)

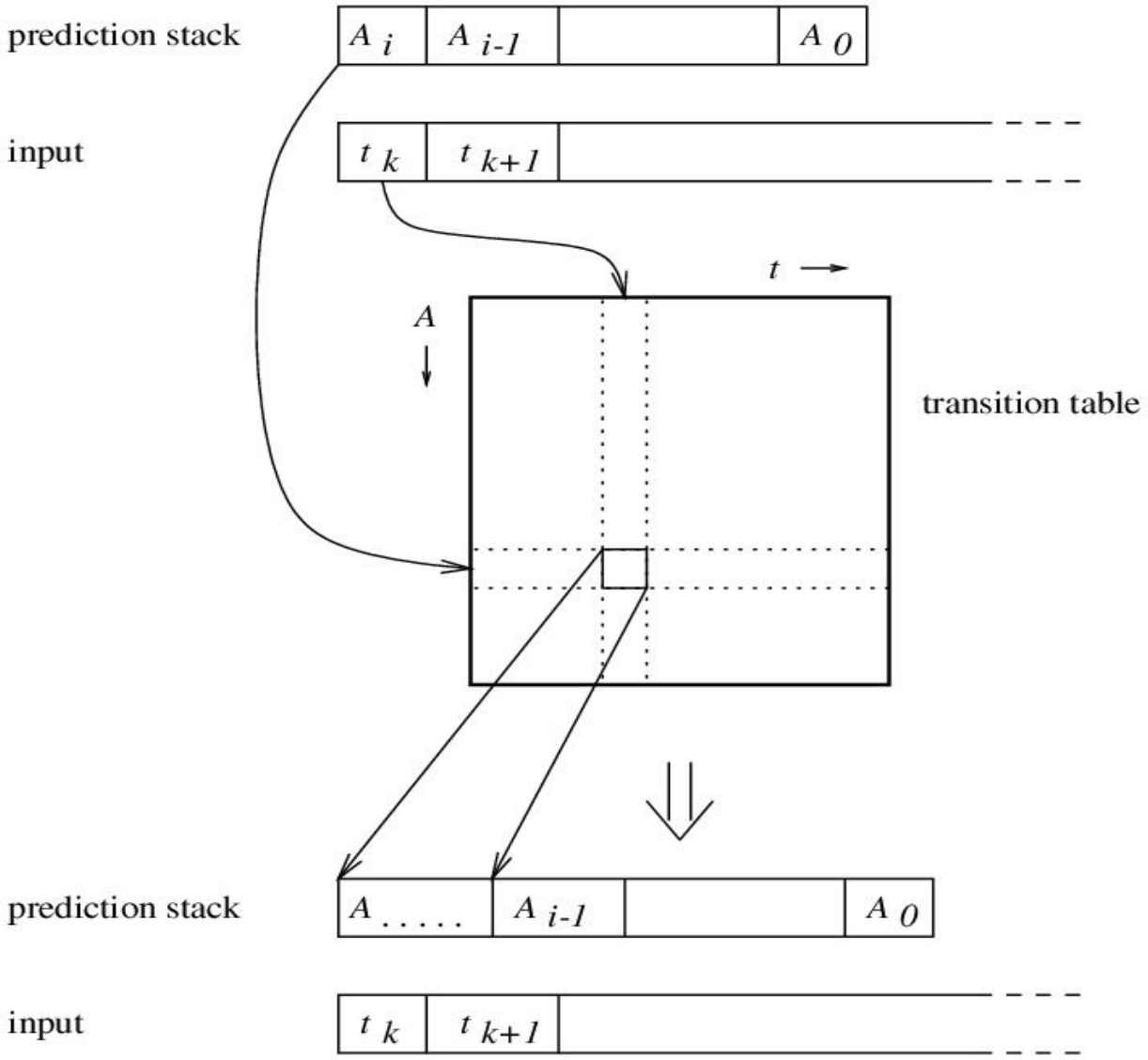

Figure 2.67 Prediction move in an LL(1) push-down automaton.

Parsing mit Tabelle (Token auf Stack)

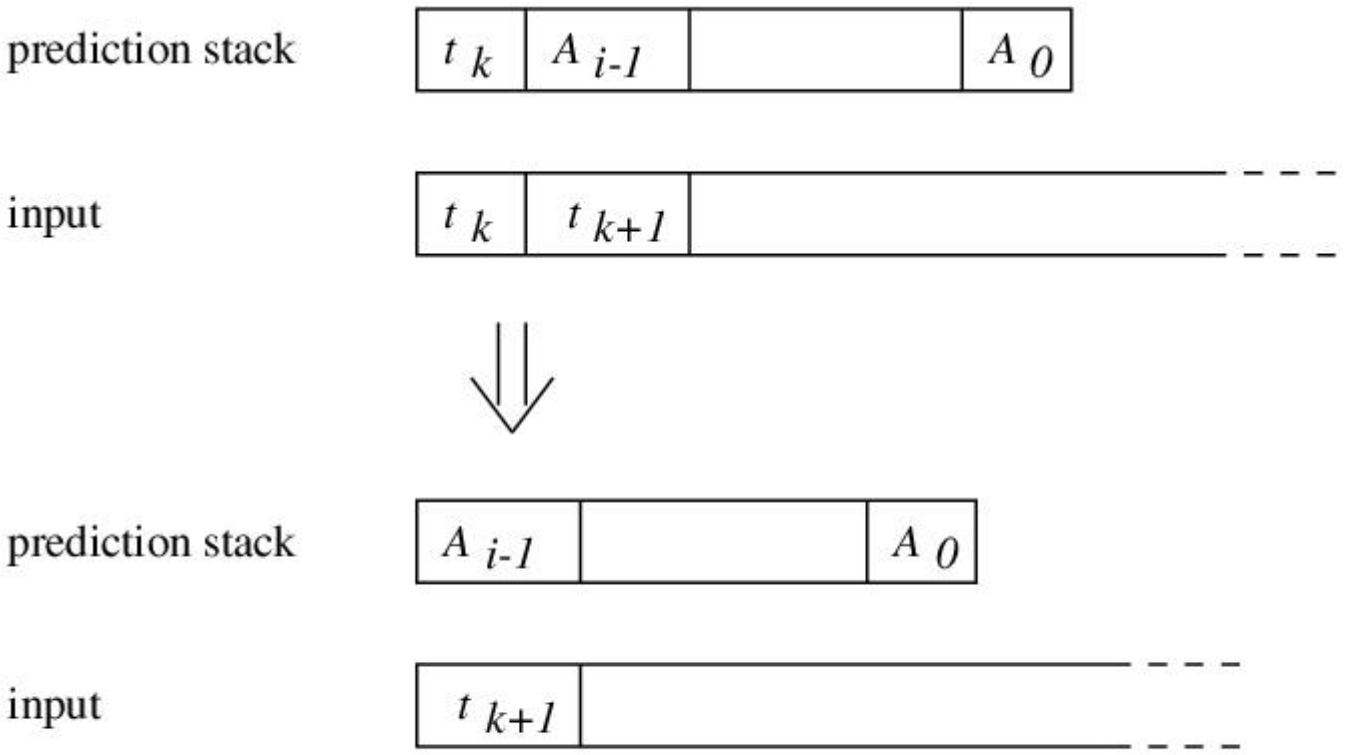

Figure 2.68 Match move in an LL(1) push-down automaton.

Aktionen des LL(1)-Parsers

Eingabe = Token *t*

- Vorhersage (Top of Stack = Nichtterminal *N*) (Abb. 2.67)
  - pop *N* vom Stack
  - lies in Tabelleneintrag unter (*N*,*t*)
  - push Eintrag (rückwärts) auf den Stack
  - LL(1) ⇒ höchstens ein Eintrag
  - kein Eintrag ⇒ Fehler
  - ggfs. semantische Aktion:
    - erzeuge Parsebaumknoten für die gepushten Symbole
    - trage diese als Nachfolger des Knotens für *N* ein
- Match (Top of Stack = Token *s*) (Abb. 2.68)
  - *s*=*t* ⇒ entferne Token aus Stack und Eingabe,
    ggfs. semantische Aktion: ergänze Attributwerte
  - *s*≠*t* ⇒ Fehler

LL(1)-Parsing-Algorithmus

```
IMPORT Input token [1..];        // from lexical analyzer

SET Input token index TO 1;
SET Prediction stack TO Empty stack;
PUSH Start symbol ON Prediction stack;

WHILE Prediction stack /= Empty stack:
    Pop top of Prediction stack into Predicted;
    IF Predicted is a Token class:
        // Try a match move:
        IF Predicted = Input token [Input token index] .class:
            SET Input token index TO Input token index + 1; // matched
        ELSE Predicted /= Input token:
            ERROR "Expected token not found: ", Predicted;
    ELSE Predicted is a Non-terminal:
        // Try a prediction move, using the input token as look-ahead:
        SET Prediction TO
            Prediction table [Predicted, Input token [Input token index]];
        IF Prediction = Empty:
            ERROR "Token not expected: ", Input token [Input token index]];
        ELSE Prediction /= Empty:
            PUSH The symbols of Prediction ON Prediction stack;
```

Figure 2.69 Predictive parsing with an LL(1) push-down automaton.

LL(1)-Parsing, Beispiel

```
Initial situation:
Prediction stack:     input
Input:                '(' IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF

Prediction moves:
Prediction stack:     expression EOF
Prediction stack:     term rest_expression EOF
Prediction stack:     parenthesized_expression rest_expression EOF
Prediction stack:     '(' expression ')' rest_expression EOF
Input:                '(' IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF

Match move on '(':
Prediction stack:     expression ')' rest_expression EOF
Input:                IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF
```

Figure 2.70 The first few parsing moves for `(i+i)+i`.

Prädiktiver Parser oder Stackautomat?

- Vorteile eines prädiktiven (recursive-descend) Parsers
  - leicht zu bauen
  - einfache Aufnahme von semantischen Aktionen, deshalb
  - besser für schmale (narrow) Compiler (weniger aufwändig)
- Vorteile eines Stackautomaten (nicht rekursiv)
  - sprachunabhängig
  - ausgereifte Theorie
  - effektivere Fehlerbehandlung (über Stackinformation)

Fehlererkennung/-behandlung

- Erkennung
  - Gesuchter Eintrag in Parsetabelle leer ⇒ Fehler
- Behandlung
  - Top of Stack falsch
    - Top of Stack entfernen:
      erzeugt fehlerhafte Parsebäume

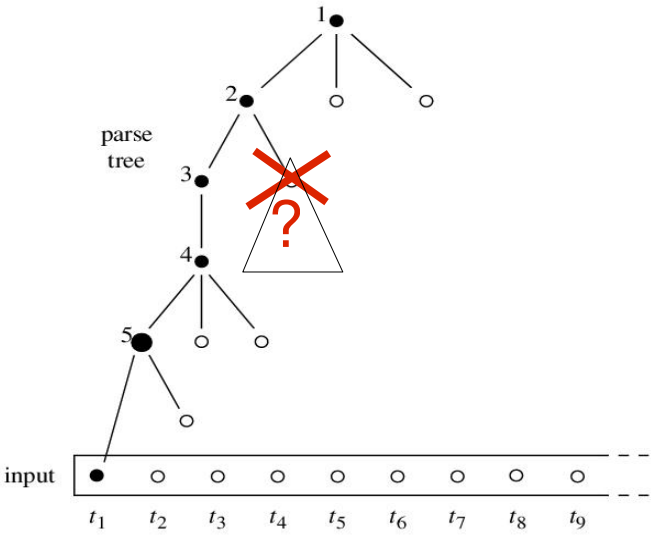

Figure 2.51 A top-down parser recognizing the first token in the input.

  - Eingabezeichen falsch
    (a) richtiges vorne anfügen: Gefahr der Nichttermination
    (b) einen Teil der Eingabe überlesen: ja, aber wieviel?

LL-Fehlerbehandlung

- Ziele
  - Termination: Verbrauch mindestens eines Eingabetokens
  - Parsebaum intakt lassen
    - keine Manipulation des Vorhersagestacks
    - Knoten kennzeichnen, bei denen Fehler aufgetreten sind
- Acceptable-Set-Methode (auch: Synchronizing Set)

  1. Parserzustand $-\{\text{Alg. } C\}\rightarrow$ Acceptable Set A ($\texttt{EOF} \in A$)
  2. überspringe alle Eingabetoken bis zu einem $t_A \in A$
  3. Parser $-\{\text{Alg. } R\}\rightarrow$ Zustand, in dem t_A akzeptiert werden kann

- Wahlmöglichkeiten (Einschränkung $C \Rightarrow R$)
  - Algorithmus C (Konstruktion des Acceptable Sets A)
  - Algorithmus R (Resynchronisation Zustand/Eingabe)

Acceptable Set: Konstruktionsmöglichkeiten

- Nicht hilfreiche Optionen
  - A = {EOF}: die gesamte Eingabe wird übersprungen
  - A = alle Tokens: es wird nichts übersprungen
- Panic Recovery (panische Behandlung)
  - funktioniert ohne Schritt 3 (Synchronisation)
  - Heuristik 1 (Grune/Bal/Jacobs/Langendoen)
    - A = korrekte Tokens z.Zt. der fehlerhaften Eingabe
    - überspringe alle fehlerhaften Eingabetokens (können viele sein, wenn Fehlerursache das Fehlen eines Tokens war)
    - Nachteil: beim Überspringen kann wichtige Struktur verloren gehen!
  - Heuristik 2 (Aho/Sethi/Ullman)
    - Top of Stack entfernen, falls es ein Nichtterminal (N) ist, zusätzlich
      - A = FOLLOW(N) $\cup$ Anfangssymbole höher stehender Konstrukte
      - überspringe alle Eingabezeichen, die nicht in A sind

Acceptable Set: Konstruktionsmöglichkeiten

- zwei Möglichkeiten
  - per Hand definiert
  - FOLLOW-Menge des auf dem Stack liegenden Symbols
    - Bsp.: Acceptable Set für `expression = {')',':',','}`
- übergehe inkorrekte Eingabe
- Dummy-Knoten im Parsebaum für das Nichtterminal auf dem Stack
- Problem: Definition von *A* ist nicht kontextspezifisch

  Bsp. in C: `a(b+int; c)`
  - Fehler: int
  - Semikolon wird nicht übersprungen, da in FOLLOW-Menge des arithmetischen Ausdrucks
  - ein Semikolon kann einem Ausdruck folgen, aber nicht in einer Parameterliste

Acceptable Set mit Continuations

- Continuation:
  - einzufügende Token, die den Erwartungen des Parsers entsprechen
  - (meistens) unendlich viele Möglichkeiten
  - Interesse nur an der kürzesten Ergänzung
- `Shortest production table[]:`
  - enthält: für jede Produktion und jedes Symbol die Länge L der kürzesten Continuation
  - Berechnung: Hüllenalgorithmus (Abb. 2.72)
    - Initialisierung: L(ε)=0, L(Token)=1, L(Nichtterminal)=∞
    - Inferenzregeln:
      1. kürzeste Länge einer Produktion :=
         Summe der kürzesten Längen ihrer Komponenten
      2. kürzeste Länge eines Nichtterminals :=
         Minimum der kürzesten Längen seiner Alternativen

Konstruktion der kürzesten Produktionen

Nichtterminal		Alternative	
`input`	2	`exp EOF`	2
`exp`	1	`term rest_exp`	1
`term`	1	`ID`	1
		`par_exp`	3
`par_exp`	3	`'(' exp ')'`	3
`rest_exp`	0	`'+' exp`	2
		ε	0

Data definitions:

1. A set of pairs of the form (production rule, integer).
2a. A set of pairs of the form (non-terminal, integer).
2b. A set of pairs of the form (terminal, integer).

Initializations:

1a. For each production rule $N \rightarrow A_1 ... A_n$ with $n>0$ there is a pair ($N \rightarrow A_1 ... A_n$, ∞).
1b. For each production rule $N \rightarrow \varepsilon$ there is a pair ($N \rightarrow \varepsilon$, 0).
2a. For each non-terminal N there is a pair (N, ∞).
2b. For each terminal T there is a pair (T, 1).

Inference rules:

1. For each production rule $N \rightarrow A_1 ... A_n$ with $n>0$, if there are pairs (A_1, l_1) to (A_n, l_n) with all $l_i<\infty$, the pair ($N \rightarrow A_1 ... A_n$, l_N) must be replaced by a pair ($N \rightarrow A_1 ... A_n$, l_{new}) where $l_{new}=\sum_{i=1}^{n} l_i$ provided $l_{new}<l_N$.

2. For each non-terminal N, if there are one or more pairs of the form ($N \rightarrow \alpha$, l_i) with $l_i<\infty$, the pair (N, l_N) must be replaced by (N, l_{new}) where l_{new} is the minimum of the l_is, provided $l_{new}<l_N$.

Figure 2.72 Closure algorithm for computing lengths of shortest productions.

Kontextspezifische Eingabe-*Reparatur*

- Schritt 1: bestimme Acceptable Set
  - Acceptable Set = FIRST-Mengen aller Stack-Konfigurationen beim Durchlauf der Continuation
  - Beispiel
    - Stack: `A B C EOF`
    - Eingabezeichen sei nicht in FIRST(A)
    - Ansatz: vergiss Eingabezeichen, die nach Synchronisation des Parsers gelesen werden. Was müsste man einfügen, wenn `EOF` in der Eingabe?

`A B C EOF`	
`p Q B C EOF`	A→pQ kürzeste Alternative für A
`Q B C EOF`	eingefügtes Token p matcht
`q B C EOF`	Q→q kürzeste Alternative für Q
`B C EOF`	eingefügtes Token q matcht
...	
`EOF`	EOF matcht

Kontextspezifische Eingabe-*Reparatur*

- Schritt 2: übergehe Eingabe bis zum nächsten Token im Acceptable Set
  dies ist nicht unbedingt in FIRST(Top of Stack)
- Schritt 3: versuche, normal fortzufahren ... solange das nicht geht ...
  - Top of Stack ist Terminal ⇒ füge dies in die Eingabe ein
  - Top of Stack ist Nichtterminal ⇒
    - finde in der Tabelle dessen kürzeste Alternative
    - ersetze Top of Stack durch deren rechte Seite

Termination spätestens, wenn die kürzeste Continuation durchlaufen

Algorithmus: Abb. 2.71

```
// Step 1: construct acceptable set:
SET Acceptable set TO Acceptable set of (Prediction stack);

// Step 2: skip unacceptable tokens:
WHILE Input token [Input token index] IS NOT IN Acceptable set:
    MESSAGE "Token skipped: ", Input token [Input token index];
    SET Input token index TO Input token index + 1;

// Step 3: resynchronize the parser:
SET the flag Resynchronized TO False;
WHILE NOT Resynchronized:
    Pop top of Prediction stack into Predicted;
    IF Predicted is a Token class:
        // Try a match move:
        IF Predicted = Input token [Input token index] .class:
            SET Input token index TO Input token index + 1; // matched
            SET Resynchronized TO True;          // resynchronized!
        ELSE Predicted /= Input token:
            Insert a token of class Predicted, including representation;
            MESSAGE "Token inserted of class ", Predicted;
    ELSE Predicted is a Non-terminal:
        // Do a prediction move:
        SET Prediction TO
            Prediction table [Predicted, Input token [Input token index]];
        IF Prediction = Empty:
            SET Prediction TO Shortest production table [Predicted];
        // Now Prediction /= Empty:
        PUSH The symbols of Prediction ON Prediction stack;
```

Figure 2.71 Acceptable-set error recovery in a predictive parser.

Fehlerbehandlung, Beispiel

Error detected, since `Prediction table [expression, '+']` is empty:
Prediction stack: `expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Shortest production for expression:
Prediction stack: `term rest_expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Shortest production for term:
Prediction stack: `IDENTIFIER rest_expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Token IDENTIFIER inserted in the input and matched:
Prediction stack: `rest_expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Normal prediction for rest_expression, resynchronized:
Prediction stack: `'+' expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Figure 2.74 Some steps in parsing `(i++i)+i`.