1 Antlr4

Antlr4 (Akronym für **AN**nother **T**ool for **L**anguage **R**ecognition) ist ein Parsergenerator...

1.1 Akzeptierte Grammatik

Antlr4 verwendet eine neuartige Parsing Strategie (ALL(*) genannt), die es erlaubt alle nicht linksrekursiven, kontextfreien Grammatiken zu akzeptieren. Eine Grammatik wird in erweiterter Backus-Nauer Form definiert, welche dann von Antlr4 auf Fehler überprüft wird. Dabei kann die eingegebene Grammatik direkt linksrekursiv sein, intern wird jedoch jegliche direkte Linksrekursion durch Umschreiben eliminiert. Dadurch können Parser für weitaus mehr Grammatiken generiert werden als für Parsing Techniken, die auf Token Lookaheads angewiesen sind. (LR(k), LL(k) etc.). Dies hat zur Konsequenz, dass der Algorithmus eine worst-case Zeitkomplexität von $O(n^4)$ aufweist. In **??** wird jedoch darauf hingewiesen, dass diese obere Schranke in der Praxis häufig nicht erreicht wird und nur von theoretischer Interesse ist. Der Autor des Berichts macht klar, dass ALL(*) um ein vielfaches schneller ist als vergleichbare Algorithmen wie z.B. Generalized LR Parsing, welches alle kontextfreien Grammatiken akzeptiert und in $O(n^3)$ **??** Eingaben parsen kann. In dieser Arbeit wird auf GLR Parsing in Abschnitt **??** separat eingegangen und auch ein Vergleich mit ALL(*) wird angestellt, um diese Behauptung unabhängig und speziell für die *C++* Version verifizieren zu können.
Im Folgenden soll die Funktionsweise von ALL(*) erläutert werden.

1.2 Funktionsweise

Antlr4 erzeugt auf rekursivem Abstieg basierende Parser (die Idee des rekursiven Abstiegs wird in Abschnitt **??** erklärt). Während die meisten gängigen Parsingstrategien darauf angewiesen sind statisch, d.h. vor der Erzeugung des Parsers, Lookahead Tokenmengen zu konstruieren (Parser für LL(k), LR(k) Grammatiken zum Beispiel) verschiebt ALL(*) diese Analyse auf die Laufzeit. Am besten ist der Algorithmus an einem Beispiel erklärt. Man betrachte dazu Grammatik 1.2 (eine vereinfachte Version von der in **??** vorgestellten).

```
stmt -> expr '=' expr ';' | expr ';'
expr -> expr '*' expr | expr '(' expr ')' | ID
```

Ein Predictive Parser für *stmt* entscheidet basierend auf der Lookahead Tokenmenge $'='$ und $';'$, ob nach dem Parsen von expr, die linke oder die rechte Alternative als nächstes erkannt (besseres Wort für Parsen??) werden muss, unter der Annahme, dass *expr* von diesem ebenso geparst werden kann. Antlr4 generiert ähnlichen Code für *stmt*, siehe dafür Listing 1. Die Funktion *adaptivePredict* empfängt als Parameter das aktuelle Nichtterminal und den Callstack und bestimmt die zu parsende Alternative. Hierzu wird inkrementell ein deterministischer endlicher Automat gebaut, welcher die Lookahead Tokenmenge repräsentiert. Die Konstruktion soll gleich erklärt werden, zuerst ist es wichtig ein

Verständnis für die Funktionsweise des DEAs zu bekommen. Die richtige Produktion wird berechnet, indem vom Startzustand ausgehend die Ausführung des DEAs simuliert wird. Dieser erhält als Eingabe die noch zu parsenden Zeichen, d.h. die Terminale, die man noch vorausschauen kann. Die Zustandsübergänge sind mit möglichen Lookahead Terminalen beschriftet. Akzeptierende Zustände verweisen eindeutig auf die zu parsende Produktion. Pfade vom Startzustand zu einem Endzustand geben daher ebenso eindeutig die Terminalfolge an, welche als Vorausschaumenge in der Eingabe enthalten sein muss, damit die entsprechende Produktion genommen wird. Man betrachte als Beispiel Abbildung **??** (b) und nehme weiterhin an, dass die Eingabe $a = b(x);$ sei. Die Funktion *adaptivePredict* simuliert die Ausführung des DEAs: a ist ein Identifier und man wechselt in den Zustand s_1. Das nächste Zeichen ist ein $=$, also wird in Zustand : 1 gewechselt, welcher als akzeptierender Zustand signalisiert, dass die erste Produktion genommen werden soll (siehe hierfür Listing 1). Analog wird auch beim Parsen von $b(x)$ vorgegangen.
Nun soll die Konstruktion erläutert werden, dazu wird ein Eingabestring von $x = y$ angenommen. Beim ersten Aufruf von *adaptivePredict* für ein Nonterminal wird der zugehörige DEA mit einem Startzustand initialisiert. Nun werden alle alternativen Produktionen nacheinander (im Gleichschritt sozusagen) pseudoparallel geparst. Das heißt zuerst wird versucht für die alternative Produktion 1 ($stmt- > expr' =' expr';'$) *expr* zu erkennen. hier weiter

```
void stmt () {
    switch ( adaptivePredict ("stmt", call stack)) {
        case 1: expr(); match('='); expr(); match (';'); break;
        case 2: expr(); match (';'); break;
    }
}
```

Listing 1: Code zum Erkennen von stmt