1 Sprach-Features

- Klassen
- Felder
- Methoden (mit Parametern)
- Konstruktoren (mit Parametern)
- Standardkonstruktoren
- Lokale Variablen
- Zuweisungen (Feld- und lokale Variablen)
- Arithmetik (`+`, `-`, `*`, `/`, `%`, Klammern, Korrekte Operations-Präzedenz)
- Arithmetische Zuweisungen (`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`)
- Vergleichsoperationen (`<`, `>`, `<=`, `>=`, `==`, `!=`)
- Boolsche Operationen (`||`, `&&`)
- Unäre Operationen (`-`,)
- Binar-Operationen (`&`, `|`, `^`)
- Pre/Post-Inkrement & Dekrement
- Kontrollflussstrukturen:
  - If/Else
  - While
  - For
  - Return (mit/ohne Rückgabewert)
- Default-Werte für alle Klassenfelder
- Mehrere Klassen in einer Datei
- Implizites `this`
- Beliebig verschachtelte Namensketten
- Beliebige Deklarationsreihenfolge
- Literale für Integer, Characters, Booleans
- Platzhalter/Separatoren in Integerliteralen (z.B. `1_000_000`)
- Deklaration und Zuweisung in einer Anweisung
- Beliebig verschachtelte Blöcke
- Überladung von Methoden & Konstruktoren

2 Lexer & Parser

2.1 Lexer

Der Lexer wurde mit dem Alex tool implementiert. Dieser ist dafür zuständig den langen String in einzelne Tokens umzuwandeln. In der Alex Datei gibt es für jedes Token einen regulären Ausdruck. Bei den meisten Tokens ist das einfach das Schlüsselwort. Etwas komplexer waren Identifier, Integerliterale Strings und Chars. Für die Definition wurde sich eng an die offizielle Java Language Specification gehalten. Es ist beispielsweise auch möglich Unterstriche in Integerliterale einzubauen (Bsp.: `234_343_000`) Es sind fast alle Schlüsselwörter von Java im Lexer implementiert, auch wenn nicht alle davon vom Parser geparst werden können. Whitespace und Kommentare werden direkt ignoriert und verworfen. Für Charliterale und Integerliterale gibt es auch spezielle Fehlermeldungen. Die meisten Tokens haben nur die Information, zu welchem Keyword sie gehören. Eine Ausnahme bilden der Identifier und die Literale. Für den Identifier wird noch der Name gespeichert und für die Literale der entsprechende Wert. Mit der Funktion alexScanTokens kann dann ein beliebiger String in Tokens umgewandelt werden.

Die komplexeren Tokens haben Unittests, welche mit dem Testframework HUnit geschrieben wurden. Es gibt Tests für Kommentare, Identifier, Literale und ein paar weitere Tokens.

2.2 Parser

Der Parser wurde mit dem Happy tool implementiert. Er baut aus einer Liste von Tokens einen ungetypten AST. Wir haben bereits eine Grammatik bekommen und mussten diese noch in den AST umwandeln.

Um den Parser aufzubauen wurde zuerst ein Großteil der Grammatik auskommentiert und Stück für Stück wurden die Umwandlungen hinzugefügt. Immer wenn ein neues Feature umgesetzt wurde, wurde dafür ein weiterer Unit Test geschrieben. Es gibt also für jede komplexe Ableitungsregel mindestens einen Unittest.

2.2.1 Klassenaufbau

Als erstes wurden leere Konstruktoren Methoden und Felder umgesetzt. Da in Java Konstruktoren, Methoden und Felder durcheinander vorkommen können geben die Ableitungsregeln einen Datentyp namens ‘MemberDeclaration‘ zurück. Die classbodydeclarations Regel baut dann einen 3-Tupel mit einer Liste aus Konstruktoren, einer aus Methoden und einer aus Feldern. Über pattern matching werden diese Listen dann erweitert und in der darüberliegenden Regel schließlich extrahiert.

Bei folgender Klasse:

```
class TestClass {
    int field;

    TestClass() {}
```

```
    void foo() {}
}
```

würde die Regel folgendes Tupel zurückgeben:

```
(
    [ConstructorDeclaration "TestClass" [] (Block [])],
    [MethodDeclaration "void" "foo" [] (Block [])],
    [VariableDeclaration "int" "field" Nothing]
)
```

und folgende Klasse wird erstellt

```
Class "TestClass"
    [ConstructorDeclaration "TestClass" [] (Block [])]
    [MethodDeclaration "void" "foo" [] (Block [])]
    [VariableDeclaration "int" "field" Nothing]
```

Das Nothing ist in diesem Fall ein Platzhalter für eine Zuweisung, da unser Compiler auch Zuweisung bei der Felddeklaration unterstützt.

In Java ist es möglich mehrere Variablen in einer Zeile zu deklarieren (Bsp.: ‘int x, y;‘). Beim Parsen ergiebt sich dann die Schwierigkeit, dass man in dem Moment, wo man die Variable parst nicht weiß welchen Datentyp diese hat. Aus diesem Grund gibt es den Datentyp Declarator, welcher nur den Identifier und eventuell eine Zuweisung enthält. In den darüberliegenden Regeln fielddeclaration und localvariabledeclaration wird dann die Typinformation hinzugefügt mithilfe der Funktion convertDeclarator.

2.2.2 Syntactic Sugar

Für die Zuweisung wird auch die Kombination mit Rechenoperatoren unterstützt. Das ganze ist durch Syntactic Sugar im Parser umgesetzt. Wenn es einen Zuweisungsoperator gibt, dann wird der Ausdruck in eine Zuweisung und Rechnung aufgeteilt. Bsp.: `x += 3;` wird umgewandelt in `x = x + 3`.

For-Schleifen wurde auch rein im Parser durch Syntactic Sugar implementiert. Eine For-Schleife wird dabei in eine While-Schleife umgewandelt. Dafür wird zuerst ein Block erstellt, sodass die deklarierten Variablen auch nur für den Bereich der Schleife gültig sind. Die Bedingung der For-Schleife kann in die While-Schleife übernommen werden. Innerhalb der While-Schleife folgen zuerst die Statements, die im Block der For-Schleife waren und danach die Update-Statements.

```
for (int i = 0; i < 9; i++) {
    foo();
}
```

wird umgewandelt in:

```
{
    int i = 0;
    while (i < 9) {
        foo();
        i++;
    }
}
```

3 Typecheck

3.1 Überblick & Struktur

Die Typprüfung beginnt mit der Funktion `typeCheckCompilationUnit`, die eine Kompilationseinheit als Eingabe erhält. Diese Kompilationseinheit besteht aus einer Liste von Klassen. Jede Klasse wird einzeln durch die Funktion `typeCheckClass` überprüft. Innerhalb dieser Funktion wird eine Symboltabelle erstellt, die den Namen der Klasse als Typ und `this` als Identifier enthält. Diese Symboltabelle wird verwendet, um Typinformationen nach dem Lokalitätsprinzip während der Typprüfung zugänglich zu machen und zu verwalten. Die Typprüfung einer Klasse umfasst die Überprüfung aller Konstruktoren, Methoden und Felder. Die Methode `typeCheckConstructorDeclaration` ist für die Typprüfung einzelner Konstruktordeklarationen verantwortlich, während `typeCheckMethodDeclaration` für die Typprüfung einzelner Methodendeklarationen zuständig ist. Beide Funktionen überprüfen die Parameter und den Rumpf der jeweiligen Konstruktoren bzw. Methoden. Der Rumpf wird durch rekursive Aufrufe von `typeCheckStatement` überprüft, die verschiedene Arten von Anweisungen wie If-Anweisungen, While-Schleifen, Rückgabeanweisungen und Blockanweisungen behandelt.

3.2 Ablauf & Symboltabellen

Eine zentrale Komponente des Typecheckers ist die Symboltabelle (ßymtab"), die Informationen über die Bezeichner und ihre zugehörigen Datentypen speichert. Die Symboltabelle wird kontinuierlich angepasst, während der Typechecker die verschiedenen Teile des Programms durchläuft.

3.2.1 Anpassung der Symboltabelle

- **Klassenkontext** Beim Typcheck einer Klasse wird eine initiale Symboltabelle erstellt, die die `this`-Referenz enthält. Dies geschieht in der Funktion `typeCheckClass`.
- **Konstruktorkontext** Innerhalb eines Konstruktors wird die Symboltabelle um die Parameter des Konstruktors erweitert. Dies geschieht in `typeCheckConstructorDeclaration`. Der Rückgabetyp eines Konstruktors ist implizit `void`, was überprüft wird, um sicherzustellen, dass kein Wert zurückgegeben wird.
- **Methodenkontext** Innerhalb einer Methode wird die Symboltabelle um die Parameter der Methode erweitert sowie den Rückgabetyp der Methode, um die einzelnen Returns dagegen zu prüfen. Dies geschieht in `typeCheckMethodDeclaration`.
- **Blockkontext** Bei der Überprüfung eines Blocks (`typeCheckStatement` für Block) wird die Symboltabelle für jede Anweisung innerhalb des Blocks aktualisiert. Lokale Variablen, die innerhalb des Blocks deklariert werden, werden zur Symboltabelle hinzugefügt. Das bedeutet, dass automatisch, sobald der Block zu Ende ist, alle dort deklarierten Variablen danach nicht mehr zugänglich sind.

3.2.2 Unterscheidung zwischen lokalen und Feldvariablen

Bei der Typprüfung von Referenzen (`typeCheckExpression` für Reference) wird zuerst in der Symboltabelle nach dem Bezeichner gesucht. Sollte dieser gefunden werden, handelt es sich um eine lokale Variable. Wenn der Bezeichner nicht gefunden wird, wird angenommen, dass es sich um eine Feldvariable handelt. In diesem Fall wird die Klasse, zu der die `this`-Referenz gehört, durchsucht, um die Feldvariable zu finden. Dies ermöglicht die Unterscheidung zwischen lokalen Variablen und Feldvariablen. Dies ist auch nur möglich, da wir die Feldvariablen und Methoden nicht in die Symboltabelle gelegt haben und stattdessen nur die `this`-Referenz.

3.3 Fehlerbehandlung

Ein zentraler Aspekt des Typecheckers ist die Fehlerbehandlung. Bei Typinkonsistenzen oder ungültigen Operationen werden aussagekräftige Fehlermeldungen generiert. Beispiele für solche Fehlermeldungen sind:

- **Typinkonsistenzen** Wenn der Rückgabetyp einer Methode nicht mit dem deklarierten Rückgabetyp übereinstimmt oder die Anzahl der Parameter nicht übereinstimmt.
- **Ungültige Operationen** Wenn eine arithmetische Operation auf inkompatiblen Typen durchgeführt wird.
- **Nicht gefundene Bezeichner** Wenn eine Referenz auf eine nicht definierte Variable verweist.

Diese Fehlermeldungen helfen Entwicklern, die Ursachen von Typfehlern schnell zu identifizieren und zu beheben. Generell sind diese oftmals sehr spezifisch, was das Problem recht schnell identifizieren sollte. Z.B. falsche Reihenfolge / falsche Typen der Parameter beim Methodenaufruf sind direkt erkennbar.

3.4 Typprüfung von Kontrollstrukturen und Blöcken

3.4.1 If-Anweisungen

Bei der Typprüfung einer If-Anweisung (`typeCheckStatement` für If) wird zuerst der Typ der Bedingung überprüft, um sicherzustellen, dass es sich um einen booleschen Ausdruck handelt. Anschließend werden die Then- und Else-Zweige geprüft. Der Typ der If-Anweisung selbst wird durch die Vereinheitlichung der Typen der Then- und Else-Zweige bestimmt. Falls einer der Zweige keinen Rückgabewert hat, wird angenommen, dass der Rückgabewert `void` ist. Dies wurde so gelöst, um im Typchecker feststellen zu können, ob beide Zweige einen Return haben. Wenn nur einer der Zweige ein Return hat, wird im umliegenden Block ein weiteres benötigt, was durch den Typ `void` erzwungen wird. Dadurch weiß der Typchecker Bescheid.

3.4.2 Block-Anweisungen

Die Typprüfung eines Blocks erfolgt in `typeCheckStatement` für Block. Jede Anweisung im Block wird nacheinander überprüft und die Symboltabelle wird entsprechend aktualisiert. Der Typ des Blocks wird durch die Vereinheitlichung der Typen aller Anweisungen im Block bestimmt. Wenn der Block keine Anweisungen hat, wird der Typ `void` angenommen.

3.4.3 Rückgabeanweisungen

Die Typprüfung einer Rückgabeanweisung (`typeCheckStatement` für Return) überprüft, ob der Rückgabewert der Anweisung mit dem deklarierten Rückgabetyp der Methode übereinstimmt. Dafür wurde zu Beginn der Methodentypprüfung der Rückgabetyp der Methode in die Symboltabelle eingetragen. Wenn der Rückgabewert `null` ist, wird überprüft, ob der deklarierte Rückgabetyp ein Objekttyp ist. Dies stellt sicher, dass Methoden immer den korrekten Typ zurückgeben. Generell wird bei der Prüfung nach dem UpperBound geschaut und ebenfalls wird nachgeschaut, ob, wenn der Rückgabetyp `Object` ist, der Return-Wert auch eine tatsächlich existierende Klasse ist, indem in die Klassentabelle geschaut wird.

3.4.4 Konstruktorüberladung und -prüfung

Die Typprüfung unterstützt Konstruktorüberladung. Bei der Typprüfung von Konstruktoraufrufen (`typeCheckStatementExpression` für `ConstructorCall`) wird überprüft, ob es mehrere Konstruktoren mit derselben Anzahl von Parametern gibt. Falls mehrere passende Konstruktoren gefunden werden, wird ein Fehler gemeldet.

- **Parameterabgleich** Die Parameter eines Konstruktors werden gegen die Argumente des Aufrufs abgeglichen. Dies umfasst die Prüfung der Typen und, falls es sich um `null` handelt, die Überprüfung, ob der Parameter ein Objekttyp ist.
- **Fehlerbehandlung** Wenn kein passender Konstruktor gefunden wird, wird eine detaillierte Fehlermeldung generiert, die die erwarteten Signaturen und die tatsächlichen Argumenttypen anzeigt. Wenn mehrere passende Konstruktoren gefunden werden, wird ebenfalls ein Fehler gemeldet.

3.4.5 Methodenüberladung und -prüfung

Die Typprüfung unterstützt auch Methodenüberladung. Bei der Typprüfung von Methodenaufrufen (`typeCheckStatementExpression` für `MethodCall`) wird überprüft, ob es mehrere Methoden mit demselben Namen, aber unterschiedlichen Parametertypen gibt.

- **Parameterabgleich** Die Parameter einer Methode werden gegen die Argumente des Aufrufs abgeglichen. Dies umfasst die Prüfung der Typen und, falls es sich um `null` handelt, die Überprüfung, ob der Parameter ein Objekttyp ist.
- **Fehlerbehandlung** Wenn keine passende Methode gefunden wird, wird eine detaillierte Fehlermeldung generiert, die die erwarteten Signaturen und die tatsächlichen Argumenttypen anzeigt. Wenn mehrere passende Methoden gefunden werden, wird ebenfalls ein Fehler gemeldet.

4 Bytecodegenerierung

Die Bytecodegenerierung ist letztendlich eine zweistufige Transformation:

```
Getypter AST -> [ClassFile] -> [[Word8]]
```

Vom AST, der bereits den Typcheck durchlaufen hat, wird zunächst eine Abbildung in die einzelnen ClassFiles vorgenommen. Diese ClassFiles werden anschließend in deren Byte-Repräsentation serialisiert.

4.1 Codegenerierung

Für die erste der beiden Transformationen (`Getypter AST -> [ClassFile]`) werden die Konzepte der "Builder" und "Assembler" eingeführt. Sie sind wie folgt definiert:

```
type ClassFileBuilder a = a -> ClassFile -> ClassFile
type Assembler a = ([ConstantInfo], [Operation], [String]) -> a
                -> ([ConstantInfo], [Operation], [String])
```

Die Idee hinter beiden ist, dass sie jeweils zwei Inputs haben, wobei der Rückgabewert immer den gleichen Typ hat wie einer der Inputs. Das erlaubt es, eine Faltung durchzuführen. Ein ClassFileBuilder z.B bekommt als ersten Parameter den AST, und als zweiten Parameter (und Rückgabewert) eine ClassFile. Soll nun eine Klasse gebaut werden, wird der ClassFileBuilder mit dem AST und einer leeren ClassFile aufgerufen. Der Zustand dieser anfangs leeren ClassFile wird durch alle folgenden Builder/Assembler durchgeschleift, was es erlaubt, nach und nach kleinere Transformationen auf sie anzuwenden. Der Nutzer ruft beispielsweise die Funktion `classBuilder` auf. Diese wendet nach und nach folgende Transformationen an:

1. Allen Konstruktoren werden Initialisierer aller Felder hinzugefügt
2. Für jedes Feld der Klasse wird ein Eintrag im Konstantenpool & der Classfile erstellt
3. Für jede Methode wird ein Eintrag im Konstantenpool & der Classfile erstellt
4. Allen Methoden wird der zugehörige Bytecode erstellt und zugewiesen
5. Allen Konstruktoren wird der zugehörige Bytecode erstellt und zugewiesen

Die Unterteilung von Deklaration der Methoden/Konstruktoren und Bytecodeerzeugung ist deswegen notwendig, weil der Code einer Methode auch eine andere, erst nachher deklarierte

Methode aufrufen kann. Nach dem Hinzufügen der Deklarationen sind alle Methoden/Konstruktoren der Klasse bekannt. Wie oben beschrieben wird auch hier der Zustand über alle Faltungen mitgenommen. Jeder Schritt hat Zugriff auf alle Daten, die aus dem vorherigen Schritt bleiben. Sukzessive wird eine korrekte ClassFile aufgebaut. Besonders interessant sind hierbei die beiden letzten Schritte. Dort wird das Verhalten jeder einzelnen Methode/-Konstruktor in Bytecode übersetzt. In diesem Schritt werden zusätzlich zu den `Buildern` noch die `Assembler` verwendet (Definition siehe oben.). Die Assembler funktionieren ähnlich wie die Builder, arbeiten allerdings nicht auf einer ClassFile, sondern auf dem Inhalt einer Methode; Sie verarbeiten jeweils ein Tupel der Form:

```
([ConstantInfo], [Operation], [String])
```

Dieses repräsentiert:

```
(Konstantenpool, Bytecode, Lokale Variablen)
```

In der Praxis werden meist nur Bytecode und Konstanten hinzugefügt. Prinzipiell können Assembler auch Code/Konstanten entfernen oder modifizieren. Als Beispiel dient hier der Assembler `assembleExpression`:

```
assembleExpression (constants, ops, lvars) (TypedExpression _ NullLiteral)
    = (constants, ops ++ [Opaconst_null], lvars)
```

Hier werden die Konstanten und lokalen Variablen des Inputs nicht berührt, dem Bytecode wird lediglich die Operation `aconst_null` hinzugefügt. Damit ist das Verhalten des gematchten Inputs - eines Nullliterals - abgebildet. Die Assembler rufen sich teilweise rekursiv selbst auf, da ja auch der AST verschachteltes Verhalten abbilden kann. Der Startpunkt für die Assembly einer Methode ist der Builder `methodAssembler`. Dieser entspricht Schritt 3 in der obigen Übersicht.

4.2 Serialisierung

Damit Bytecode generiert werden kann, braucht es Strukturen, die die Daten halten, die letztendlich serialisiert werden. Die JVM erwartet den kompilierten Code in handliche Pakete verpackt. Die Struktur dieser Pakete ist hier dokumentiert. Jede Struktur, die in dieser übergreifenden Class File vorkommt, haben wir in Haskell abgebildet. Es gibt z.B die Struktur ClassFile", die wiederum weitere Strukturen wie z.B Informationen über Felder oder Methoden der Klasse beinhaltet. Alle diese Strukturen implementieren folgende Type-Class:

```
class Serializable a where
    serialize :: a -> [Word8]
```

Hier ist ein Beispiel anhand der Serialisierung der einzelnen Operationen:

```
instance Serializable Operation where
    serialize Opiadd                    = [0x60]
    serialize Opisub                    = [0x64]
    serialize Opimul                    = [0x68]
    ...
    serialize (Opgetfield index)        = 0xB4 : unpackWord16 index
```

Die Struktur ClassFile ruft für deren Kinder rekursiv diese `serialize` Funktion auf und konkateniert die Ergebnisse. Am Ende bleibt eine flache Word8-Liste übrig, die Serialisierung ist damit abgeschlossen. Da der Typecheck sicherstellt, dass alle referenzierten Methoden/-Felder gültig sind, kann die Übersetzung der einzelnen Klassen voneinander unabhängig geschehen.

5 Aufgabenverteilung

- **Marvin Schlegel** Parser & Lexer
- **Fabian Noll** Semantik- & Typcheck
- **Christian Brier** Bytecodegenerierung
- **Matthias Raba** Bytecodegenerierung

Marvin Schlegel und Fabian Noll haben ihre Teilaufgaben eigenständig bearbeitet.

Die Bytecodegenerierung wurde von Matthias Raba und Christian Brier im Stile des Pair Programmings zu zweit erarbeitet. Durch bisher gute Erfahrungen in vorherigen Projekten, sowie dem Interesse, alle Teile der Bytecodegenerierung zu sehen, wurde diese Programmierungsform als die Beste ausgewählt.

Während der Implementierungsphase wurde viel zwischen den 3 einzelnen Teams kommuniziert. Wurden Fehler in einer der Komponenten gefunden, wurden die jeweiligen Verantwortlichen informiert um das Problem zu beheben. Jedes der Teams arbeitete auf einem eigenen Branch, die einzelnen Beiträge wurde regelmäßig auf dem master-Branch zusammengeführt. Insgesamt lief die Implementierungsphase wie geplant und ohne weitere Komplikationen ab.