Karlsruher Institut für Technologie
Lehrstuhl für Programmierparadigmen
Sprachtechnologie und Compiler WS 2010/2011
Dozent: Prof. Dr.-Ing. G. Snelting
Übungsleiter: Matthias Braun `matthias.braun@kit.edu`

Übungsblatt 6 | Ausgabe: 23.5.2012 | Besprechung: 4.6.2012

Aufgabe 1: *Earley Parser Implementierung (Zusatzaufgabe)*

Unter `http://pp.info.uni-karlsruhe.de/lehre/SS2012/compiler/uebung/earley_framework.zip` finden sie den Beginn einer Implementierung des Earley-Parsers in python. Aufgabe ist es die fehlenden Teile einzufügen.

Tips:

- Alle benötigten Datenstrukturen sind bereits vorhanden.
- Im Archiv ist auch Code zum Aufbau eines LR(0)-Automaten enthalten. Der Code ist als Orientierung und Beispiel zur Benutzung der Datenstrukturen gedacht, er kann nicht für den Earley-Parser benutzt werden!
- Am Ende von earley.py befindet sich bereits eine Beispielgrammatik mit Ausgabe
- Es fehlen genau die Funktionen _predict, _scan und _complete
- Wie in vielen Programmiersprachen kann man auch in python eine Menge(set) nicht verändern während man über sie iteriert. Bei der Implementierung von Fixpunktiterationen vergisst man dies leicht (python meldet dann einen Fehler).

Aufgabe 2: *Earley Parser*

Gegeben folgende Grammatik:

$$\begin{array}{rcl} T & \rightarrow & T \texttt{ <- } T \\ & | & T \texttt{ -> } T \\ & | & \texttt{\&} \; T \\ & | & \textbf{id} \end{array}$$

Konstruieren Sie mit Hilfe des Earley Algorithmus alle möglichen Parsebäume für folgende Sätze:

- **id -> id <- id**
- **& id <- id**

Aufgabe 3: *Attributierte Grammatiken*

Gegeben folgender Auszug aus der Grammatik einer Java-ähnlichen Sprache:

$$
\begin{aligned}
\mathit{CompilationUnit} &\rightarrow \mathit{Class}\ \mathit{CompilationUnit} \mid \varepsilon \\
\mathit{Class} &\rightarrow \textbf{class id \{}\ \mathit{ClassMembers}\ \textbf{\}} \\
\mathit{ClassMembers} &\rightarrow \mathit{ClassMember}\ \mathit{ClassMembers} \mid \varepsilon \\
\mathit{ClassMember} &\rightarrow \mathit{Field} \mid \mathit{Method} \\
\mathit{Field} &\rightarrow \mathit{Type}\ \textbf{id ;} \\
\mathit{Method} &\rightarrow \mathit{Type}\ \textbf{id}\ \mathit{CompoundStatement} \\
\mathit{CompoundStatement} &\rightarrow \textbf{\{}\ \mathit{StatementList}\ \textbf{\}} \\
\mathit{StatementList} &\rightarrow \mathit{Statement}\ \mathit{StatementList} \mid \varepsilon \\
\mathit{Statement} &\rightarrow \mathit{CompoundStatement} \mid \textbf{;} \\
\mathit{Type} &\rightarrow \textbf{void}
\end{aligned}
$$

Weiter seien Funktionen mit folgenden Signaturen zum Aufbau eines AST gegeben:

$$
\begin{aligned}
\text{newCompilationUnit} &: \mathcal{P}(\mathit{Class}) \rightarrow \mathit{CompilationUnit} \\
\text{newClass} &: \mathit{Symbol} \times \mathcal{P}(\mathit{ClassMember}) \rightarrow \mathit{Class} \\
\text{newField} &: \mathit{Symbol} \times \mathit{Type} \rightarrow \mathit{ClassMember} \\
\text{newMethod} &: \mathit{Symbol} \times \mathit{Type} \times \mathit{Statement} \rightarrow \mathit{ClassMember} \\
\text{newCompoundStatement} &: \mathcal{P}(\mathit{Statement}) \rightarrow \mathit{Statement} \\
\text{emptyStatement} &: \mathit{Statement} \\
\text{voidType} &: \mathit{Type}
\end{aligned}
$$

Das Terminal **id** besitzt ein vordefiniertes Attribut $\mathit{symbol} : \mathit{Symbol}$.

Stellen Sie Attributierungsregeln auf, die zu einem Programm einen passendes AST erzeugen.

Aufgabe 4: *Auswertung Boolescher Ausdrücke*

Boolesche Ausdrücke können oft mit bedingten Sprüngen ausgewertet werden. Dabei muss jeweils die Adresse spezifiziert werden, an der die Berechnung nach dem Sprung fortgesetzt wird. Gegeben sei folgende Syntax Boolescher Ausdrücke:

$$
\begin{aligned}
\mathit{conditional\_clause} &\rightarrow \textbf{if}\ \mathit{boolean\_expr}\ \textbf{then}\ \mathit{statement\_list}\ \textbf{else}\ \mathit{statement\_list}\ \textbf{end} \\
\mathit{boolean\_expr} &\rightarrow \mathit{boolean\_expr}\ \mathit{boolean\_op}\ \mathit{boolean\_expr} \\
\mathit{boolean\_expr} &\rightarrow \textbf{not}\ \mathit{boolean\_expr} \\
\mathit{boolean\_op} &\rightarrow \textbf{and} \mid \textbf{or}
\end{aligned}
$$

Die Codeerzeugung soll aus dem AST erfolgen. Dafür müssen die Knoten mit Sprungmarken versehen werden, die mit bedingten Sprüngen angesprungen werden können.

Erzeugen Sie dazu die folgenden Attribute:

- *location* gibt die Sprungmarke eines Knotens an.
- *jump_true* und *jump_false* die Sprungziele bei positiver bzw. negativer Auswertung eines Ausdrucks.

Sie dürfen die folgenden Funktionen in den Regeln benutzen:

- new_label() erzeugt eine neue Sprungmarke.
- emit_expr(expr) erzeugt Code der den Ausdruck expr auswertet. Dieser Code verändert das Flags Register des Prozessors. Ist der Ausdruck wahr, wird das *equal* flag gesetzt, ansonsten wird es gelöscht.
- emit_jump_equal(label) erzugt Code, der zum angegebenen Label springt falls das *equal* Flag gesetzt ist.
- emit_jump_notequal(label) erzeugt Code, der zum angegebenen Label springt falls das *equal* Flag nicht gesetzt ist.
- emit_statement_list(list) erzeugt Code für eine Liste von Anwendungen (einen *statement_list* Knoten).

Geben Sie eine attributierte Grammatik an, die Code für boolesche Ausdrücke erzeugt.