LR(1)-Syntaxanalyse

- Bei der LL(1)-Syntaxanalyse wird allein aufgrund des nächsten Tokens die zu verwendende Produktion ermittelt.
- Bei der LR(1)-Syntaxanalyse braucht diese Entscheidung erst gefällt werden, wenn die gesamte rechte Seite einer Produktion (plus ein Zeichen Vorausschau) gelesen wurde.
- Der Parser legt die eingelesenen Symbole nach und nach auf einen Stack. Wenn die Stacksymbole auf eine rechte Produktionsseite passen werden sie durch das Symbol der linken Seite ersetzt.
- Der Inhalt des Stacks besteht stets aus Grammatiksymbolen, die die bisher gelesene Eingabe erzeugen. Enthält der Stack nur das Startsymbol und wurde die gesamte Eingabe eingelesen, so ist die Analyse zuende.

Aktionen des LR(1)-Parsers

Der LR(1)-Parser kann zu jedem Zeitpunkt eine der folgenden beiden Aktionen durchführen:

- Ein weiteres Eingabesymbol lesen und auf den Stack legen. (*Shift*-Aktion)
- Eine Produktion auf die oberen Stacksymbole rückwärts anwenden, also den Stackinhalt $\sigma\gamma$ durch σX ersetzen, falls eine Produktion $X \to \gamma$ vorhanden ist. (*Reduce*-Aktion)

Eine Grammatik ist per definitionem LR(1), wenn die Entscheidung, welche der beiden Aktionen durchzuführen ist, allein aufgrund der bisher gelesenen Eingabe, sowie dem nächsten Symbol, getroffen werden kann.

LR(1)-Parser – Beispiel

Grammatik:
(0) S'→S$
(1) S→E
(2) E→E∧B
(3) E→E∨B
(4) E→B
(5) B→tt
(6) B→ff

Input: tt ∧ tt

Stack	Input	Aktion
	tt ∧ tt $	shift 'tt'
tt	∧ tt $	reduce (5) B→tt
B	∧ tt $	reduce (4) E→B
E	∧ tt $	shift '∧'
E ∧	tt $	shift 'tt'
E ∧ tt	$	reduce (5) B→tt
E ∧ B	$	reduce (2) E→E∧B
E	$	reduce (1) S→E
S	$	accept

LR(1)-Syntaxanalyse

- Aus technischen Gründen wird die Grammatik um eine Regel S'→S$ erweitert.
- Die Eingabe wird von links nach rechts verarbeitet, dabei wird eine Rechtsableitung vollzogen, und die Aktion hängt vom Stack und dem ersten Symbol der verbleibenden Eingabe ab.
  - LR(1): **l**eft-to-right parsing, **r**ight-most derivation, 1 token lookahead
- Die Rechtsableitung wird rückwärts vollzogen (Bottom-up-Ansatz).
- Wie wird bestimmt, welche Aktionen durchgeführt werden soll?

Wann soll man mit $X \to \gamma$ reduzieren?

Wenn γ oben auf dem Stack liegt ($\gamma \in (\Sigma \cup V)^*$) und außerdem angesichts der bisher gelesenen Eingabe und des nächsten Symbols keine Sackgasse vorhersehbar ist:

$$L^{\text{Reduce by } X \to \gamma \text{ when } a} = \{\sigma\gamma \mid \exists w.\ S \Rightarrow^*_{\mathbf{rm}} \sigma X a w\}$$

Wann soll man das nächste Symbol a einlesen?

Wenn es eine Möglichkeit gibt, angesichts des aktuellen Stackinhalts später eine Produktion anzuwenden.

$$L^{\text{Shift } a} = \{\sigma\alpha \mid \exists w.\ \exists (X \to \alpha a \beta) \in P.\ S \Rightarrow^*_{\mathbf{rm}} \sigma X w\}$$

Es liegt eine LR(1)-Grammatik vor genau dann, wenn diese Shift- und Reduce-Mengen disjunkt sind, also wenn die erforderliche Entscheidung eindeutig getroffen werden kann.

Das Wunder der LR(1)-Syntaxanalyse

Die Mengen $L^{\text{Reduce by } X\to\gamma \text{ when } a}$ und $L^{\text{Shift } a}$ sind regulär.
Es existiert also ein endlicher Automat, der nach Lesen des Stackinhaltes anhand des nächsten Eingabesymbols entscheiden kann, ob ein weiteres Symbol gelesen werden soll, oder die obersten Stacksymbole mit einer Produktion reduziert werden sollen und wenn ja mit welcher.

Konstruktion des Automaten

Der Automat wird zunächst nichtdeterministisch konzipiert.
Die Zustände haben die Form $(X{\rightarrow}\alpha.\beta, a)$ wobei $X \rightarrow \alpha\beta$ eine Produktion sein muss und a ein Terminalsymbol oder „?“ ist. Solch ein Zustand heißt „LR(1)-Item“.
Die Sprache, die zum Erreichen des Items $(X{\rightarrow}\alpha.\beta, a)$ führen soll, ist:

$$L(X \rightarrow \alpha.\beta, a) = \{\gamma\alpha \mid \exists w.\ S \Rightarrow^*_{\mathbf{rm}} \gamma X a w\}$$

$$L(X \rightarrow \alpha.\beta, ?) = \{\gamma\alpha \mid \exists w.\ S \Rightarrow^*_{\mathbf{rm}} \gamma X w\}$$

also gilt insbesondere:

$$L^{\text{Reduce by } X\rightarrow\gamma \text{ when } a} = L(X \rightarrow \gamma., a)$$
$$L^{\text{Shift } a} = \bigcup_{(X\rightarrow\alpha a\beta)\in P, c\in\Sigma\cup\{?\}} L(X \rightarrow \alpha.a\beta, c)$$

Jetzt muss man nur noch die Transitionen so bestimmen, dass tatsächlich diese Sprachen „erkannt“ werden:

Transitionen des Automaten

Erinnerung:

$$L(X \to \alpha.\beta, a) = \{\gamma\alpha \mid \exists w.\ S \Rightarrow^*_{\mathbf{rm}} \gamma X a w\}$$

- $(X \to \alpha.s\beta, a) \longrightarrow^{s} (X \to \alpha s.\beta, a)$, $s \in \Sigma \cup V$,
- $(X \to \alpha.Y\beta, a) \longrightarrow^{\epsilon} (Y \to .\gamma, b)$, falls $Y \to \gamma$ und $b \in \mathrm{FIRST}(\beta a)$
- Startzustand des Automaten: $(S' \to .S\$, ?)$

Man zeigt durch Induktion, dass der so definierte Automat tatsächlich die gewünschten Sprachen erkennt.

Aktionen

Nun determinisiert man den Automaten und erhält so Mengen von LR(1)-Items als Zustände des Automaten. Der Automat wird auf den Stacksymbolen ausgeführt.

Enthält der Endzustand das Item $(X \to \gamma., a)$ und ist das nächste Eingabesymbol a, so reduziert man mit $X \to \gamma$.

Enthält der Endzustand das Item $(X \to \alpha.a\beta, c)$ und ist das nächste Eingabesymbol a, so wird geshiftet.

Gibt es mehrere Möglichkeiten, so liegt ein Shift/Reduce, beziehungsweise ein Reduce/Reduce-Konflikt vor und die Grammatik ist nicht LR(1).

Konflikte in JavaCUP

Von JavaCUP gemeldete Konflikte muss man ernst nehmen; in den meisten Fällen deuten sie auf Fehler in der Grammatik hin.

Nützliche JavaCUP-Optionen zum Finden von Fehlern:

- `-dump` gibt alle Automatenzustände und die Parse-Tabelle aus.
- `-expect` *n* erzeugt einen Parser auch bei maximal *n* Konflikten. Er wird im Zweifel immer Shiften und ansonsten weiter oben geschriebenen Regeln Priorität einräumen (wie beim Lexer).

Einziger Fall, bei dem solch ein Konflikt sinnvoll ist:
„dangling else“:

$$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S$$
$$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S$$

Optimierung: Parser-Tabelle

- Annotiere Stackeinträge mit erreichtem Automatenzustand (in der Praxis lässt man die ursprünglichen Stacksymbole ganz weg und arbeitet mit Automatenzuständen als Stackalphabet).
- Konstruiere Tafel, deren Zeilen mit Zuständen und deren Spalten mit Grammatiksymbolen indiziert sind. Die Einträge enthalten eine der folgenden vier *Aktionen*:

Shift(*n*)	*„Shift" und gehe in Zustand n;*
Goto(*n*)	*Gehe in Zustand n;*
Reduce(*k*)	*„Reduce" mit Regel k;*
Accept	*Akzeptiere.*

  Leere Einträge bedeuten Syntaxfehler.

Der LR(1)-Algorithmus

- Ermittle Aktion aus der Tabelle anhand des obersten Stackzustands und des nächsten Symbols.
- Ist die Aktion. . .

  Shift(n): Lies ein Zeichen weiter; lege Zustand n auf den Stack.

  Reduce(k):
  - Entferne soviele Symbole vom Stack, wie die rechte Seite von Produktion k lang ist,
  - Sei X die linke Seite der Produktion k:
  - Finde in Tabelle unter dem nunmehr oben liegenden Zustand und X eine Aktion „**Goto(*n*)**“;
  - Lege n auf den Stack.

  Accept: Ende der Analyse, akzeptiere die Eingabe.

LR(1)-Parser mit Tabelle – Beispiel

Grammatik:

(0) S'→S$
(1) S→E
(2) E→E∧B
(3) E→E∨B
(4) E→B
(5) B→tt
(6) B→ff

Input: tt ∧ tt

Stack	Input	Aktion
0	tt ∧ tt $	shift 'tt'
0 2_{tt}	∧ tt $	reduce (5) B→tt
0 4_{B}	∧ tt $	reduce (4) E→B
0 3_{E}	∧ tt $	shift '∧'
0 3_{E} $5_{\wedge}$	tt $	shift 'tt'
0 3_{E} $5_{\wedge}$ 2_{tt}	$	reduce (5) B→tt
0 3_{E} $5_{\wedge}$ 7_{B}	$	reduce (2) E→E∧B
0 3_{E}	$	reduce (1) S→E
0 9_{S}	$	accept

	∧	∨	ff	tt	$	E	B	A
0			s1	s2		g3	g4	g9
1	r6	r6			r6			
2	r5	r5			r5			
3	s5	s6			r1			
4	r4	r4			r4			
5			s1	s2			g7	
6			s1	s2			g8	
7	r2	r2						
8	r3	r3						
9					acc			

LALR(1) und SLR

LR(1)-Tabellen sind recht groß (mehrere tausend Zustände für typische Programmiersprache).

LALR(1) ist eine heuristische Optimierung, bei der Zustände, die sich nur durch die Vorausschau-Symbole unterscheiden, identifiziert/zusammengelegt werden. Eine Grammatik heißt LALR(1), wenn nach diesem Prozess keine Konflikte entstehen.

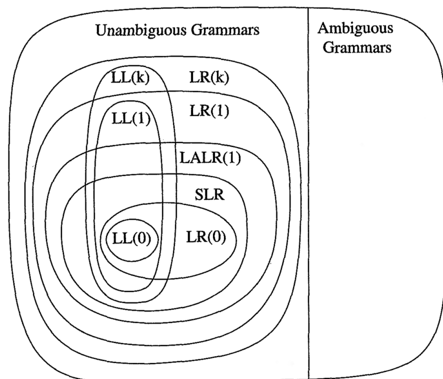
Bei SLR wird auf Vorausschau-Symbole in den Items verzichtet, stattdessen verwendet man FOLLOW-Mengen, um Konflikte aufzulösen.

Zusammenfassung LR-Syntaxanalyse

- **Bottom-up parsing:** LR-Parser haben einen Stack von Grammatiksymbolen, der der bisher gelesenen Eingabe entspricht. Sie legen entweder das nächste Eingabesymbol auf den Stack („Shift“) oder wenden auf die obersten Stacksymbole eine Produktion rückwärts an („Reduce“).
- Die Shift/Reduce-Entscheidung richtet sich nach der bereits gelesenen Eingabe und dem nächsten Symbol. Eine **LR(1)-Grammatik** liegt vor, wenn diese Entscheidung in eindeutiger Weise möglich ist. Die Entscheidung lässt sich dann durch einen endlichen Automaten automatisieren, der auf dem Stack arbeitet. Um Platz zu sparen, werden nur die Automatenzustände auf dem Stack gehalten.
- LR-Parser sind **allgemeiner und effizienter** als LL-Parser, aber auch deutlich komplexer.
- LALR(1) und SLR sind heuristische Optimierungen von LR(1).

Überblick

Grammatikklassen



Sprachklassen

$$LL(1) \subsetneq LL(2) \subsetneq \cdots \subsetneq LR(1) = LR(2) = \cdots = L(DPDA) \subsetneq L(CFG)$$

Ihe heutige Aufgabe

Erweitern Sie Ihren MiniJava-Parser so, dass er einen abstrakten Syntaxbaum erzeugt.

- In der ZIP-Datei auf der Vorlesungsseite finden Sie alle Klassen, die zum abstrakten Syntaxbaum von MiniJava gehören.
- Schreiben Sie semantische Aktionen für jede Regel, so dass entsprechende AST-Knoten erzeugt und mit den Unterknoten verbunden werden.
- Testen Sie Ihren Parser, indem Sie den PrettyPrint-Visitor aus der ZIP-Datei verwenden, um das eingegebene MiniJava-Programm wieder auszugeben.