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. Der Parser fordert die Tokens vom Scanner an. Er prüft, ob die Reihenfolge der Tokens sinnvoll ist, und gibt Fehlermeldungen aus, falls er ein Syntaxfehler findet. Um eine semantische Analyse zu durchführen, die zur Typprüfung benutzt wird, baut der Parser ein Strukturbaum (Parse Tree). Der Baum wird auch benötigt um am Ende den Assembler-Code zu generieren.

**Vorgehensweise**

Der Parser wird anhand eines Dateipfads initialisiert. Es wird auch eine neue Symboltabelle angelegt. Der Parser initialisiert den Scanner, in dem er der Dateipfad und die Symboltabelle übergibt. Zusätzlich werden noch ein *parseVisitor* und root(Wurzelknoten) für ein Parsebaum initialisiert. Es wird auch ein outputstream für die Assemlber-Code Ausgabe angelegt. Bei der Initialisierung werden in die Symboltabelle die ersten Identifier hinzugefügt: *while*, *if*, *else*, *int*, *read*, *write*. Wobei jedes wird zwei Mal hinzugefügt. Einmal Großgeschrieben und einmal Klein.

Der Parser ruft die Methode *parseNode()* vom *ParseVisitor* und prüft die Reihenfolge der Tokens und baut den Strukturbaum auf. Dann ruft er die *checkType()* um den Code auf semantische Fehler zu überprüfen. Zum Schluss ruft er *makeCode()* auf und generiert den Code in den Outputstream.

**Testen**

Der erzeugte Code wird in ein „*output-xxx.code*“ Datei geschrieben. Dieses kann mit einem gegebenen Java 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.

# Parsen

Parsen der Datei erfolgt mittels der *parse()* Methode. Diese ruft eine *parseVisitor()* Methode vom Visitor. Die 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. Der Parser erzeugt den Baum von oben nach unten. Er fängt mit dem Wurzelknoten(root) und endet mit Terminalsymbolen (z.B. „+“).

Das folgende Stück Code zeigt was im Fall von *DECLS* passiert:

Zuerst prüft der Parser den Typ des Knotens. Kreiert eine neue *DECL* Knote und ruft *parse()* auf. Dieser Knoten wird zu dem Parsebaum hinzugefügt. Nach der Regel kommt danach ein Semikolon (;). Dieses wird als ein Blatt zu dem Baum hinzugefügt, sonst gibt der Parser eine Fehlermeldung aus. Es wird *DECLS* Knote kreiert, die wieder eine *parse()* Methode aufruft, und wird zu dem Baum hinzugefügt.

Unser Parser ist ein LL-Parser, d.h. er abarbeitet die Angabe von links nach rechts. Was uns möglich macht eine Technik des rekursiven Abstiegs zu benutzen, um die Grammatik entsprechend abarbeiten zu können. Also die Eingabe wird von oben nach unten bearbeitet, wobei die Funktionen rekursiv aufgerufen werden (z.B. im Code oben *declChild->parse(this);*).

Mit dem erfolgreich erzeugten Strukturbaum kann der Parser eine Typprüfung und Codegenerierung durchführen.

# Typüberprüfung

Man nimmt den Strukturbaum und zu jedem Knoten ermittelt den entsprechenden Typ und überprüft, ob die Typen der Unterbäume zusammenpassen. Typ-Informationen werden im Knoten und für Identifier in der Symboltabelle gespeichert. Es sind folgende Typ-Informationen abzuspeichern:

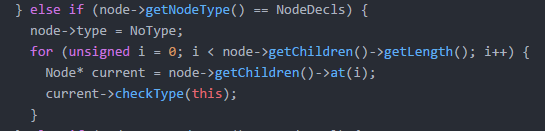
intType , intArrayType, arrayType,

noType, errorType,

opPlus, opMinus, opMult, opDiv, opLess,

opGreater, opEqual, opUnEqual, opAnd

Im Rot sind die Identifier. Die Technik die man beim Typprüfung benutzt, ist ähnlich wie beim parsen. Man sieht es am folgenden Beispiel:



Man überprüft den übergegebenen Knoten auf Typ. Da es hier um keinen Typ handelt, wird der Typ auf *NoType* gesetzt. Dann wird absteigend rekursiv alle weitere Knoten überprüft. Falls man ein semantischer Fehler findet, gibt es eine Fehlermeldung.

# Codegenerierung

Um den gegebenen Quellcode zu evaluieren muss den zugehörigen Assembler-Code generiert werden. Die Kommandos für den Assembler-Code haben folgende Struktur:

**label cmd arg**

* **label**: Ein optionales identifier mit führenden „#“
* **cmd**: Ein Schlüsselwort (z.B. *ADD* für Summe)
* **arg**: Ein optionales Integer, Variable (Identifier mit führenden „$“) oder Label

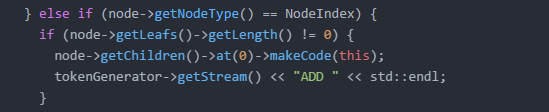
Mit Ausnahme von Kommando zur Speicherreservierung: *DS variable Integer*. Dieser Code soll auf eine einfache Stack-Maschine arbeiten.

Für die Codegenerierung benutzt man verschiedene Regel. Mit Hilfe von Strukturbaums und mittels rekursiven Abstiegs wenden man diese Regeln an und erzeugt den Code. Für den Knoten INDEX sieht der Regel so aus:

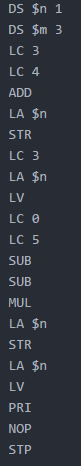
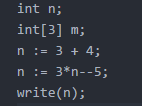
makeCode (INDEX ::= **[** EXP **]**){makeCode(EXP); code << “ ADD ”;}

makeCode (INDEX ::= e){}

Man ruft rekursiv *makeCode(EXP)* und fügt die „*ADD*“ am Ende, falls der Index Knote nicht eine leere Knote war. Implementierung dieser Regel:



Da Index, wenn er nicht leer ist, mit einer „[„Klammer anfangen muss, prüft die Methode *getLeafs()* die Länge des Blattknoten. Dann nimmt man den ersten nicht Blatt Knoten und ruft *makeCode(this)* auf.

Zum Schluss bekommt man aus dem Quellcode den folgenden Assembler-Code.