* **SableCC Lexer und Parser:**
* Parser beherrscht nicht die Präzedenzen, die in der FDR-Dokumentation beschrieben werden. Beispiel waren die Lambda-Terms, die auf die gleiche Ebene müssen, wie z.B. der Präfixoperator, da es sonst entweder zwangsweise zu Klammerungen kommen muss (korrekte Präzedenz) oder die Präzedenz verletzt ist, also z.B. Parameter eines Lambda-Terms nur in eingeschränkten Bereichen erkannt werden.
* Mapping wird nur teilweise unterstützt. Es fehlen manche Funktionen ab FDR Version 3.0
* Nach Korrektur einiger Fehler in den Beispieldateien (waren auch z.T. für cspmf falsch), werden diese in der Zahl 124 Stück erfolgreich geparst.
* **CSPMparser Hauptdatei:**
  + Enthält Hilfsmethoden zum Löschen von Kommentaren, zum Suchen und einfügen von Dateien für den Fall, dass in der zu parsenden CSP-Datei der include-Befehl auftaucht und zum Konvertieren von Files zu Strings.
  + Diese Klasse führt alle Anderen schrittweise nacheinander aus (wie Lexer, Parser usw.).
* **TriangleBruteForce:** 
  + Umwandlung aller Sequenzklammern in « und », Umwandeln aller größer- und kleiner-Zeichen in £ bzw. € durch geschicktes Substituieren und vorsortieren.
  + In seltenen Fällen wird mit dem Bruteforce-Algorithmus solange nach parsbaren Klammerkombinationen gesucht, bis ein korrektes Beispiel gefunden wird. Dies kann im schlimmsten Fall dazu führen, dass unlösbare Berechnungen versucht werden, da bei nicht-erfolgreichem Parsen eine weitere Zeile in den Algorithmus zusätzlich geworfen wird. Die Komplexität ist x^n, wobei n die Anzahl der restlichen Klammern ist, die im Dokument vorkommen. Ein weiteres Problem, das auftauchen könnte ist, dass es mehrere parsbare Klammerungen geben könnte. Dieses Problem sollte eigentlich der Typechecker lösen, da dieser garantiert, dass immer nur DAS EINE korrekte Klammernpaar gefunden wird. Es können zwar mehrere Klammerungen durch den Parser möglich sein, aber nur eine Möglichkeit ist auch typenkorrekt. Die hier beschriebenen Probleme tauchen in keiner der Beispieldateien auf. Nur ein einziges Mal muss der Algorithmus zum Einsatz kommen.
* **Occurrence Check:** 
  + Überprüft die Gebundenheit aller Identifier, auch in Abhängigkeit der Schachtelungstiefe und Lokalität von Let-Within-Blöcken.
  + Enthält auch die Renaming-Analyse. Orientiert sich stark am cspmf-Parser. Fehler sind mir bisher keine bekannt.
* **StatementPatternCheck:** 
  + Überprüft, ob im linken Teil eines Generator-Statements ein pattern steht. Beispiel: {1,2,3|x<-{4,5,6}}. Hier wird also geprüft ob x ein pattern ist. Dies ist über die Grammatik nicht zu lösen, da an der gleichen Stelle auch ein predicate Statement stehen könnte, das auf die Expressions-Grammatik abbildet, die jedoch eine Obermenge der Pattern-Grammatik ist. Dies hat reduce/reduce-conflicts zufolge.
  + Gelöst wird das Problem durch einen AST-Visitor, der für den Fall des Aufrufens eines ID-Knotens schaut, ob aktuell auch ein Generator-Statement-Knoten aufgerufen wird. Wenn ja, dann werden alle Regeln verboten, die patterns nicht zulassen. Ebenso wird geschaut ob das explizite Set mehr als ein Element aufnimmt. Durch das zurechtbiegen der Obermengengrammatik entstehet gerade die Untermenge, die das Erzeugen von patterns gewährleistet.
* **TreeLogicChecker:** Anbindung des Parsers für CTL/LTL-Formeln (Kaum Eigenleistung).
* **Typechecker:** 
  + Sehr frühe Phase, da funktionaler Teil sehr kompliziert und zeitaufwändig.
  + Alles was keine Parameter hat und irgendwann zugewiesen wird, kann ohne Weiteres gecheckt werden (noch ausstehend).
* **Prolog-Codegenerierung:** 
  + Fehlt.
  + Nötige Fakten sind unklar und erfordern eine Erklärung.
  + Falls sämtliche Zeilenangaben erforderlich sind, so wird dies zu einer Überarbeitung aller oben genannten Klassen führen müssen.
* **Schriftliche Ausarbeitung:** 
  + Bereits strukturiert.
  + Planung des Schreibbeginns: 20-25. Juli.