Naturel - Sprache und Compiler - Dokumentation

Andreas Textor und Ralph Erdt
(Compilerbau WS07/08)

Februar 2008

Autoren: Ralph Erdt und Andreas Textor

Wir erklären hiermit, dass das vorliegende Programm ausschließlich von Ralph Erdt und Andreas Textor und ausschließlich unter der Nutzung der erlaubten Hilfsmittel erstellt wurde.

1 Die Sprache Naturel

Naturel steht für „*N*aturel ist die *A*ndreas *T*extor *U*nd *R*alph *E*rdt *L*anguage“. Die Sprache ist objektorientiert und orientiert sich mit der Syntax an C, Java, Pascal und UML. Bei der Syntax wurde auf für die Semantik nicht notwendige Schlüsselwörter wie z.B. `class` verzichtet, andere Schlüsselwörter werden durch Symbole dargestellt, wie z.B. `+` statt `public`. Kommentare werden wie in Java und C++ üblich durch `//` bzw. `/* ... */` notiert. Überschreiben von Methoden ist erlaubt in Naturel, überladen ist nicht erlaubt.

1.1 Syntax

In Naturel gilt für Bezeichner die selbe Nomenklatur wie in C, es gibt keine durch den Compiler vorgeschriebenen Gross-/Kleinschreibungsregeln (Klassen dürfen also z.B. auch mit einem Kleinbuchstaben beginnen). Im folgenden werden zur Klarheit die aus Java bekannten Benennungsregeln verwendet. Jede Anweisung wird durch ein Semikolon abgeschlossen. Wie in Pascal und anderen Sprachen werden die Typen von Variablen mit einem Doppelpunkt getrennt hinter den Variablennamen geschrieben. Zuweisung geschieht über den `:=`-Operator, Vergleich durch den `=`-Operator. Methoden und Attribute haben die Sichtbarkeiten public, private und protected, die durch die UML-Modifier `+`, `-` und `#` ausgedrückt werden. Eine Klassenmethode oder -variable hat statt des in Java verwendeten `static` ein doppeltes Modifierzeichen.

```
1 +a:num;  // public
2 -b:num;  // private
3 #c:num;  // protected
4 ++d:num; // public static
5 --e:num; // private static
6 ##f:num; // protected static
```

Um auf Methoden und Attribute einer Klasse oder eines Objektes zuzugreifen, wird die aus Java bekannte Punkt-Notation verwendet:

```
1 Klasse.klassenMethode();
2 objekt.methode();
3 x:str := objekt.getWert().tostr();
```

1.2 Module und Packages

Jedes Modul (d.h. Quellcodedatei) beginnt optional mit einer Liste von `use`-Anweisungen (entspricht Javas `import` und kann danach beliebig viele Klassen enthalten. Die Paket-Struktur und -Syntax ist ebenfalls an Java angelehnt:

```
1 use paket.unterpaket.Klasse;
2 use paket.unterpaket.Klasse2;
```

Der voll qualifizierte Name einer Klasse (also Klassenname mit vollem Paket-Pfad) darf im Quellcode nicht verwendet werden, lediglich der Klassenname. Sollen mehrere Klassen mit dem selben Namen aus unterschiedlichen Paketen verwendet werden, so wird dazu eine lokale Umbenennung der Klasse mittels `as` verwendet:

```
1 use paket1.Klasse;
2 use paket2.Klasse as KlasseAusPaket2;
```

1.3 Klassen

Klassen in Naturel haben genau eine Oberklasse (Einfachvererbung) und können beliebig viele Interfaces implementieren, wie in Java. Da weder das Schlüsselwort `class` noch ein vergleichbares vorhanden ist, folgt auf den Modifier der Klasse direkt der Name. An der Stelle der Klassendefinition wird auch der Paket-Pfad angegeben, in dem sich die Klasse befindet. Eine Klassendefinition kann folgendermaßen aussehen:

```
1 +paket.unterpaket.Klasse < Oberklasse : Interface1,
     Interface2 { ... }
```

Um ein Interface zu schreiben, wird der ++-Modifier verwendet:

```
1 ++Interface { ... }
```

1.4 Methoden

Eine Methode wird definiert durch einen Modifier, den Methodennamen, eine Liste von Parametern und einen optionalen Rückgabetyp:

```
1 +methodeOhneRueckgabe(a:num, b:num) { ... }
2 +methodeMitRueckgabe(a:num, b:num):num { return(0); }
3 ++statischeMethode() { ... }
```

Eine spezielle Methode ist der Konstruktor einer Klasse. Der Konstruktor wird durch die öffentliche statische Methode `new` ausgedrückt (Ein Rückgabetyp wird nicht explizit angegeben, ebenso kein `return`).

1.5 Datentypen

Es gibt die eingebauten Datentypen `bool`, `num`, `fnum`, `str` und `Object`. `Object` ist wie in Java die Basisklasse aller Objekte. Obwohl die Schreibweise das nicht vermuten lässt, sind auch bool, num, fnum und str echte Objekte und keine „Basisdatentypen“ wie in Java; sie haben `Object` als Oberklasse. Die Klassen haben folgende Methoden:

- `Object: tostr():str`

- `num: tostr():str, addNum(num):num, subNum(num):num, multNum(num):num, divNum(num):num, modNum(num):num, eq(num):bool, neq(num):bool, lteq(num):bool, lt(num):bool, gteq(num):bool, gt(num):bool`
  Es gibt einen Konstruktor, der ein `num`-Objekt mit einem Ganzzahlliteral initialisiert. Dieser wird nur intern genutzt und wird nicht explizit im Naturel-Programm aufgerufen (hier wird nur das Ganzzahlliteral geschrieben).

- `fnum: analog zu num`

- `str: tostr():str, size():num, append(str):str, asnum():num`
  Es gibt einen Konstruktor, der ein `str`-Objekt mit einem Stringliteral initialisiert. Dieser wird nur intern genutzt und wird nicht explizit im Naturel-Programm aufgerufen (hier wird nur das Stringliteral geschrieben).

Darüber hinaus gibt es einen Listen-Datentyp `list`, von dem nicht explizit Objekte erzeugt werden, sondern der implizit für andere Typen zur Verfügung steht:

```
1 x:num[];
2 y:str[];
```

Für den Listentyp exisitiert ein Konstruktor, der ein Object und eine Liste als Argument bekommt und daraus eine neue Liste erzeugt. Ausserdem besitzt eine Liste die Methoden `tostr():str, append(Object):void`. Die Liste speichert zwar intern nur Objects, die Typsicherheit wird aber dadurch gegeben, dass eine Liste immer nur als „Erweiterung“ eines Types auftritt (d.h. eine `str`-Liste kann keine `num`-Objekte enthalten).

1.6 Operatoren

Es gibt die Operatoren `+ - * / % = := < <= != > >= . ! & ^ |`. Dabei ist `=` der Vergleichsoperator und `:=` der Zuweisungsoperator. Arithmetische Operationen und Vergleichsoperatoren sind nur für Zahlen definiert, allerdings kann der `+`-Operator auch auf `str`-Objekten aufgerufen werden, was einem Aufruf der `append`-Methode entspricht.

1.7 Kontrollstrukturen

Naturel enthält die aus C und Java bekannten Kontrollstrukturen für Verzweigungen `if` und Schleifen `while`:

```
1 if ((2 + 3) = 5) {
2     ...
3 }
4 x:num := 1;
5 while (x < 5) {
```

```
    ...
    x := x + 1;
}
```

1.8 Beispielklasse

```
+Punkt {
    -x:num;
    -y:num;

    +getX():num { return(x); }
    +getY():num { return(y); }
    ++new(px:num, py:num) { x := px; y := py; }

    +addierePunkt(p:Punkt) {
        x := x + p.getX();
        y := y + p.getY();
    }

    +tostr():str {
        return("[" + x.tostr() + "," + y.tostr() + "]");
    }

    ++main(args:str[]):num {
        p1:Punkt := Punkt.new(1, 2);
        p2:Punkt := Punkt.new(3, 4);
        out("Punkt 1: " + p1.tostr() + "\n");
        out("Punkt 2: " + p2.tostr() + "\n");
        p1.addierePunkt(p2);
        out("Punkt 1: " + p1.tostr() + "\n");
        return(0);
    }
}
```

2 Der Compiler

2.1 Umgesetzte Features

- Grammatik und Parser
- Einfach-Vererbung
- Überschreiben von Methoden
- Sichtbarkeiten (public, protected, private)
- Aufrufen von Methoden mit Parametern und Rückgabewerten

- Auswertung von Expressions, auch hintereinandergeschaltete Zugriffe auf Methoden und Attribute über den Punkt-Operator
- Umfangreiche semantische Prüfung des Programmes
- Standardbibliothek mit Datentypen für Object, Zahlen, Strings, Listen; jeweils als Klassen
- Ein- und Ausgabefunktionen in Standardbibliothek
- Eigener Programm-Crash-Handler (Speicherverbrauchs-Statistik falls das kompilierte Programm abstürzt
- Erzeugt gültiges ANSI-C
- Erzeugter Code enthält keine Schleifen mehr (C als besserer Assembler)

2.2 Nicht Umgesetzte Features

- Pakete bzw. Auswertung von `use`
- Tupel-Datentyp (Unterstützung in Grammatik und Teilen des Codegenerators, jedoch nicht bei Operatorenbehandlung, daher nicht nutzbar)
- Eigene Stack-Verwaltung, Garbage-Collector

2.3 Visitors

2.3.1 TypeChecker

Es existieren zwei TypeChecker-Vistors (verschiedene interne Behandlung des Programmes, daher nicht in einer Klasse vereinbar). Diese führen neben der Typprüfung folgende semantische Überprüfungen durch:

- Fehlende Typen in Deklarationen
- Falsche Initialisierung von Klassenvariablen
- Es muss genau eine `main`-Methode vorhanden sein
- Überladen von Methoden (wird nicht unterstützt)
- Unbenannte Methodenparameter
- Doppelte Methodenparameter
- Falscher Typ auf linker Seite von Zuweisung
- Falsche Sichtbarkeit von Interfaces
- Fehlende verwendete Oberklasse
- Fälschlich als Oberklasse verwendete Interfaces
- Fehlendes verwendetes Interface

- Fehlende oder falsch getypte Methodenimplementierung aus Interface in Klasse

Fehlende oder falsche Variablen, Methoden oder Typen werden größtenteils in anderen Visitors oder während der Codegenerierung abgeprüft, um Codeverdoppelung zu vermeiden.

2.3.2 SammleKlassenMethoden

Dieser Visitor sammelt alle Klassen und deren Variablen und Methoden. Da immer wieder Informationen über die Klassenstruktur gebraucht werden, werden diese hier gesammelt. Die ganzen Informationen werden in den Strukturen gespeichert, die in Abbildung 1 gezeigt werden. ClassDef enthält alle wichtigen

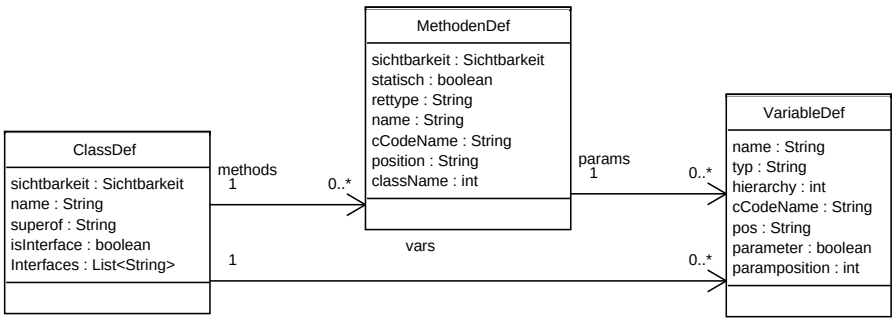

Abbildung 1: Klassendiagramm: SammleKlassenMethoden

Informationen zu einer Klasse. Auch enthält diese eine Liste aller Methoden und Klassenvariablen. MethodenDef enthält die Informationen über die Methoden in der Klasse. Es hat eine Liste der Parameter, was wiederum `VariablenDef`-Objekte sind.

2.3.3 DepthVarManager

Dieses ist ein Helfer-Visitor, von dem einige andere Visitoren erben. Dieser Visitor verwaltet die Klassenliste, die der Visitor `SammleKlassenMethoden` zusammengestellt hat. Auch sammelt der Visitor methodenlokale Variablen auf. Da diese auch blocklokal sein können, wird die Information `hierarchy` mitgeführt: Wenn ein neuer Block geöffnet wird, bekommen alle vorhanden lokalen Variablen plus eins auf `hierarchy`. Wenn ein Block geschlossen wird, bekommen alle lokalen Variablen minus eins auf `hierarchy`. Wenn `hierarchy` nun >= -1 ist, wird diese gelöscht. Dadurch sind zu jedem Zeitpunkt nur die tatsächlich an dieser Stelle verfügbaren Variablen in der Liste. Zudem stellt dieser Visitor Methoden bereit, um Variablen und Methoden zu suchen.

2.3.4 ASTAddThis

In der Sprache soll es möglich sein, Klassenvariablen und Methoden direkt anzusprechen, ohne das this zu verwenden. Allerdings ist es nicht möglich, direkt auf

die Variablen / Methoden zuzugreifen. Daher wird hier überall - wo es notwendig ist - ein this vor solchen Aufrufen generiert.

Beispiel:

```
+Klasse {
    +a:num;
    +methode() {
        a := 5;
        ...
    }
}
```

Dann generiert dieser visitor das:

```
this.a := 5;
```

2.3.5 ASTDefaultConstructor

Dieser Visitor erzeugt einen Default-Konstruktor in Klassen, in denen kein Konstruktor vorhanden ist. Außerdem überprüft er alle Klassendefinitionen auf die Angabe einer Oberklasse. Ist keine Oberklasse angegeben, so wird explizit eine Vererbung von `Object` eingetragen. Zuletzt wird die Initialisierung von Exemplarvariablen aus ihrer Deklaration in den Konstruktor der Klasse verschoben.

2.3.6 ASTLiterals

In diesem Visitor werden alle Zahl- und Stringliterale durch entsprechende Konstruktoraufrufe ersetzt:

```
x:num := 1;
s:str := "Hallo";
```

Daraus wird:

```
x:num := num.new(1);
s:str := str.new("Hallo");
```

Die Konstruktoren befinden sich in der Core-Library und erhalten als Parameter die tatsächlichen Literale (Typen `num_t` bzw. `char*`, siehe `Core.h` und `Core.c`), woraus sie die entsprechenden Objekte konstruieren.

2.3.7 ASTOperators

Alle Operatoren, die im Code verwendet werden, müssen zu Methodenaufrufen der entsprechenden Objeke verwandelt werden, dies passiert in diesem Visitor. Beispiel: Aus folgendem Code:

```
a + 5
s + "\n"
```

wird

```
a.addnum(5)
s.append("\n")
```

In Kombination mit ASTLiterals würde also folgender Ausdruck: [1]

```
x:num;
x := 5 + 3;
```

zu folgendem:

```
x:num;
x := num_new(5).addnum(num_new(3));
```

2.3.8 ASTDotExpr

```
Fibonacci.new(10).getZahl().tostr()
```

Solche Aufrufe werden vom Parser in einen `DOT-Baum` umgewandelt:

```
ADotExp
   AMethodcallExp: Fibonacci [109,63] (0)
   ADotExp
      AMethodcallExp: new [109,73] (1)
         ANumExp: 10 [109,77]
      ADotExp
         AMethodcallExp: getZahl [109,81] (1)
            ANoneExp
         AMethodcallExp: tostr [109,91] (1)
            ANoneExp
```

Im ersten Schritt wird daraus eine flache Liste erstellt:

```
AListExp
   AMethodcallExp: Fibonacci [109,63] (0)
   AMethodcallExp: new [109,73] (1)
      ANumExp: 10 [109,77]
   AMethodcallExp: getZahl [109,81] (1)
      ANoneExp
   AMethodcallExp: tostr [109,91] (1)
      ANoneExp
```

In einem zweiten Schritt werden die Klassenaufrufe identifiziert (hier `Fibonacci`), und mit der folgenden Klassenvariable / Methode zusammen in ein neuen Node gespeichert:

[1] Der hier erzeugte Code wäre aber kein gültiger Naturel-Code, weil die Konstruktoren von num und str nicht explizit aufgerufen werden dürfen.

```
AListExp
   AMethodcalldefExp
      AMethodcallExp: Fibonacci [0,0] (0)
      AMethodcallExp: new [109,73] (1)
         ANumExp: 10 [109,77]
   AMethodcallExp: getZahl [109,81] (1)
      ANoneExp
   AMethodcallExp: tostr [109,91] (1)
      ANoneExp
```

Eine weitere Verarbeitung macht der Visitor `ASTDotListKorr`.

2.3.9 ASTCondKorr

```
if (a=b) {..}
```

In den Bedingungen von den `if`s und `while`s soll nur eine Variable stehen. Dieser Visitor erzeugt eine neue Variable, und baut daraus eine Zuweisung. Und in das `if`/`while` kommt nur noch die neue Variable.

```
x:bool := a=b;
if (x) {..}
```

Bei den `while`s wird zusätzlich noch dafür gesorgt, dass am Ende des while-Blocks die Bedingung neu berechnet wird, indem einfach der Code verdoppelt wird:

```
while (a=b) {
   ...
}
```

wird zu

```
x:bool := a=b;
while (x) {
   ...
   x := a=b;
}
```

2.3.10 ASTDotListKorr

Durch den Visitor `ASTDotExpr` hat man für das Programmfragment

```
Fibonacci.new(10).getZahl().tostr()
```

folgende Liste im AST:

```
AListExp
   AMethodcalldefExp
      AMethodcallExp: Fibonacci [0,0] (0)
      AMethodcallExp: new [109,73] (1)
         ANumExp: 10 [109,77]
   AMethodcallExp: getZahl [109,81] (1)
      ANoneExp
   AMethodcallExp: tostr [109,91] (1)
      ANoneExp
```

Es wird für jeden Methodenaufruf ein `this`-Parameter benötigt. Daher muss diese Liste in einzelne Aufrufe aufgesplittet werden. Dazu werden immer die zwei ersten Knoten der Liste zu einem einzelnen DOT zusammengeführt. Das Ergebnis wird in eine neue Variable gespeichert. Diese Variable wird dann am den Anfang der Liste gehängt. Sollte das erste aber bereits eine Methode sein, so wird diese einzeln ausgeführt, und in eine Variable gespeichert. Dadurch entsteht eine Reihenfolge von Aufrufen:

```
Variable1:Klasse1 := methode();
Variable2:Klasse2 := Variable1.methode2();
Variable3:Klasse3 := Variable2.methode3();
```

Der Beispielcode sieht dann im AST so aus:

```
ADeclarationStatement
   ANamedVariable: !ListCorrFirst_43 [-1,-1]
      ATypeType: str [-1,-1]
         AFalseFlag
      AMethodcalldefExp
         AMethodcallExp: Fibonacci [0,0] (0)
         AMethodcallExp: new [109,73] (1)
            ANumExp: 10 [109,77]
ADeclarationStatement
   ANamedVariable: !ListCorr_44 [-1,-1]
      ATypeType: str [-1,-1]
         AFalseFlag
      ADotExp
         AMethodcallExp: !ListCorrFirst_43 [-1,-1] (0)
         AMethodcallExp: getZahl [109,81] (1)
            ANoneExp
ADeclarationStatement
   ANamedVariable: !ListCorr_45 [-1,-1]
      ATypeType: str [77,9]
         AFalseFlag
      ADotExp
         AMethodcallExp: !ListCorr_44 [-1,-1] (0)
         AMethodcallExp: tostr [109,91] (1)
```

```
24                ANoneExp
```

Der Code, der dem obigen Listing entspricht, wäre dieser: [2]

```
1 !ListCorrFirst_43: str :=  Fibonacci.new(10);
2 !ListCorr_44: str := !ListCorrFirst_43.getZahl();
3 !ListCorr_45: str := !ListCorr_44.tostr();
```

2.3.11 ASTDeckKorr

Der Visitor `ASTDeckKorr` trennt die Zuweisung von den Deklarationen. Aus:

```
1 a:num := 5;
```

wird:

```
1 a:num;
2 a := 5;
```

2.3.12 ASTVarCollect

Der Visitor sammelt alle Variablendeklarationen aus einer Methode auf, und verschiebt sie an den Anfang der Methode, da in ANSI-C zuerst alle Variablen deklariert werden müssen.

2.4 Generatoren

Aus einer Quelldatei `src.naturel` werden eine `.c` und eine `.h`-Datei erzeugt, die den Namen der Klasse tragen, die die `main`-Methode enthält (Eine Quelldatei darf mehr als eine Klasse enthalten). Diese werden anschliessen zusammen mit `Core.c` vom `gcc` übersetzt.

2.4.1 GenHeader

GenHeader erzeugt den Header. Dieser enthält:

- Makroguard
- Core.h-Include
- Typedefs von Klassen-Struct-Pointer: z.B. `typedef struct SKlasse* Klasse;`
- Klassen-Structs mit Exemplarvariablen und Methoden als Funktionspointer
- Prototypen der C-Funktionen, die die Methoden implementieren
- `extern`-Deklaration von Klassenvariablen

[2] Achtung: Interne Variablennamen haben ein „!“ davor, um so Kollisionen mit den Variablennamen aus dem Code zu vermeiden. Im C-Code werden alle Variablennamen umgewandelt.

2.4.2 GenCode

GenCode erzeugt letztendlich aus dem transformierten AST den C-Code. Die erzeugte Datei *Klasse*`.c` enthält:

- Include von *Klasse*`.h`
- Funktionen für Methodenimplementierung und Konstruktoren
- Klassenvariablen als globale Variablen
- Eine Funktion `fillClassVars()`, die von der `main`-Funktion aufgerufen wird und die Klassenvariablen mit den jeweiligen Initialisierungswerten füllt
- Die `main`-Funktion, die ggf. den SIGSEGV-Handler registriert (Betriebssystem-Callback-Funktion, falls das Programm einen Speicherzugriffsfehler verursacht), die Funktion `fillClassVars()` aufruft, eine neue `str`-Liste erzeugt und mit dem Inhalt von `argv` füllt, sowie die main-Methode der Klasse mit der `str`-Liste als Parameter aufruft.

2.5 Aufruf

Der Compiler wird aufgerufen mittels `java naturel.Main` und wird über Kommandozeilenargumente gesteuert. Dabei können folgende Optionen angegeben werden:

- -h oder -help: Gibt eine Meldung über die Nutzung aus, auch die Defaultwerte für die Optionen.
- -version: Gibt die Version des Compilers aus.
- -debug: Annotiert den erzeugten C-Code mit Debug-Kommentaren.
- -v: Verbose: Gibt den AST und andere Informationen beim Kompilieren.
- -c: Wie bei gcc: Kompilieren, aber nicht Linken.
- -o name: Wie bei gcc: Legt Namen der Ausgabedatei fest.
- -sigsegv: Aktiviert den eigenen SIGSEGV-Handler im kompilierten Programm. Achtung: Das Programm kann anschließend nicht mehr mit externen Handlern wie `catchsegv` verwendet werden.
- -conly: Erzeugt C-Code, aber ruft keinen C-Compiler auf.
- -compiler name: Setzt den Compiler, der aufgerufen werden soll, inklusive Parametern; Default: `gcc -g -Wall -pedantic -ansi`.
- -prettyprint: Gibt nur den Formatierten Quellcode aus.
- datei: Der Name der zu kompilierenden `.naturel`-Datei, dieser muss als letzter Parameter angegben werden. Das Verzeichnis, in dem die Datei liegt, muss auch `Core.h` und `Core.c` enthalten.

Beispielaufruf:

```
# Kompilieren ohne Debug-Symbols (kein -g für gcc):
java naturel.Main -v -debug -o zahlenraten -compiler \
"gcc -Wall -pedantic -ansi" \
grammatik/zahlenraten.naturel
```

Ausgabe des Kommandos:

```
(AST)
Writing grammatik/Zahlenraten.h
Writing grammatik/Zahlenraten.c
gcc -Wall -pedantic -ansi -I. -Igrammatik/ -o
   zahlenraten grammatik/Core.c grammatik/Zahlenraten.c
grammatik/Zahlenraten.c: In function 'Zahlenraten_new':
grammatik/Zahlenraten.c:59: warning: passing argument 1
    of '((struct SDingens *)this)->setBlah' from
incompatible pointer type
grammatik/Zahlenraten.c: In function 'initZahlenraten':
grammatik/Zahlenraten.c:67: warning: assignment from
   incompatible pointer type
```

Die „passing argument from incompatible pointer type“ und „assignment from incompatible pointer type“ Meldungen sind normal und können ignoriert werden (Quelle: Zuweisung von Funktionspointern für Unterklassen).