Parser Dokumentation

# Der Parser

Bei der "Systemnahes Programmieren" Veranstaltung ist ein Compiler zu kreieren. Diese erfolgt, indem zwei Hauptkomponenten implementiert werden: der Scanner und der Parser. Jedes Modul ist ein wichtiges Bestandteil eines Compilers.

**Theorie**

Der Parser beschäftigt sich mit dem Parsen(Zerteilen) des Quelltexts und Umwandeln des Quelltexts in einen Assembler-Code. Der Inhalt wird zerlegt und ein Strukturbaum(Parse-Tree) erstellt. Der Parser prüft den Inhalt des Quelltextes und gibt Fehlermeldungen aus, falls er ein Syntaxfehler findet. Danach macht der Parser eine semantische Analyse und generiert ein Assembler-Code.

**Vorgehensweise**

Der Parser wird anhand eines Dateipfads initialisiert. Es wird auch eine Symboltabelle angelegt. Der Pfad und die Symboltabelle werden an den Scanner weitergegeben. Zusätzlich werden noch ein Visitor und Wurzelknoten für ein Parsebaum initialisiert.

Sobald die parse() Methode aufgerufen wird, werden es die Methoden nextToken() und parseNode(root) von Visitor ausgeführt. Die nextToken() Methode parst die Tokens gemäß folgender Grammatik:

*PROG ::= DECLS STATEMENTS*

*DECLS ::= DECL* **;** *DECLS* | e

*DECL ::=* **int** *ARRAY* ***identifier***

*ARRAY ::=* **[ *integer* ]** | e

*STATEMENTS ::= STATEMENT* **;** *STATEMENTS* | e

*STATEMENT ::=* ***identifier*** *INDEX* **:=** *EXP* | **write(** *EXP* **)** | **read ( *identifier*** *INDEX***)** | **{***STATEMENTS***}** |

**if (** *EXP* **)** *STATEMENT* **else** *STATEMENT* |

**while (** *EXP* **)** *STATEMENT*

*EXP ::= EXP2 OP\_EXP*

*EXP2 ::=* **(** *EXP* **)** | ***identifier*** *INDEX* | ***integer*** | **-** *EXP2* | **!** *EXP2*

*INDEX ::=* **[** *EXP* **]** | e

*OP\_EXP ::= OP EXP* | e

*OP ::=* **+** | **-** | **\*** | **:** | **<** | **>** | **=** | **=:=** | **&&**

Falls der Token gültig ist und gehört zu dieser Grammatik wird dieses zu dem Strukturbaum hinzugefügt und ein neues Aufruf der nextToken() Methode erfolgt. Es wird ein Prinzip des rekursiven Abstiegs benutzt um die ganze Datei zu parsen.

Mit dem gleichen Prinzip werden dann eine Typprüfung jedes Knotens des Baums und Erzeugung zu jedem Knoten eines Code-Segments.

**Testen**

Der erzeugte Code wird in ein „xxx.code“ Datei geschrieben. Dieses kann mit einem gegebenen Interpreter ausgeführt und getestet werden.

**Probleme (allgemein)**

Da man kann keine Bibliotheken nutzen dürfte, musste man die Sets und List Klassen selber implementieren. Die List Klasse war ohne vorheriges Wissen über Templates Konzept schwer zu implementieren.

# Typecheck/Codegenerierung

Die Typüberprüfung und Codegenenrierungs-Methoden sehen sind ähnlich im Aufbau, wobei der Unterschied ist, dass man beim Typprüfung Typ des Knotens prüft und den Fehlermeldung ausgibt, falls ein semantischer Fehler gefunden wurde. Bei der Codegenerierung wird ein Code generiert, der die Regel entspricht.

Man fängt mit dem Wurzelknoten des Strukturbaums und prüft rekursiv jeden Knoten.