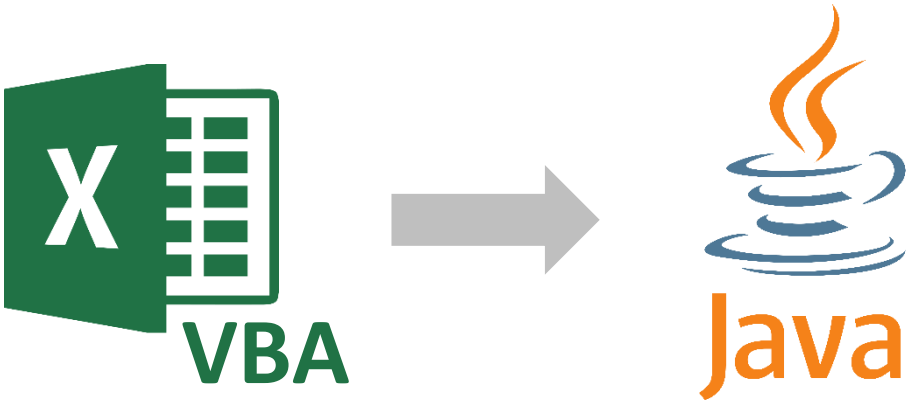
Excel Visual Basic for Application zu Java Compiler



Paul Wenzel
wenzel.paul@de.ibm.com

08-03-2015

1 Inhaltsverzeichnis

1 Aufgabe

1.1 Problem

Das MDS IVK Projekt nutzt einen DDL (Data Definition Language) File Generator zum Generieren ihrer Datenbankstrukturdateien. Dieser Generator liest Daten aus einem Workbook mit 36 Sheets ein, um die Statements für die Erzeugung der verschiedenen Abhängigkeiten und Zusammenhänge zwischen den Datenbankobjekten zu generieren. Die Logik, die dies tut ist aktuell in einem Excel Makro geschrieben. Dieses Makro hat 129.626 Zeilen und ist auf über 110 Module verteilt[1].

Das Kernproblem ist, dass der Quellcode dieses Makros ständig angepasst werden muss und des Öfteren mehrere Mitarbeiter des Teams gleichzeitig am Code arbeiten wollen. Da in Excel der Code von den einzelnen Modulen direkt in die Excel-Datei eingebettet ist, ist dieser von außen nicht ohne weiteres erreichbar und in einem Binärformat abgespeichert. Dies macht es sehr umständlich[2] eine Versionsverwaltung wie Rational Team Concert (RTC) zu verwenden. Das wiederum bedeutet, dass jeweils nur ein Mitarbeiter am Generator arbeiten kann und nur sehr umständlich eine Änderungshistorie für den DDL Generator erzeugt werden kann.

Ein weiter Negativpunkt ist die Laufzeit des Generators, die deutlich über drei Minuten liegt. Die Laufzeit stellt zwar während der eigentlichen Anwendung des Excel-Makros kein Problem dar, jedoch hemmt es den Entwicklungsfluss erheblich, da jeder Testlauf über drei Minuten dauert.

1.2 Lösung

Eine Lösung der genannten Probleme ist die Umwandlung des Excel-VBA-Codes in Java-Code, sodass der Generator in Zukunft in Java weiterentwickelt wird. Zusätzlich vereinfacht dies die Weiterentwicklung des Generators, da für Java weitaus mächtigere IDEs zur Verfügung stehen, als der Makroeditor in Excel.

1.3 Umsetzung

Da das Excel Makro extrem umfangreich ist, entfällt ein manuelles überführen vom Excel-VBA-Code zu Java-Code. Stattdessen muss ein Compiler geschrieben, der diese Aufgabe übernimmt.

Dieser Compiler kann recht einfach gehalten werden. Für den konkreten Anwendungsfall genügt es, wenn der Compiler aus Scanner, Parser und Zielcode-Generator besteht. Somit entfällt die semantische Analyse, das Überführen der Eingabe in eine unabhängige Darstellung und die Code-Optimierung.

---

[1] Angaben übernommen aus „rtc/tools/com.ibm.tool.ddlgenerator/VBCompilerDoku.doc“

[2] Die Module werden derzeitig mit der Hilfe von einem Excel-Makro in .bas-Dateien Exportiert. Diese Dateien liegen dann im Textformat vor und können von einer Versionsverwaltung getrackt werden. Auf diese Weise ist sichergestellt, dass immer zu einem früheren Entwicklungsstand zurückgekehrt werden kann. Allerdings gestaltet sich dies unnötig kompliziert. Um zu einer früheren Version zurückzukehren, muss der Code aus den exportierten .bas-Dateien wieder in die Module innerhalb der Excel-Datei manuell eingearbeitet werden.

2 Grundlagen

2.1 Compilerbau

Ein Compiler besteht grob aus drei Teilen: Der Analyse der Eingabe, der Darstellung von der Eingabe auf eine sprachenunabhängige Weise und zuletzt aus der Überführung der unabhängigen Darstellung in die Ziel Sprache.

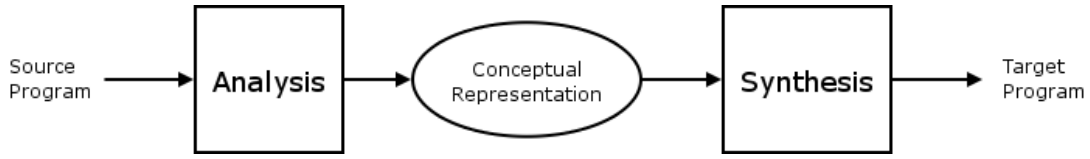


Abbildung 1:Bestandteile eines Compilers grob aufgelistet[3]

2.1.1 Analyse

Die Analyse lässt sich wiederum in drei Teile unterteilen. Der lexikalischen Analyse, der syntaktischen Analyse und der semantischen Analyse.

Die lexikalische Analyse wird von einem Scanner durchgeführt, welcher die Eingabe in einzelne Tokens zerlegt. Ein Token besteht dabei aus einem oder mehreren Zeichen. Auf diese Weise werden die Bestandteile des Textes grob herausgearbeitet.

Nachdem der Text lexikalisch analysiert wurde, führt ein Parser die Syntaktische Analyse durch. Bei dieser werden gegebene Syntax-Regeln[4] auf die Ausgabe vom Scanner angewendet, um einen Syntaxbaum zu erstellen. Dabei wird gleichzeitig überprüft, ob die Eingabe syntaktisch korrekt ist.
Ein Parser lässt sich gut mit einem deterministischen, endlichen Automaten umsetzen. Die Syntax-Regeln sind somit nichts anderes als eine Grammatik für einen Automaten.
In einem Syntaxbaum sind die Tokens Blätter und die Syntax-Regeln Knoten.

[3] http://cs.lmu.edu/~ray/notes/compilerarchitecture/

[4] Regeln nach denen die Tokens, aus welchen die Eingabe besteht, strukturiert sein dürfen.

Grammatik:

1 $S \rightarrow S\ ;\ S$
2 $S \rightarrow \text{id} := E$
3 $S \rightarrow \text{print}\ (\ L\ )$
4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E + E$
7 $E \rightarrow (S\ ,\ E\ )$
8 $L \rightarrow E$
9 $L \rightarrow L\ ,\ E$

Eingabe:

id:=num; id := id + (id:=num+num, id)

Syntax-Tree:

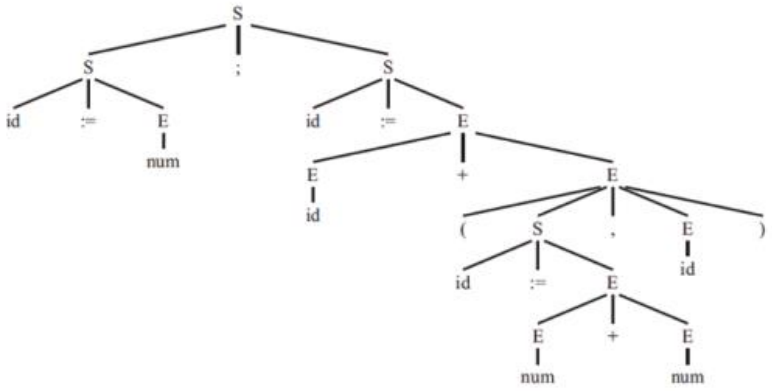


Abbildung 2: Beispiel für eine Eingabe, die in einem Syntaxbaum strukturiert dargestellt wird (Grafik wird noch überarbeitet)

Um die Analyse der Eingabe abzuschließen, muss diese noch semantisch analysiert werden. Das bedeutet, dass ein semantischer Analysierer Zusammenhänge innerhalb vom Syntaxbaum herausarbeitet und somit den Sinn, bzw. die Bedeutung der Eingabe darstellt.

2.1.2 Sprachunabhängige Darstellung

Wenn ein Compiler die Eingabe nicht nur in eine, sondern in viele verschiedene Sprachen übersetzen soll, lohnt es sich die Eingabe nicht direkt in die Ziel-Sprache zu übersetzen. Sinnvoller ist es, die Eingabe zunächst in eine Eigenständige Sprache zu übersetzen (IR-Code[5]). Dadurch ist der Compiler viel flexibler und effizienter, weil die Ein- und Ausgabesprachen variieren können.

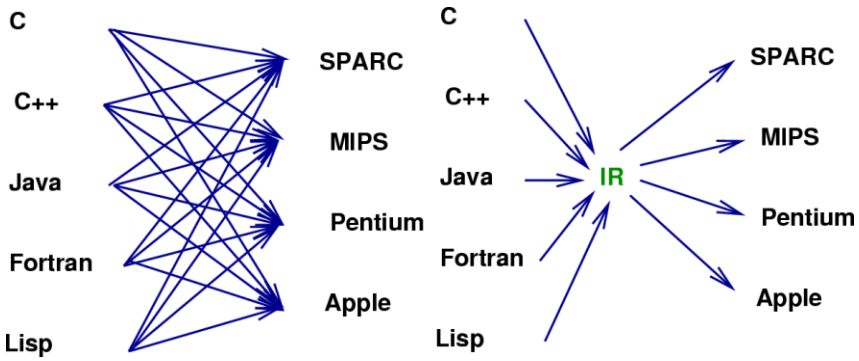


Abbildung 3: die Vorteile, durch die Sprachunabhängige Darstellung[6]

[5] Intermediate Representation

[6] http://www.csd.uwo.ca/~moreno/CS447/Lectures/Introduction.html/node6.html

Damit die Flexibilität erreicht werden kann, ist es nötig einen IR-Code-Generator zu haben. Sobald dieser den Code generiert hat, optimiert ein IR-Optimierer diesen. Die Optimierung erfolgt Zielsprachenunabhängig.

2.1.3 Überführung in die Zielsprache

Ein Generator übersetzt im vorletzten Schritt den IR-Code in die gewünschte Zielsprache. Abgerundet wird diese Übersetzung wiederum durch einen Code-Optimierer.

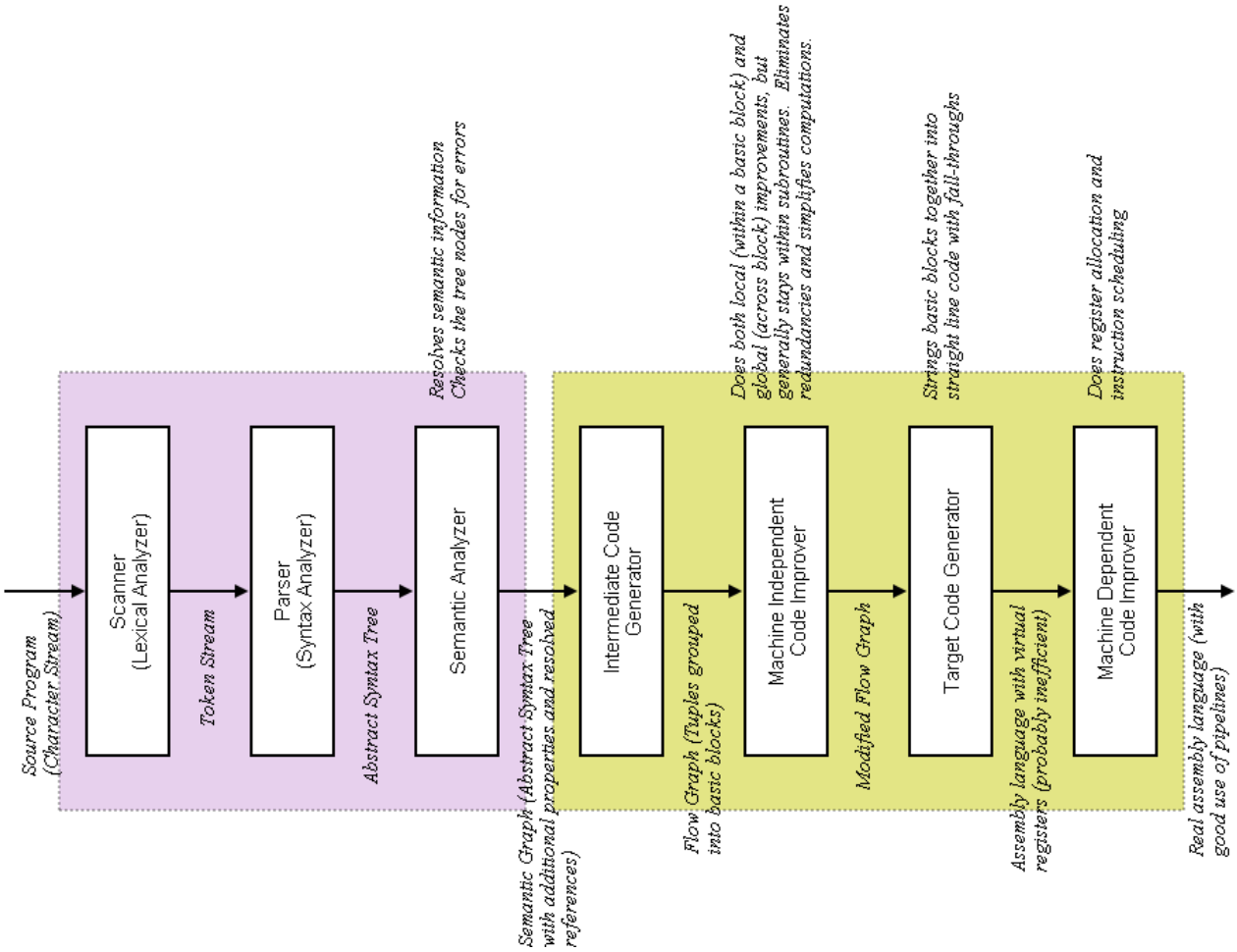


Abbildung 4: Bestandteile eines Compilers[7] (Grafik wird noch überarbeitet)

[7] http://cs.lmu.edu/~ray/notes/compilerarchitecture/

2.2 SableCC[8]

SableCC ist ein Open Source Compiler Generator. Das Programm selber ist in Java geschrieben und erzeugt Parser und Lexer ebenfalls in Java. Dabei generiert SableCC einen LALR(1) -Parser[9]. Ein LALR(1)-Parser ist ein Lokkahead-LR-Parser[10] mit einem Lookahead[11] von eins. Die Grammatik, die SableCC entgegen nimmt, muss in der Erweiterten Backus-Natur-Form (EBNF)[12] vorliegen.

Der Vorteil an SableCC im Vergleich zu anderen Java-Parser-Generatoren, wie JavaCC, ist vor allem, dass SableCC nicht nur Parser und Lexer erzeugt, sondern auch gleich einen Syntaxbaum generiert.

Diese Dokumentation bezieht sich auf SableCC in der Version 3.3, da sich SableCC 4 noch im Beta Stadium befindet. Es sei an dieser Stelle darauf hingewiesen, dass die Grammatiken von Version 3.3 nicht mit den der Version 4 kompatibel sind.

2.2.1 SableCC Grammatikspezifikation

Eine SableCC Grammatikspezifikation besteht unteranderem aus:

0. Package
1. Helpers
2. (States)
3. Tokens
4. Ignored Tokens
5. Productions
6. (Abstract Syntax Tree)

Helper, States und Tokens dienen zur Deklaration der lexikalen Analyse. Ignored Tokens, Productions und Abstract Syntax Tree definieren den Parser.
Zusätzlich gibt es eine Package-Deklaration, die dazu dient, SableCC mitzuteilen, wo es die generierten Dateien ablegen soll.

Im Folgenden wird nicht weiter auf die State- und Abstract Syntax Tree-Deklaration eingegangen, weil sie keine Verwendung in diesem Projekt finden.

2.2.1.1 Package

Über die Package-Definition wird der gewünschte Pfad im src-Ordner angegeben, in dem die von SableCC generierten Java-Dateien gespeichert werden sollen. SableCC erzeugt diesen Pfad, falls er noch nicht vorhanden ist. Des Weiteren setzt es das Package-Statement in die Java-Dateien. Deshalb muss der Dateipfad zum gewünschten Ordner, vom src-Ordner ausgehend, angegeben werden.

---

[8] dieses Kapitel beinhaltet teilweise Passagen aus:
Aho, Alfred V. / Lam, Monica S. / Sethi, Ravi / Ullman, Jeffrey D.: Compiler – Prinzipien, Techniken und Werkzeuge. München [2]2003, S.1198 – 1220.

[9] http://en.wikipedia.org/wiki/SableCC

[10] http://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf

[11] http://de.wikipedia.org/wiki/Lookahead

[12] http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form

***Package** com.folder.sablecc*

Quelltext 1: wenn die Grammatik-Datei im src-Ordner liegt, speichert SableCC die generierten Java-Dateien unter: [...]/src/com/folder/sablecc/<SableCC Dateien>

SableCC generiert die komplette Ordnerstruktur mit den Java-Dateien in den selber Ordner, in dem auch die SableCC-Grammatik-Datei liegt. Damit die generierten Dateien nicht nach jedem Generiervorgang manuell an die richtige Stelle im Projekt eingefügt werden müssen, muss die SableCC-Grammatik-Datei im src-Ordner vom Projekt gespeichert werden.

Um die generierten Dateien etwas zu sortieren, teilt SableCC die Java-Dateien auf vier verschiedene Ordner auf: „analysis“, „lexer“, „parser“ und „node“. Die Ordner „lexer“ und „parser“ beinhalten jeweils die Klassen, die für das Lexing bzw. Parsing benötigt werden. Der Ordner „node“ beinhaltet alle Objekte, aus welchen hinterher der Syntaxbaum aufgebaut wird. Im Ordner „analysis“ befinden sich Klassen, die zur Durchführung einer Analyse auf den generierten Syntaxbaum benutzt werden können. Besonders der „DepthFirstAdapter“ ist eine große Hilfe, da er eine Tiefensuche auf den Syntaxbaum implementiert. Indem eine eigene Klasse vom "DepthFirstAdapter" erbt, wird eine sehr gute Schnittstelle erlangt, um einen Syntaxbaum zu durchschreiten.

2.2.1.2 Helpers und Tokens

Die beiden Teile Helpers und Tokens dienen zur Deklaration der lexikalischen Analyse.

Unter dem Schlüsselwort „Tokens“ werden alle Terminalsymbole der Grammatik durch reguläre Ausdrücke beschrieben.

Operator	**Bedeutung**	**Beispiel**
|	Oder	‚a‘ | ‚b‘ → „a“; „b“
*	Kleene‘sche Hülle	‚a‘* → „“; „a“; „aa“; „aaa“; …
+	Positive Hülle	‚a‘+,b‘ → „ab“; „aab“; „aaab“; …
?	Optionaler Ausdruck	‚a‘?,b‘ → „b“; „ab“
()	Klammerung	(‚abc‘ | ‚xyz‘) → „abc“; „xyz“
(Leerzeichen)	Konkatenierung	‚a‘ ‚b‘ → „ab“
‚<Text>‘	Zeichenkette	‚a b*+ C‘ | d → „a b*+ C“; „d“
[<Start>..<Ende>]	Wertebereich	[‚0‘ .. ‚2‘] → „0“; „1“; „2“

Tabelle 1:Operatoren, um reguläre Ausdrücke zu formulieren

Tokens

blank = (' ');*
zero = ‚0‘
digit = ['0' .. '9'];

Quelltext 2: Tokendefinitionen

Bei der Tokendefinition kann es schnell zu Mehrdeutigkeiten kommen. Im Quelltext 2: Tokendefinitionen ist dies zum Beispiel bei der Eingabe „0“ der Fall. Die Eingabe kann entweder als das Token „zero“ oder aber als das Token „digit“ erfasst werden. Um solch eine Mehrdeutigkeit aufzulösen, hält sich der Lexer an zwei Regeln[13]:

1. *Längste Übereinstimmung:*
   Das längste Präfix, das zu einer Token-Regel passt, wird das nächste Token.
2. *Höchste Priorität:*
   Sollte die erste Regel nicht ausreichen, um die Mehrdeutigkeit aufzulösen, wird die Token-Regel mit der höheren Priorität gewählt. In SableCC wird die Priorität durch die Reihenfolge, in welcher die Token-Regeln definiert werden, bestimmt. Die zuerst definierte Token-Regel hat die höchste Priorität.

Dementsprechend würde die Eingabe „0“ der Token-Regel „zero“ zugewiesen.

Es ist verboten, bereits definierte Tokens in weiteren Token-Definitionen zu verwenden. Genau für diesen Fall gibt es jedoch die Helpers. Helpers werden wie Tokens definiert. Im Gegensatz zu Tokens sind sie jedoch innerhalb von Produktionsregeln nicht sichtbar und werden auch nicht direkt beim Lexing beachtet.

Helpers

end_of_line = (10 | 13 | 9);

Tokens

t1 = ‚token‘ end_of_line;
t2 = ‚token 2‘ end_of_line;

Quelltext 3: Tokendefinition mit Helpers

2.2.1.3 Ignored Tokens und Productions

Die Abschnitte „Ignored Tokens“ und „Productions“ spezifizieren den Parser.

Unter „Ignored Tokens“ werden alle Tokens aufgelistet, die zwar vom Lexer, aber nicht vom Parser beachtet werden sollen. Das bedeutet, dass diese Tokens dann auch nicht in Produktionsregeln vorkommen dürfen.

Ignored Tokens

blank;

Quelltext 4: Ignored Tokens-Definition

[13] Appel, Andrew W.: modern compiler implementation in Java. Cambridge [2]2002, Abschnitt 2.2.

Der Productions-Abschnitt beschreibt die Produktionsregeln der Grammatik. Auf der Grundlage von diesen Regeln baut SableCC den Parser auf. Eine Produktionsregel wird aus einem Nichtterminalsymbol, gefolgt von einem Gleichheitszeichen und einer Aufreihung von Nichtterminal- und Terminalsymbolen gebildet. Da SableCC ein LALR(1) Parserframework erzeugt, darf die Grammatik sowohl Links-, als auch Rechtsrekursiv formuliert werden.

Operator	**Bedeutung**	**Beispiel**
{<Name>}	benennt eine Ausprägung einer Produktionsregel, um diese später bei der Analyse noch erkennen zu können	production = {a}a
[<Name>]	weist den Symbolen einer Produktionsregel einen Namen zu (Hinweis: Wenn in einer Produktionsausprägung mehrmals das gleiche Symbol vorkommt, müssen diese einen eindeutigen Namen zugewiesen bekommen!)	production = {a}[first]a [second]a
|	ermöglicht es unterschiedliche Ausprägungen einer Produktionsregel zu formulieren (Hinweis: Es Pflicht unterschiedlichen Ausprägungen einen eindeutigen Namen zugeben!)	production = {a}a | {b}b;
*	Kleene‘sche Hülle	production = a* b;
+	Positive Hülle	production = a+;
?	Optionaler Ausdruck	production = a? b;

Tabelle 2: Operatoren, für die Definition von Produktionsregeln

Productions

data_type = {integer}integer | {string}string;
var_dec = modifier id as data_type;*

Quelltext 5: Productions-Definition

2.2.1.4 Kommentare

Um die Grammatik verständlicher zu gestalten oder zu debuggen, können auch Kommentare formuliert werden.

Operator	Bedeutung	Beispiel
// <Kommentar>	Zeilenkommentar	production = {a}a // Kommentar
/* <Kommentar> */	Blockkommentar	/* Ich bin ein Kommentar. */

Quelltext 6: Operatoren, um in der Grammatik Kommentare zu setzen

2.2.2 SableCC ausführen

Sobald die Grammatik definiert ist, können Lexer und Parser generiert werden. Lexer und Parser werden dann eine Eingabe in einen Syntaxbaum darstellen.

Damit Lexer und Parser gemäß einer gegebenen Grammatik gebaut werden, muss die sablecc.jar-Datei über die Kommandozeile ausgeführt werden. Zusätzlich wird als Parameter noch der Speicherort der Grammatik angegeben. Außerdem lohnt es sich, bei komplexeren Grammatiken mehr Arbeitsspeicher freizugeben, um die Laufzeit zu verringern.

```
java -Xms128m -Xmx3524m -XX:MaxPermSize=256m -jar ../tool/sablecc.jar sample.scc
```

Quelltext 7: Kommandozeilenbefehl, um Compiler bauen zu lassen

Wenn einen Compiler zum wiederholten Mal generiert wird, sollte zuvor die bestehenden Dateien gelöscht werden, um Konflikte zu verhindern. Nachdem die Dateien generiert wurden, muss die Projektstruktur in der IDE aktualisiert werden, um sicherzustellen, dass die IDE die neu generierten Dateien kompiliert und verwendet.

In der Praxis lohnt es sich außerdem die Laufzeit vom Generiervorgang im Auge zu behalten. An dieser lässt sich gut erkennen, ob die Grammatik durch die letzten Veränderungen wesentlich komplexer wurde. Falls die Laufzeit sprunghaft ansteigt, empfiehlt es sich zu überprüfen, ob die Grammatik nicht doch weniger komplex formuliert werden kann.

Da das Generieren vom Compiler durch all diese Arbeitsschritte sehr langwierig wird, lohnt es sich so viele Schritte wie möglich zu automatisieren. Eine .bat-Datei[14] ermöglicht es, fast alle oben aufgeführten Punkte zu automatisieren. Ausschließlich die Aktualisierung der Projektstruktur in der IDE muss noch manuell erfolgen.

2.2.3 SableCC Syntaxbaum

Nachdem SableCC alle nötigen Java-Klassen für das Lexing und Parsing generiert hat, muss nachfolgend der Lexer die Eingabe in Tokens zerteilen. Anschließend ordnet der Parser dann die Tokens in einen Syntaxbaum ein.

[14] siehe unter 2.3 Sample-Projekt

Der Lexer erzeugt eine Liste mit Objekten, die jeweils ein Token repräsentieren. Der Syntaxbaum wird durch verkettete Objekte dargestellt. Die Klassen zu diesen Objekten finden sich im von SableCC generierten Ordner „node“. Grundsätzlich wird für jedes Token, jede Produktionsregel und jede Produktionsregelausprägung einer Grammatik eine Klasse generiert.

Art des Knotens	**Aufbau des Klassennamens**
Token	**T**<Name des Tokens>
Produktionsregel	**P**<Name der Produktionsregel> (ist eine Abstrakte Klasse)
Ausprägung einer Produktionsregel	**A**<Name der Produktionsregelausprägung><Name der Produktionsregel> (erben von der Produktionsregel-Klasse zu der sie gehören)
EOF	Standard Token, welches das Ende vom Tokenstream darstellt

Tabelle 3: Aufbau der Klassennamen, von den von SableCC generierten Klassen

Über die Methode „getText()“ ist es möglich von einem Token-Objekt die Zeichenkette, die hinter diesem Token steht, zu erfahren. Um auf die Bestandteile einer Produktion zugreifen zu können, generiert SableCC Getter für jedes Terminal und Nonterminalsymbol einer Produktions-Ausprägung. Falls in einer Produktionsregel der * oder + Operator verwendet wurde, so gibt der Getter eine LinkedList zurück. Diese LinkedList enthält entweder Objekte einer Token Klasse, oder Objekte einer Abstrakten-Produktionsregel-Klasse. Bei Letzterem ist es notwendig die Objekte zunächst explizit zu casten, bevor sie ausgewertet werden können.

```
Tokens
    id = (['a' .. 'z'] | ['A' .. 'Z']) ( '_' | ['a' .. 'z'] | ['A' .. 'Z'] |  ['0' .. '9'])*;
    allocation = '=';
    digit = ['0' .. '9'];

Productions
    number = digit*;
    set_variable = [name]:id allocation [value]:number;
```

Quelltext 8: Beispiel Grammatik

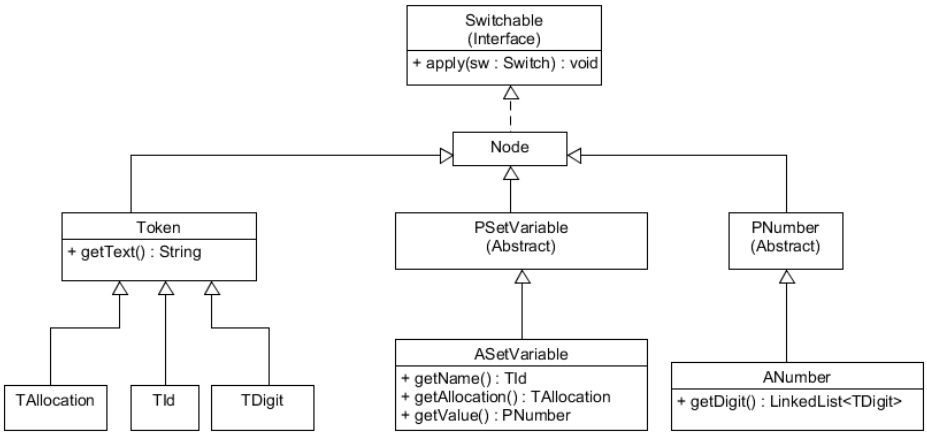


Abbildung 5: die Klassen, die zu der Grammatik aus Quelltext 8 generiert werden

2.2.3.1 Erstellen

Über die Anweisung

```
Lexer lexer = new Lexer(new PushbackReader(new FileReader("input/Eingabe.txt")));
```

Quelltext 9: Lexer ausführen

wird das Lexing durchgeführt. Um zu sehen, wie der Lexer die Eingabe zerteilt hat, hilft der Befehl

```
lexer.next();
```

Quelltext 10: Token von Lexer auslesen

der Reihe nach alle Tokens einzusehen, bis ein Objekt der Klasse EOF erreicht ist, welches das Ende vom Tokenstream darstellt.

Damit nun der Syntaxbaum generiert wird, muss ein Parserobjekt erstellt werden und diesem der Lexer mitgegeben werden.

```
Parser parser = new Parser(lexer);
```

Quelltext 11: Parser ausführen

2.2.3.2 Auswerten

Um den Syntaxbaum auszuwerten, generiert SableCC Klassen, auf dessen Grundlage eine Traversierung durch den Syntaxbaum leicht implementiert werden kann. Diese Klassen befinden sich in dem vom SableCC generierten Ordner „analysis“. Im Folgenden wird die Klasse „DepthFirstAdapter“ als Grundlage für die Implementierung eines eigenen Analyse-Werkzeugs verwendet.

Die Klasse „DepthFirstAdapter“ basiert auf dem Besuchermuster (visitor pattern). Für jede generierte Klasse, die eine Produktionsregel-Ausprägung repräsentiert, gibt es im „DepthFirstAdapter“ eine Methode mit dem Namen „case<Klassenname>“. Diese wird aufgerufen, wenn ein Produktionsregel-Ausprägungs-Knoten vom Visitor („DepthFirstAdapter“) besucht wird. Standardmäßig besucht der „DepthFirstAdapter“ jeden Produktionsregel-Ausprägungs-Knoten, ohne eine Aktion durchzuführen. Tokens-Knoten werden dabei ignoriert.

```
public void caseASetVariable(ASetVariable node) {
    node.getValue().apply(this);
}
```

Quelltext 12: Beispiel-Methode aus dem "DepthFirstAdapter"

Soll dies geändert werden, muss eine eigene Implementation vom „DepthFirstAdapter“ geschrieben werden. Dafür wird eine neue Klasse erstellt, die von der Klasse „DepthFirstAdapter“ erbt. Innerhalb der neuen Klasse können nun die gewünschten Methoden vom „DepthFirstAdapter“ überschrieben werden.

```
public class Visitor extends DepthFirstAdapter {
    public StringBuffer result = "";

    […]

    @Override
    public void caseASetVariable(ASetVariable node) {
        String result = "";

        result += node.getName().getText();

        result += “ = ”;

        Visitor numberVisitor = new Visitor();
        node.getValue().apply(numberVisitor);
        result += numberVisitor.result;

        result += “;”;

        this.result.append(result);
    }

    […]

}
```

Quelltext 13: auf Grunlage vom "DepthFirstAdapter" ein Analysewerkzeug schreiben

2.2.4 Beispiel Compiler

Das nachfolgende Beispiel zeigt, wie eine Eingabe mit Hilfe von SableCC analysiert werden kann. Zunächst wird eine Grammatik definiert, die den Aufbau der Eingabe beschreibt.
Anschließend wird mit SableCC ein Lexer und ein Parser, auf Grundlage der zuvor definierten Grammatik, generiert. Lexer und Parser werden dann eine Eingabe in einen Syntaxbaum darstellen. Dieser Syntaxbaum kann dann mithilfe eines Visitors ausgewertet werden.

Grammatik

Tokens

id = (['a' .. 'z'] | ['A' .. 'Z']) ('_' | ['a' .. 'z'] | ['A' .. 'Z'] | ['0' .. '9']);*
allocation = '=';
digit = ['0' .. '9'];

Productions

number = digit;*
set_variable = [name]:id allocation [value]:number;

Eingabe

„age = 20“

Syntaxbaum

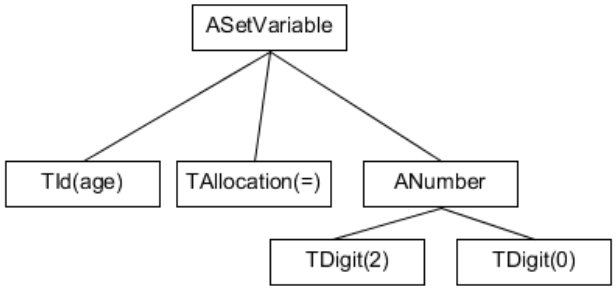


Abbildung 6: die Eingabe wird von Lexer und Parser in einem Syntaxbaum dargestellt

DepthFirstAdapter Methodenaufrufe

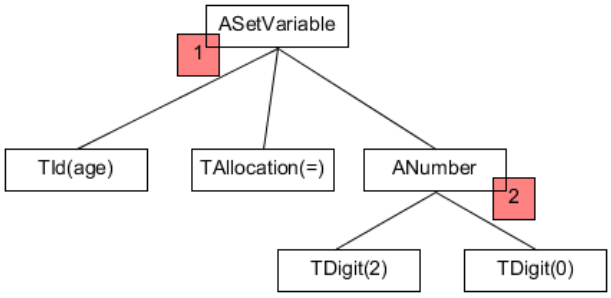


Abbildung 7: die Reihenfolge, in welcher die Knoten vom DepthFirstAdapter abgearbeitet werden (Blätter beziehungsweise Tokens werden nicht besucht!)

Reihenfolge, in welcher die Methoden vom DepthFirstAdapter aufgerufen werden:

1. caseASetVariable
2. caseANumber

Visitor implementieren

Um den Syntaxbaum aus Abbildung 6 aktiv analysieren zu können, muss ein Visitor implementiert werden, der vom DepthFirstAdapter erbt. In dem Visitor können dann die Methoden „caseASetVariable“ und „caseANumber“, mit dem gewünschten Verhalten, überschrieben werden.

```
public class Visitor extends DepthFirstAdapter {
    public StringBuffer result = "";

    @Override
    public void caseANumber (ANumber node) {
        String result = "";

        for (TDigit digit : node.getNumber()) {
            result += digit.getText();
        }

        this.result.append(result);
    }

    @Override
    public void caseASetVariable(ASetVariable node) {
        String result = "";

        result += node.getName().getText();

        result += " = ";

        Visitor numberVisitor = new Visitor();
        node.getValue().apply(numberVisitor);
        result += numberVisitor.result;

        result += ";";

        this.result.append(result);
    }
}
```

Quelltext 14:Implementierung eines Visitors, um einen Syntaxbaum auszuwerten

Reihenfolge, in welcher die Methoden vom Visitor aus Quelltext 14 aufgerufen werden:

1. caseASetVariable	result-Datenfeld: „“
2. caseANumber	result-Datenfeld: „20“
caseASetVariable (am Ende)	result-Datenfeld: „age = 20;“

2.2.5 Häufige Fehlermeldungen im Umgang mit SableCC[15]

2.2.5.1 Generierung vom Lexer und Parser durch SableCC

1. *Redefinition of A<Name einer Produktionsregel>:*
   Den verschiedenen Ausprägungen der Produktionsregel mit dem Namen <Name einer Produktionsregel> müssen explizit unterschiedliche Namen zugewiesen werden (siehe Tabelle 2: Operatoren, für die Definition von Produktionsregeln).

2. *Redefinition of A<Name einer Produktionsregel> <Name eines Symbols>:*
   In der Produktionsregel-Ausprägung mit dem Namen <Name einer Produktionsregel> wird mehrmals das Symbol mit dem Namen <Name eines Symbols> verwendet. Um die Symbole dennoch unterscheiden zu können, müssen den Symbolen explizit unterschiedliche Namen zugewiesen werden(siehe Tabelle 2: Operatoren, für die Definition von Produktionsregeln).

3. *java.lang.OutOfMemoryError:*
   Eine zu komplexe Grammatik kann dazu führen, dass SableCC beim Generieren von dem Compiler nicht über genügend Arbeitsspeicher verfügt und somit ein OutOfMemoryError wirft. Um den Error zu verhindern, kann es helfen, wenn beim ausführen der SableCC.jar mehr Arbeitsspeicher freigegeben wird. Grundsätzlich empfiehlt es sich jedoch, noch einmal die Grammatik genau zu untersuchen und zu prüfen, ob sie nicht unnötig komplex ist.

4. *Shift/Reduce Conflict:*
   Die Grammatik ist mehrdeutig und muss umformuliert werden.

5. *Reduce/Reduce Conflict:*
   Die Grammatik ist mehrdeutig und muss umformuliert werden.

[15] http://sablecc.sourceforge.net/downloads/thesis.pdf

2.2.5.2 Ausführen vom Lexer

1. *Pushback buffer overflow*[16]*:*
   Der Puffer vom PushbackBuffer ist zu klein und muss vergrößert werden.

```
Lexer lexer = new Lexer(new PushbackReader(new FileReader("input/Eingabe.txt"), 1024));
```

Quelltext 15: Lexer mit einen größeren Puffer ausführen

2. *[<Zeile>,<Spalte>] Unknown token: <Zeichen>:*
   Die Zeichen <Zeichen>, welche in der Eingabe Datei in der Zeile <Zeile> und Spalte <Spalte> stehen, konnten keinem Token zugeordnet werden. In der Grammatik muss die Definition der Tokens angepasst werden, falls gewünscht ist, dass die Zeichen <Zeichen> verarbeitet werden können.

2.2.5.3 Java-Compiler und Eclipse

1. *Method is exceeding the 65535 bytes limit*
   Java-Methoden dürfen nicht größer als 8,19 KB sein. Wenn SableCC eine Methode generiert, die größer ist, so muss die Komplexität der Grammatik verringert werden.
2. *Eclipse stürzt ab beim Indexieren vom Projekt*
   Je komplexer die Grammatik wird, desto größer werden auch die Java-Dateien, die von SableCC generiert werden. Dies kann zur Folge haben, dass Eclipse bei dem Indexieren des Projektes zu wenig Arbeitsspeicher hat und deshalb abstürzt. Um einen Absturz zu verhindern, muss für Eclipse mehr Arbeitsspeicher freigeben werden[17]. Alternativ hilft es auch die Komplexität der Grammatik zu verringern.

[16] http://lists.sablecc.org/pipermail/sablecc-discussion/msg00773.html

[17] http://blog.essential-bytes.de/10-performance-tipps-fuer-eclipse/

2.2.6 Tipps im Umgang mit SableCC

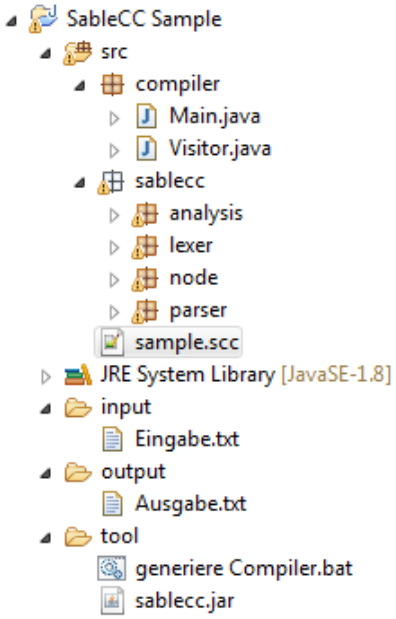
(wird noch ausformuliert)

- InParens-Pattern
- Part-Pattern → um Mehrdeutigkeit zu verhindern
    - Schlecht: wenn C+D+C gewollt
      A = B*
      B = C | D | +;
      Ist schlecht, weil auch CCC gehen würde. Das wiederum führt schnell zu Konflikten mit anderen Produktionen.
    - Gut:
      A = B B* Value
      B = Value +
      Value = C | D
- VisitorDataObject
- Datenfelder im Visitor
- Tricks bei der Fehler-/Komplexitätssuche

2.3 Sample-Projekt

Der Compiler ist in der Lage Excel-VBA Kommentare und Excel-VBA Variablen Deklarationen in Java Code umzuschreiben.

Aufbau vom Projekt



Das Projekt unterteilt sich in vier Ordner. Im src-Ordner befindet sich die SableCC-Grammatik („sample.scc“), der Java-Code, welcher von SableCC generiert wird („sablecc“) und der komplett selbst geschriebene Java-Code („compiler“).
Die Klasse „Main“ beinhaltet die Main-Methode. Sie sorgt dafür, dass eine Eingabedatei eingelesen, kompiliert und das Ergebnis wieder in eine Ausgabedatei ausgegeben wird. Die Klasse „Visitor“ ist das Analysewerkzeug. Sie wertet einen vom Parser generierten Syntaxbaum aus und führt die eigentliche Übersetzungsarbeit durch.
Die Ordner „input“ und „output“ beinhalten jeweils die Eingabedatei, bzw. die nach dem Kompiliervorgang erzeugte Ausgabedatei.
Im Ordner „tool“ befindet sich eine .bat-Datei. Die Datei automatisiert den Generiervorgang vom Lexer und Parser.

3 SableCC konkret auf das Projekt bezogen

(wird noch ergänzt)

- Projektaufbau beschreiben
- Ablauf vom Compileprozess abbilden
    - Compilerprozess VBA → while raus → .Name → VisitorDateObject aufbauen → Syntaxbaum aufbauen → Visitor drüberlaufen lassen → Java Code
- (Methoden außerhalb definieren) sehr Projekt spezifisch
- While Stmt Auflösen

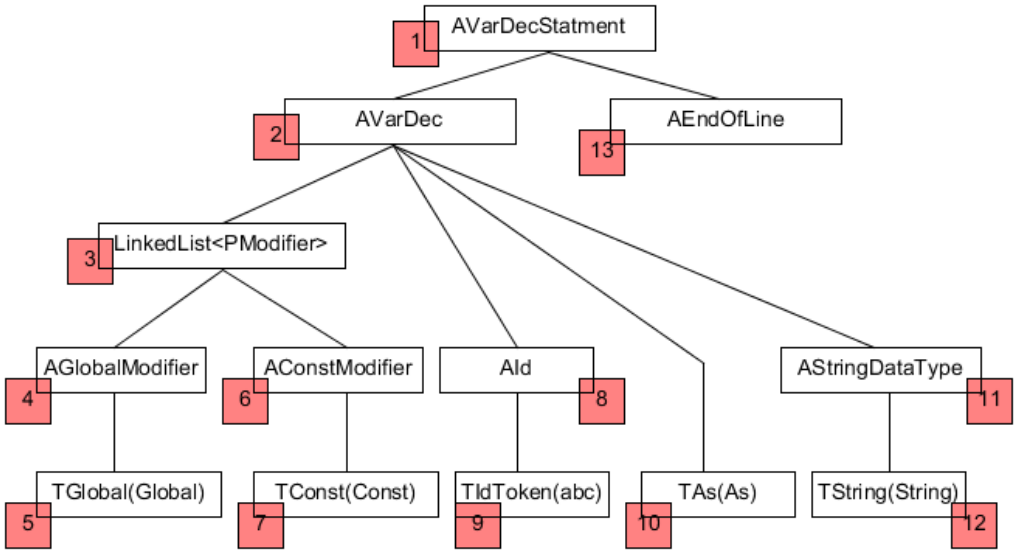
Grammatik

(siehe Sample-Projekt)

Eingabe

„Global Const abc As String\n“

Syntaxbaum



In rot die Reihenfolge (Tiefensuche), in welcher die Knoten des Syntaxbaums vom „DepthFirstAdapter“ abgearbeitet werden.

Hinweis: der DepthFirstAdapter würde 8 und 9 erst nach 12 besuchen! Dieses Verhalten wurde jedoch im Visitor, welcher vom DepthFirstAdapter erbt, abgeändert (siehe im Visitor die Methode „caseAVarDec“).

Visitor

(siehe Sample-Projekt)

Visitor Methodenaufrufe

1. caseAVarDecStatment
2. caseAVarDec
3. caseAGlobalModifier „public“
4. caseAConstModifier „final“
5. caseAStringDataType „String“
6. caseAId „abc“

caseAVarDec (am Ende) „public final String abc;“

7. caseAEndOfLine „\n“

caseAVarDecStatment (am Ende) „public final String abc;\n“

4 Einführung in die Weiterentwicklung

(wird noch ergänzt)

- Syntaxfehler beheben
  - GoTo‘s
    - Alle 100 GoTo‘s im VB-Code auflösen
  - ByRef/ByValue
  - Excel I/O
  - explizite Casts
  - int zu boolean (geht in VBA automatisch)
  - MsgBox & Alert
  - Klassenname der aktuellen Klasse innerhalb einer Typdefinition
    - innerhalb einer Klasse den Klassennamen nicht erwähnen!
- Debuggen, wenn beim Ausführen Fehler geworfen werden
- Finaler Test → Output vom in VBA geschriebenen DDL-Generator mit dem Output vom in Java geschriebenen DDL-Generator vergleichen (Vergleich am einfachsten mit RTC umsetzbar)
- wo die abgespeckte Exel-Datei liegt
- Formatierung nicht zerhauen!