Parser-Dokumentation - ”Nicht Haskell 2.0”

Dieses Dokument soll die Funktionsweise des Parsers erklären. Der Parser wurde gemeinsam von Maximilian Stahl (i22035) und Jannik Rombach (i22035) im Stil des Pair Programming entwickelt. Beide Entwickler konnten gleichberechtigt ihre Ideen einbringen und die Implementierung effizient umsetzen. Änderungen wurden iterativ vorgenommen, basierend auf der Kommunikation mit den nachgelagerten Bereichen der Semantikprüfung und der Bytecodegenerierung. Speziell entwickelte Unit-Tests wurden verwendet, um den erstellten abstrakten Syntaxbaum (AST) zu testen.

In den folgenden Abschnitten werden die implementierte Java-Grammatik, die Umsetzung des Parsers und Scanners, die Umwandlung in den abstrakten Syntaxbaum (inklusive syntaktischem Zucker) sowie das Testkonzept des ASTs beschrieben.

1 Umgesetzte Grammatik

Klassen

Es können mehrere Klassen innerhalb einer Datei definiert werden.

Felder

Variablen, die innerhalb von Klassen definiert werden und den Zustand eines Objekts speichern.

Methoden (mit Parametern)

Funktionen innerhalb von Klassen, die Aktionen definieren und Parameter für zusätzliche Informationen akzeptieren.

Standardkonstruktoren

Wird kein Konstruktor angegeben, wird automatisch ein Standardkonstruktor erstellt.

Konstruktoren (mit Parametern)

Initialisieren neue Objekte und können Parameter zur Initialisierung der Felder akzep-

tieren.

Lokale Variablen

Werden innerhalb von Methoden definiert und existieren nur während der Ausführung der Methode.

Zugriffsmodifikatoren

`public`, `protected`, `private`. Wird kein Modifikator angegeben, wird er automatisch auf `public` gesetzt.

Datentypen

Die Datentypen `int`, `char` und `boolean` sind umgesetzt.

Initialisieren von Objekten durch `new`

Objekte werden mit dem Schlüsselwort `new` instanziiert.

Zuweisungen

Sowohl bei Feldern als auch bei lokalen Variablen.

Rechenoperatoren

=, +, −, *, %, /

Vergleichsoperatoren

<, >, >=, <=, ==, !=, !

Logische Operatoren

&&, ||

Inkrementieren und Dekrementieren

Sowohl Präfix- als auch Suffixnotation sind möglich: `i++`, `++i`, `i--`, `--i`

Unterscheidung in Unäre und Binäre Operationen

Die Expressions werden grundsätzlich in zwei Arten unterteilt: `UnaryNode`, die alle Expressions mit einem einzigen Operanden beinhaltet, und `BinaryNode`, die aus zwei Operanden und einem Operator besteht.

Kontrollflussstrukturen

If/Else

Unterstützung von `if`, `else if` und `else`.

While

Schleife, die wiederholt wird, solange eine Bedingung wahr ist.

Do-While

Ähnlich der `while`-Schleife, wird jedoch mindestens einmal ausgeführt.

For

Implementiert als `while`-Schleife, mit einer Initialisierung, Bedingung und Inkrementierung.

Return

Sowohl `void`-Rückgaben als auch Rückgaben von Werten sind möglich.

2 Scanner und Parser

Der Scanner und Parser wurden mit ANTLR implementiert und verwenden das Visitor-Pattern, um den Quellcode in einen abstrakten Syntaxbaum (AST) umzuwandeln. Die Grammatik für die Sprache, die wir parsen, ist in ANTLR definiert und deckt die grundlegenden Sprachkonstrukte wie Klassen, Methoden, Felder und Kontrollflussstrukturen ab.

2.1 Scanner

Der Scanner (auch Lexer genannt) ist dafür verantwortlich, den Quellcode in eine Liste von Tokens zu zerlegen. Ein Token ist ein einzelnes Element wie ein Schlüsselwort, ein Identifier, ein Operator oder ein Literal. Der Scanner erkennt diese Tokens anhand von regulären Ausdrücken, die in der ANTLR-Grammatik definiert sind. Whitespace und Kommentare werden vom Scanner ignoriert und verworfen. Spezielle Fehlermeldungen werden für ungültige Zeichenfolgen, wie z.B. unvollständige Zeichenliterale, generiert.

2.2 Parser

Der Parser analysiert die vom Scanner erzeugte Liste von Tokens und baut daraus einen ungetypten AST. Die Grammatik definiert die Syntaxregeln, die der Parser verwendet, um die Struktur des Quellcodes zu verstehen und entsprechende AST-Knoten zu erzeugen. Der Parser wurde iter-

ativ entwickelt, wobei für jedes neue Sprachfeature ein Unit-Test geschrieben wurde, um die korrekte Implementierung sicherzustellen.

2.3 Verwendung des Visitor-Patterns

Unser Parser verwendet das Visitor-Pattern, um den AST aufzubauen. Das Visitor-Pattern ermöglicht eine klare Trennung der Logik für die Verarbeitung der verschiedenen Knotenarten des AST. Jeder Knotentyp hat eine entsprechende visit-Methode, die die spezifische Logik zur Verarbeitung dieses Knotens enthält.

Vorteile des Visitor-Patterns in unserem Projekt:

- **Trennung der Logik**: Das Visitor-Pattern trennt die Logik zur Verarbeitung der verschiedenen Knotenarten, was den Code sauberer und wartbarer macht.
- **Erweiterbarkeit**: Neue Knotentypen können leicht hinzugefügt werden, indem einfach neue visit-Methoden implementiert werden, ohne dass bestehender Code geändert werden muss.
- **Lesbarkeit**: Der Code ist besser lesbar, da die Logik zur Verarbeitung jedes Knotentyps an einem Ort konzentriert ist.

3 Testkonzept

Um die korrekte Funktion des Parsers sicherzustellen, haben wir Unit-Tests implementiert. Diese Tests überprüfen die einzelnen Komponenten des Parsers und den Aufbau des ASTs für verschiedene Java-Programme. Der Testprozess stellt sicher, dass jede Erweiterung und Änderung der Grammatik und des Parsers korrekt implementiert ist. Die Tests wurden iterativ entwickelt, um alle möglichen Szenarien abzudecken und die Robustheit des Parsers zu gewährleisten.

4 Syntaktischer Zucker

In unserem Projekt haben wir einige Konstrukte durch syntaktischen Zucker vereinfacht:

4.1 For-Schleifen mit While-Schleifen

For-Schleifen wurden im Parser durch syntaktischen Zucker implementiert, indem sie in While-Schleifen umgewandelt werden. Dies geschieht, indem zunächst ein Block erstellt wird, sodass die deklarierten Variablen nur innerhalb des Bereichs der Schleife gültig sind. Die Bedingung der For-Schleife wird in die While-Schleife übernommen. Innerhalb der While-Schleife folgen zuerst die Statements, die im Block der For-Schleife enthalten waren, gefolgt von den Update-Statements (meist Inkrementieren und Dekrementieren).

```
for (int i = 0; i < 10; i++) {
    foo();
}
```

wird umgewandelt in:

```
{
    int i = 0;
    while (i < 10) {
        foo();
        i++;
    }
}
```

4.2 Do-While-Schleifen mit While-Schleifen

Do-While-Schleifen werden ebenfalls durch While-Schleifen umgesetzt. Dabei wird sichergestellt, dass der Block mindestens einmal ausgeführt wird, bevor die Bedingung überprüft wird.

```
do {
    foo();
} while (i < 10);
```

wird umgewandelt in:

```
{
    foo();
    while (i < 10) {
```

```
        foo();
    }
}
```

4.3 Automatisches Hinzufügen des Standardkonstruktors

Falls keine Konstruktoren in einer Klasse angegeben werden, fügt der Parser automatisch einen Standardkonstruktor hinzu. Dieser Konstruktor ist öffentlich und hat keinen Inhalt.

```
class Example {
    int a;
}
```

wird umgewandelt in:

```
class Example {
    int a;

    public Example() {

    }
}
```

Durch diese Transformationen wird der Quellcode vereinfacht und die Lesbarkeit erhöht, während die Funktionalität erhalten bleibt. Gerade so sollen die nachfolgenden Stufen (Typ-Ceck, Byte-code) entlastet werden.