Parser4Kids – interaktive Lernumgebung

Parser4Kids vermittelt die grundlegende Funktionsweise eines Parsers auf eine intuitive und spielerische Art anhand des vereinfachten Modells eines Fertigbauhauses.
Parser4Kids richtet sich an Lernende an berufsbildenden Schulen, Gymnasien und Fachhochschulen.

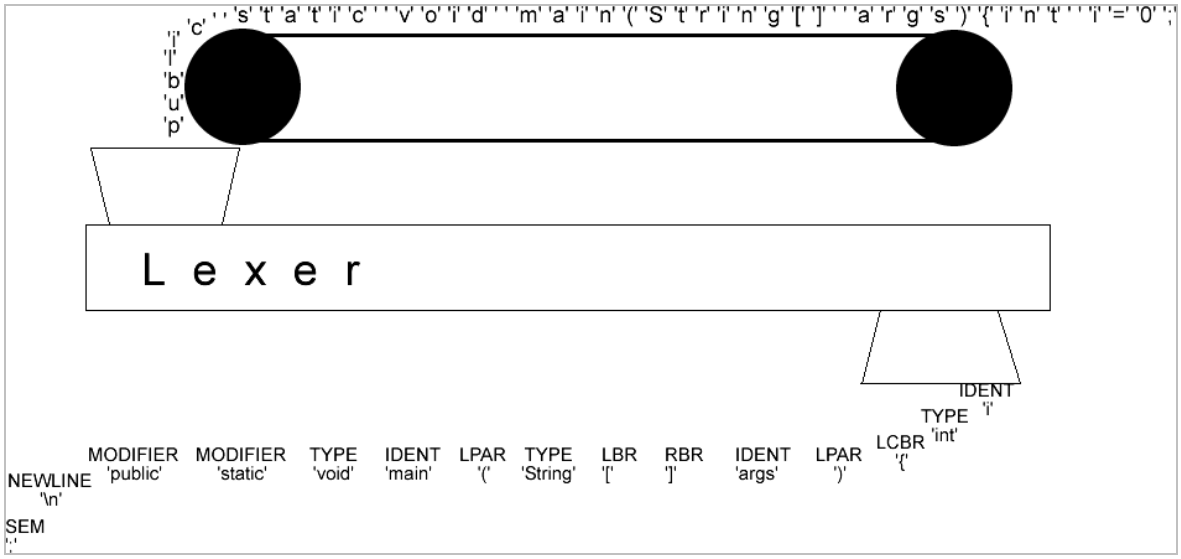
1. Theoretische Hintergründe

Ein Parser führt folgende Operationen aus:

- lexikalische Analyse
- syntaktische Analyse

Lexikalische Analyse

Vorbedingung	Reguläre Ausdrücke, welche die zu erkennenden Token beschreiben
Input	Folge von Zeichen (Zeichenstrom)
Output	„verdichteter Quellcode“. Eine Folge von Terminalsymbolen, genannt Token.



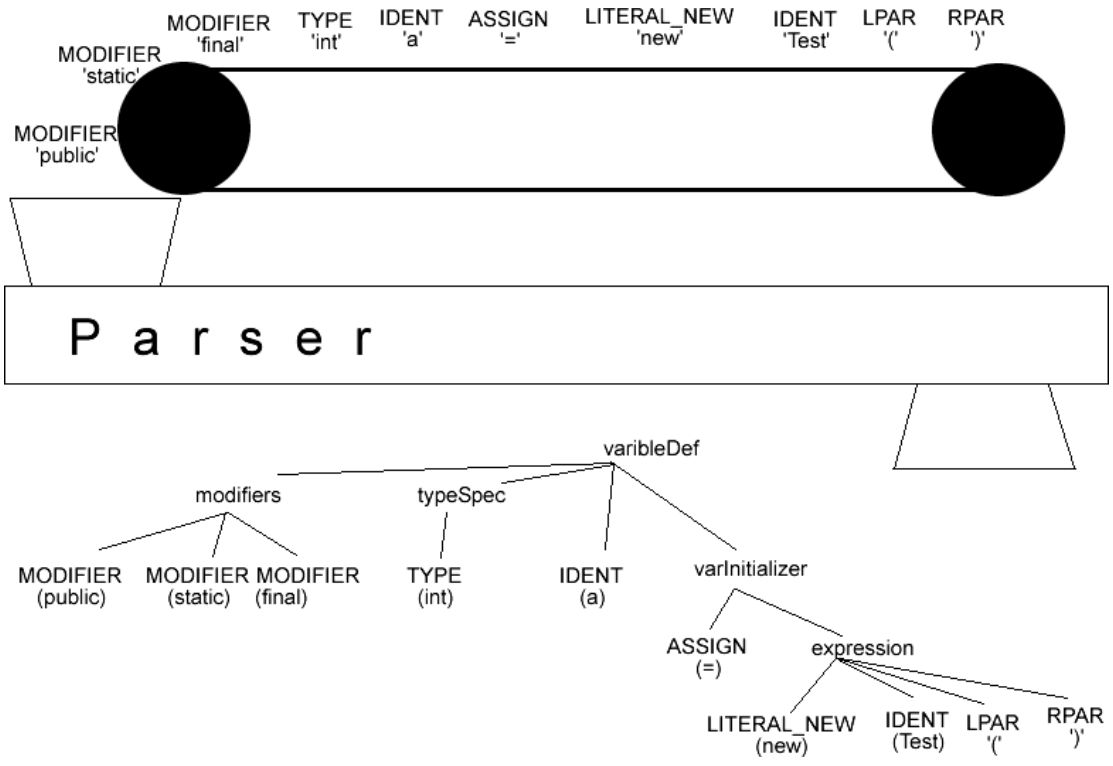
Durch die lexikalische Analyse wird eine Folge von Eingabezeichen in atomare Einheiten transformiert. Ein Token umfasst all diejenigen Zeichen, die in der anschliessenden Syntaxanalyse als gleich angesehen werden (man spricht auch von syntaktischen Kategorien).
Das Ziel der lexikalischen Analyse ist somit das “Verdichten” des Quelltextes. Beispielsweise ist es für das Prüfen der korrekten Syntax irrelevant, ob die Zahl 1 oder 32434 eingelesen wurde. Wichtig ist nur, dass es sich um eine Zahl handelt.

Bevor die lexikalische Analyse beginnen kann, müssen die Zeichenfolgen definiert werden, die als Token erkannt werden müssen. Diese Token werden meist durch reguläre Ausdrücke definiert.
Nach der Definition solcher regulärer Ausdrücke kann ein Programm zur lexikalischen Analyse automatisch generiert werden (bekannte Beispiele solcher Generatoren: Lex oder ANTLR, vgl. [Lex 2006], [ANTLR 2006]).

Der Output der lexikalischen Analyse dient als Input für die syntaktische Analyse.

Syntaktische Analyse

Vorbedingung	Grammatik, die ein korrektes Programm beschreibt (meist eindeutig und kontextfrei [Typ-2 Grammatik mit Einschränkungen, z.B. LL(k) oder LR(k)]). Die terminalen Symbole der Grammatik entsprechen den (vom Lexer generierten) Token.
Input	Die in der lexikalischen Analyse generierten Token
Output	Ableitungsbaum oder AST (Abstract Syntax Tree)



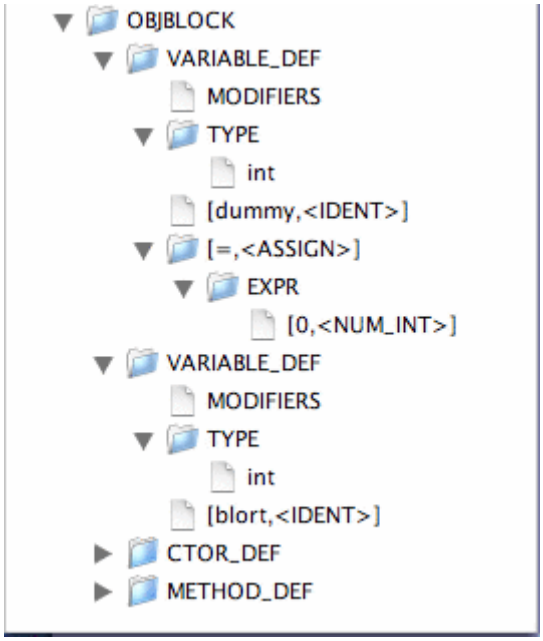
Der Ableitungsbaum wird aufgrund einer gegebenen Grammatik erstellt. Um Backtracking oder auch endlose Iterationen zu vermeiden, muss die verwendete Grammatik über gewisse Eigenschaften verfügen (Beispiel: Grammatiken vom Typ LL(k) oder LR(k), für Details, vgl. [Güting 1999]):

- LL(k): kontextfreie Grammatik mit bestimmten Eigenschaften, bei der durch Vorausschau von k Zeichen eine Analyse ohne Backtracking möglich ist. Es wird immer das am weitesten links stehende nichtterminale Symbol ersetzt. Dies

entspricht einer top-down Analyse (der Ableitungsbaum wird ausgehend von der Wurzel aufgebaut)

- LR(k): Es wird immer das am weitesten rechts stehende nichtterminale Symbol ersetzt. Dies entspricht einer bottom-up Analyse (der Ableitungsbaum wird aus partiellen Ableitungsbäumen – ausgehend von den Blattknoten - erzeugt). LR(k) Parser sind mächtiger, aber auch komplizierter als LL(k) Parser.

Die Darstellung des Ableitungsbaums könnte z.B. wie folgt umgesetzt werden (vgl. http://www.cs.usfca.edu/~parrt/course/652/lectures/AST.html):



2. Parser4Kids

Wie in den vorangegangenen Abschnitten angedeutet wurde, kann die Festlegung von Grammatiken zu folgenden Problemen führen:

- Mehrdeutigkeit: Zu einem gegebenen Terminalwort existieren verschiedene Ableitungsbäume (Bemerkung: Aufgrund einer vorgegebenen Grammatik kann man im voraus nicht entscheiden ob diese mehrdeutig ist)
- Grammatik ist nicht LL(k) oder LR(k)
- Unübersichtlichkeit: bereits bei relativ einfachen Problemen kann die Grammatik kompliziert werden

Zudem ist die Angabe einer Grammatik zeitaufwändig und alles andere als einfach (umso mehr für ungeübte Benutzer).
Dies hat zur Konsequenz, dass die Definition von Grammatiken vorgegeben werden muss. Wobei man im Expertenmodus die Möglichkeit zur Definition eigener Grammatiken vorsehen könnte.

Bei der syntaktischen Analyse wurde erwähnt, dass verschiedene Typen von Grammatiken in Frage kommen: LL(k), LR(k), u.a.
Bzgl. Einfachheit und Mächtigkeit ist aus meiner Sicht ein LL(k) Parser zu bevorzugen. Begründung:

- Die am meisten verbreiteten Parsergeneratoren (ANTLR, JavaCC, Coco/R) sind vom Typ LL(k).
- Viele bekannte Programmiersprachen können effizient durch LL(k) Parser analysiert werden (Java, Python, C++, C#, u.a.).
  Für Grammatikbeispiele, vgl. http://www.antlr.org/grammar/list.
- Der Ableitungsbaum (oder AST) wird top-down erstellt. Dies ist übersichtlicher und einfacher nachvollziehbar als der bottom-up Ansatz.

Im Idealfall sollte Parser4Kids unabhängig vom darunter liegenden Parser arbeiten. Als „Referenzimplementation“ erscheint uns TLR sinnvoll. Es handelt sich dabei um einen LL(k) Parser, der bereits seit vielen Jahren aktiv weiterentwickelt wird (vgl. [ANTLR 2006]).

3. Herausforderungen bei der Entwicklung von Parser4Kids

Die Entwicklung einer intuitiven Lernumgebung für Parser ist nicht einfach. Besondere Beachtung geschenkt werden muss:

- Erstellung einer Schicht, die den „echten“ Parser versteckt.
- Definition von Token (für den Lexer) und einer Grammatik für den Parser, die „jedes Kind“ versteht. Beispiele: Legostein, Fenster, Hauseingang, etc.
- Begriffe wie „Programmiersprache“, „Token“, „Lexer“ etc. dürfen im Beginnermodus nicht erwähnt werden. Es müssen sinnvolle Metaphern gefunden werden.

  Beispiele:

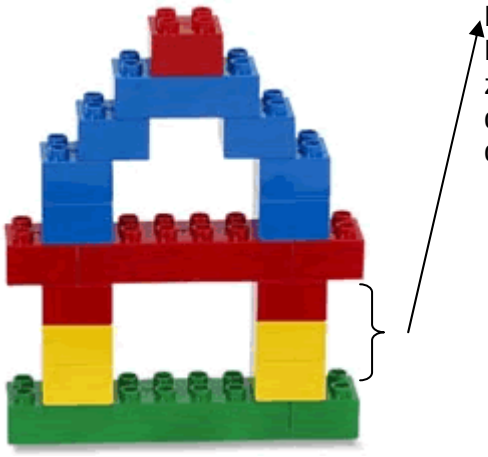
  - Zeichenstrom: verschiedenfarbige Legosteine, als Input für die unter Abbildung 1 dargestellte Maschine.

- Token: Anordnung von Legosteinen, die eine Einheit bilden, z.B. Wand, Fenster, etc.
- Parser: Ein „Baumeister“, der kontrolliert, ob es sich bei diesem Gebilde (= Programm) wirklich um ein Haus handelt. Dieser untersucht das Haus Stein um Stein. Der aktuell untersuchte Stein wird hervorgehoben (z.B. farblich markiert) und als Input für den Lexer oder Parser verwendet.
- Grammatik: Die Regeln müssen in Sätze gefasst werden. Beispiel: Anstelle von Wand -> (LegoStein)+(Fenster)*(LegoStein)+ würde man den Satz „Eine Wand kann Fenster enthalten (muss aber nicht). Diese dürfen nicht direkt am Anfang einer Wand beginnen“ verwenden.

4. Probleme bei Parser4Kids

Bereits jetzt können folgende Probleme identifiziert werden:

- Da der Lexer und der Parser das Gebilde (z.B. ein Legohaus) in einer gewissen Reihenfolge analysiert, muss diese festgelegt werden. Beispielsweise wäre dies „von unten links, zeilenweise“. Diese Reihenfolge bleibt jedoch relativ willkürlich (im Gegensatz zu Programmen, die immer von links nach rechts und von oben nach unten analysiert werden).
- Falls man einen „echten“ Parser (z.B. ANTLR) verwendet, muss infolgedessen auch die definierte Grammatik korrekt sein. Diese wird jedoch sehr schnell sehr kompliziert. Nur schon das Erkennen eines Fensters ist relativ schwierig, sobald sich dieses in der Höhe über mehrere Legosteine erstreckt.
  Ein weiteres Beispiel ist das Erkennen eines korrekten Daches (man könnte hier z.B. fordern, dass es seitlich nicht über eine Hauswand ragt und symmetrisch sein muss. Dies würde aber ebenfalls zu einer komplizierten Grammatik führen).



Aus unserer Sicht ist es durchaus möglich, Grammatiken für ein solches Legohaus zu erstellen. Z.B. ist unter http://www.antlr.org/grammar/list auch eine Grammatik für Python enthalten. Diese ist daher so schwierig zu implementieren, da bei Python bekanntlich die Leerzeichen zur Syntax gehören. Das wäre beim Erkennen unseres Fensters beim Legohaus ebenfalls der Fall (man muss wissen, dass man eine Zeile zuvor an einer bestimmten Stelle zwei „leere“ Legosteine eingelesen hat).

5. Fazit aus den Vorüberlegungen

Die Umsetzung von Parser4Kids, basierend auf einem echten Parser, ist komplizierter als erwartet: Objekte aus der realen Welt haben zwar den Vorteil des Alltagsbezugs (Wiedererkennungseffekt), passen jedoch nur schwer in das zugrundeliegende Modell (Beispielproblem: in welcher Reihenfolge werden die Elemente eines Objekts analysiert?).

Um diesen grundlegenden Konflikt zwischen Alltagsbezug und theoretischer Korrektheit zu meistern, gibt es folgende Möglichkeiten:

- Änderung der Zielgruppe: Statt Parser4Kids wäre das Ziel Parser4Stundents (natürlich müsste man hier einen anderen Namen wählen).
- Man wählt einen Kompromiss:
  - Reale Beispiele (z.B. Legohaus) werden ohne echten Parser umgesetzt. Das Prinzip des Parsers kann man auch ohne echten Parser erklären.
  - Weitere Beispiele, z.B. das Analysieren von Programmen oder Sätzen der deutschen Sprache könnte man dann (z.B. im Fortgeschrittenen- oder Expertenmodus) mit einem echten Parser umsetzen.

6. Konkrete Umsetzung

6.1 Allgemeines

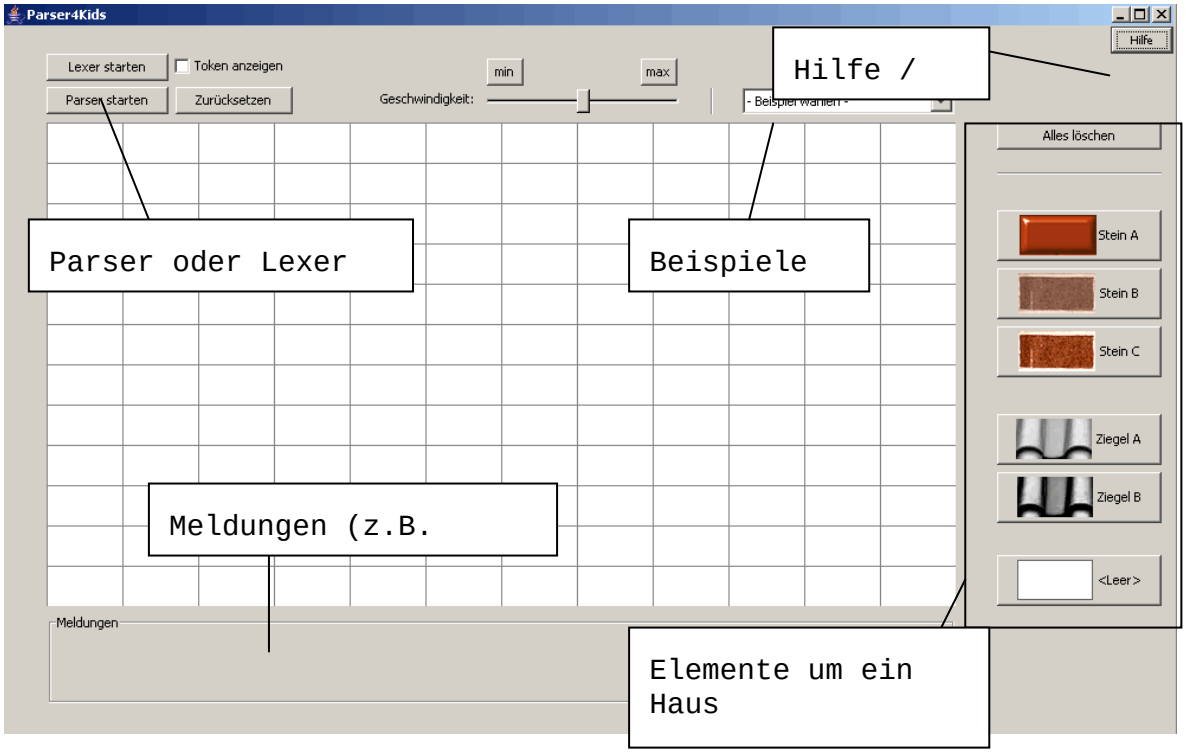
Die konkrete Umsetzung besteht aus dem bereits angesprochenen Kompromiss: Verzicht auf einen konkreten bereits existierenden Parser. Parser und Lexer werden selbst implementiert. Diese Entscheidung wurde nicht aufgrund des „not invented here Syndroms“ gefällt, sondern basiert auf folgenden Überlegungen:

- Die echten Parsergeneratoren arbeiten „verzahnt“, d.h. der Lexer generiert die Token und diese dienen als Input für den Parser. Der Parser arbeitet „versetzt“ und wartet nicht, bis der Lexer die syntaktische Analyse abgeschlossen hat. Dies erschwert die Visualisierung und Nachvollziehbarkeit jedoch massiv.
- Die Parsergeneratoren brauchen eine hohe Einarbeitungszeit und diese stand nicht zur Verfügung. Die Voraussetzung, den ganzen Vorgang visualisieren zu können hätte den Einarbeitungsvorgang erhöht, da man den Parser nicht als Blackbox verwendet. Beispielsweise muss man immer wissen, welches Token nun gelesen wird (um es visualisieren zu können). Zudem sind die Fehlermeldungen auf englisch und für unsere Ansprüche zu kompliziert (Das Abfangen und Interpretieren von diesen Fehlermeldungen macht aus meiner Sicht keinen Sinn, da dies bei jeder Version ändern kann).

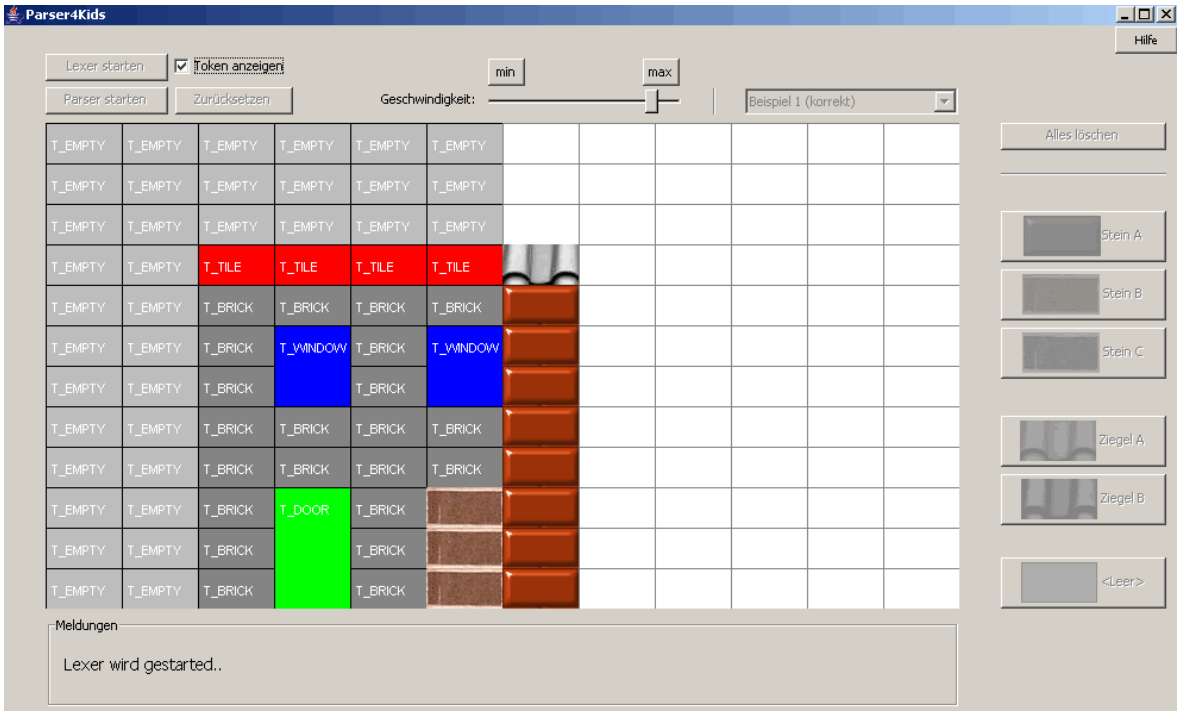
Für eine nächste Version von Parser4Kids wäre es evtl. interessant zu untersuchen, ob man die genannten Probleme nicht doch lösen kann. Aus unserer Sicht wäre ANTLR wohl am besten dafür geeignet.

6.2 Details

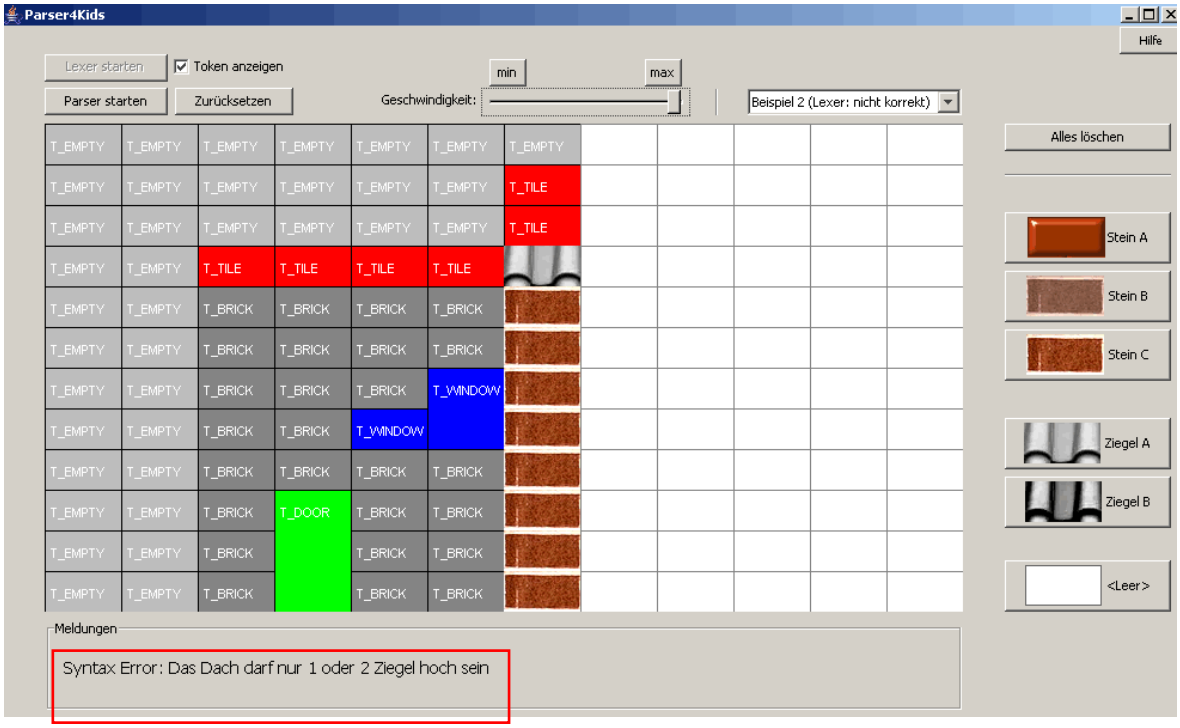
Für die Visualisierung wurde eine Desktopapplikation in Java (JDK Version >= 1.5) erstellt. Die Problemstellung bestand darin, ein Haus aus verschiedenen Elementen (Steine, Ziegel, Türen, Fenster) zu erstellen. Die „Korrektheit“ dieses Hauses wurde aufgrund von verschiedenen Regeln geprüft (Details können in der Hilfeseite der Applikation nachgelesen werden).



Nachdem ein Beispiel geladen wurde oder der Benutzer selbst ein Haus erstellt hat, erfolgt eine erste Prüfung durch den Lexer. Dabei ist immer klar ersichtlich, welches Element aktuell gelesen wurde: Alle gelesenen Elemente werden durch ein Tokensymbol ersetzt (z.B. T_BRICK für Stein oder T_TILE für Ziegel). Die Beschriftung der Tokensymbole kann angezeigt oder ausgeblendet werden. Zudem sind die Texte konfigurierbar.

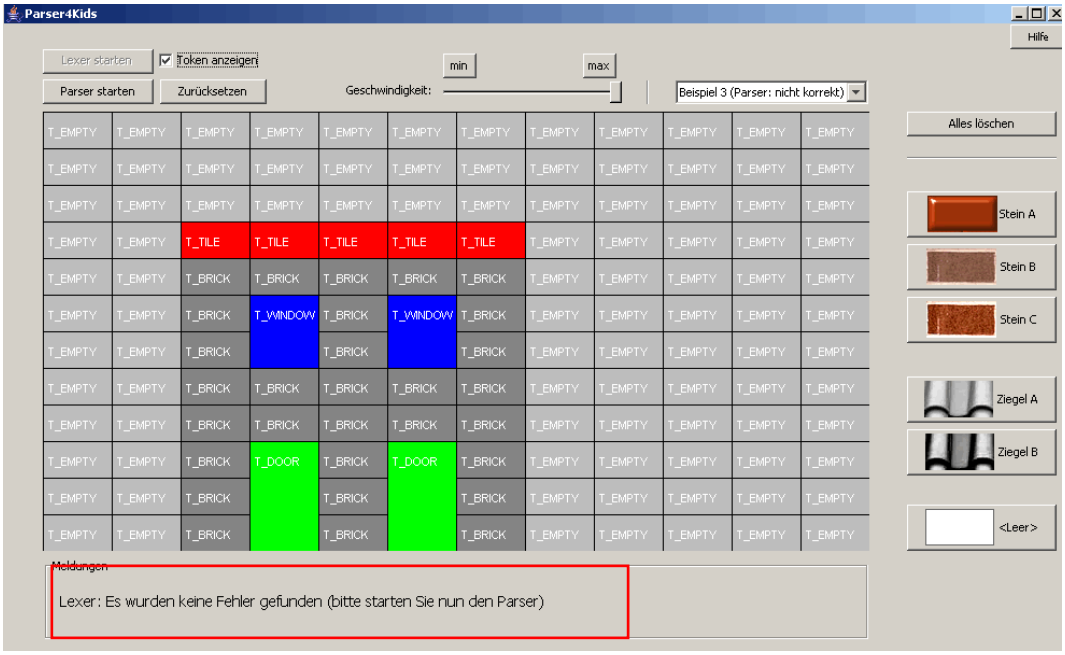


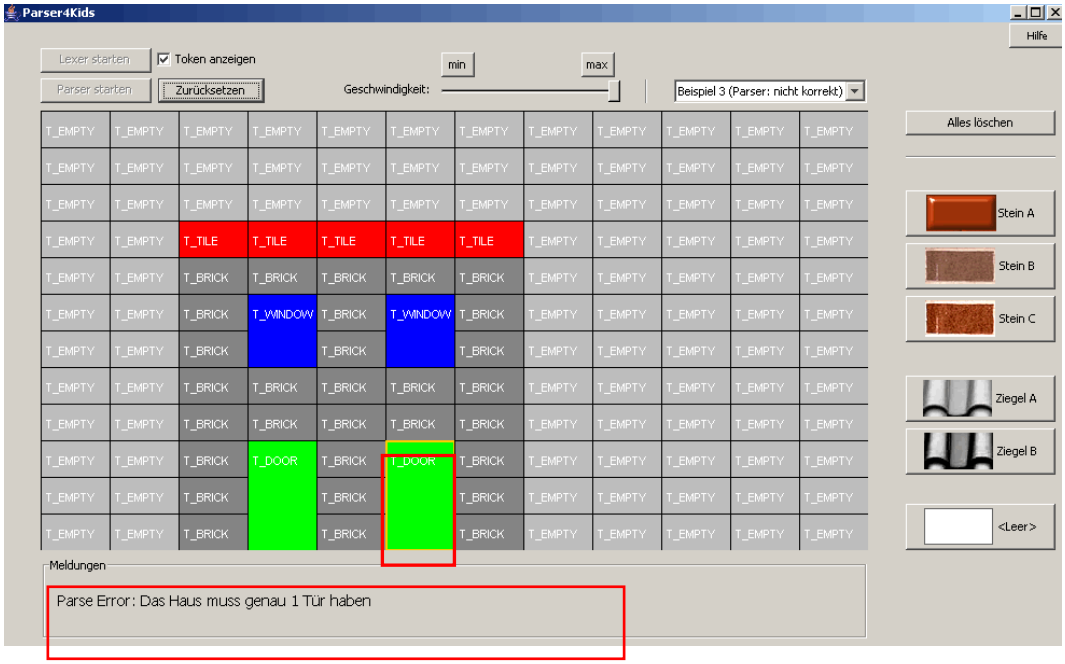
Falls der Lexer einen Fehler findet, bricht dieser an der aktuellen Position ab und gibt eine Fehlermeldung aus:



Die „Arbeitsteilung“ zwischen Lexer und Parser ist relativ willkürlich. In dieser Umsetzung erkennt der Lexer die gültigen Tokensymbole (T_WINDOW, T_TILE, etc.) und nimmt eine erste „lokale“ Prüfung vor (z.B. darf ein Dach nur 1 oder 2 Ziegel hoch sein). Der Parser hingegen ist für die „globalen“ Regeln verantwortlich (Anzahl Fenster oder Türen. Überprüfung, dass das Dach überall die gleiche Höhe besitzt).

Hier ein Beispiel, das der Lexer als korrekt beurteilt, jedoch beim Parsingvorgang zu einem Fehler führt.





7. Fazit

Wie sinnvoll diese Visualisierung ist, kann nur im konkreten Einsatz im Unterricht geprüft werden. Aus unserer Sicht wurden jedoch mit Parser4Kids sicher die folgenden Ziele erreicht:

- *Einfachheit und Nachvollziehbarkeit*. Fraglich ist jedoch, ob man anstelle der englischen Tokensymbole (T_BRICK, T_EMPTY, etc.) deutsche Wörter verwenden sollte. Im Sinne eines einfacheren Transfers in die Praxis (der Compiler spricht auch von Token und die Fehlermeldungen enthalten oft Bezeichner wie T_SYMOBOL) ist dies aus unserer Sicht jedoch kein Nachteil.

- *Realitätsbezug*. Der Vorgang und die Visualisierung sind relativ nahe an einem echten Parser/Lexer und operieren dennoch auf einem Objekt aus dem Alltag.

Eine Schwierigkeit könnte darin bestehen, dass den Schülerinnen und Schülern der Transfer „Haus ←→ Programm“ schwer fällt: Die Reihenfolge für den Überprüfungsvorgang bleibt völlig willkürlich (das Haus ist eigentlich 2-dimensional). Ein Programm wird hingegen immer zeilenweise überprüft.

Falls die Übertragung des Modells in den Programmieralltag nicht gelingt, könnte man eine ähnliche Visualisierung z.B. mit einem Taschenrechner als Modell prüfen, da hier der Transfer wesentlich einfacher wäre.

8. Begriffe

Begriff	Erklärung
Ableitungsbaum	Graphische Darstellung von Ersetzungsschritten. Sind für Grammatiken vom Typ-2 und Typ-3 definiert (da auf der linken Seite der Regeln immer nur ein nichtterminales Zeichen auftritt. Dieses Zeichen ist die Wurzel des [partiellen] Ableitungsbaums)
AST (Abstract Syntax Tree)	Entspricht einem Ableitungsbaum, wobei Kanten und Knoten, die keine zusätzliche Semantik beitragen, weggelassen werden (Beispiel Multiplikation: (x*y)*z: das Klammerpaar wird nicht in den AST übernommen, da diese semantisch nicht nötig sind). Vgl: http://en.wikipedia.org/wiki/Abstract_syntax_tree
BNF	Backus Naur Form
EBNF	Extended BNF
Grammatik	4-Tupel G = (N, T, P, S). N: Nichtterminale Symbole (syntaktische Kategorien, meist rekursiv definiert) T: Terminale Symbole (= Token aus der lexikalischen Analyse) P: Produktionen (Ableitungsschritte) S: Startsymbol
Parsingtabelle	Mit Hilfe der Parsingtabelle (leere Einträge = Error) kann zu jedem Input (Folge von Terminalsymbolen) ein Ableitungsbaum konstruiert werden. Beispiele: ▪ YACC erstellt eine Parsingtabelle aus einer LR(k) Grammatik ▪ Coco/R (vgl. [Coco/R 2006])
Token	Auch Terminalsymbol genannt. Eine Folge von Zeichen, die bedeutungsmässig zusammengehören. Token sind das Resultat der lexikalischen Analyse (durch Lexer/Scanner).

9. Literaturverzeichnis und Internetadressen

[ANTLR 2006] http://www.antlr.org/ (Zugriff am 3. März 2006)

[Coco/R 2006] http://www.ssw.uni-linz.ac.at/Coco/ (Zugriff am 3. März 2006)

[Duden 2005] Duden Informatik, 3. Auflage

[Güting 1999] R. H. Güting, M. Erwig. Übersetzerbau: Techniken, Werkzeuge, Anwendungen. Springer Verlag, Berlin.

[JavaCC 2006] http://javacc.dev.java.net/ (Zugriff am 3. März 2006)

[Lex 2006] The Lex & Yacc Page. http://dinosaur.compilertools.net/ (Zugriff am 3. März 2006).