Benchmarken van een DSL implementatie in “Oslo” en C#

Master Thesis

Jeroen van Lieshout

17 september 2009

Master Software Engineering
Universiteit van Amsterdam

Microsoft Nederland B.V.

Docentbegeleider:
Tijs van der Storm

Interne begeleiders:
Dennis Mulder
Jeroen Quakernaat

Versie:
1.0

Samenvatting

In de loop der jaren is veel onderzoek gedaan naar het nut en de noodzaak van Domain Specific Languages (DSL’s). Bij deze onderzoeken heeft men veel gekeken naar de mogelijke valkuilen bij het maken en gebruiken van DSL’s. Waar men echter nog niet zoveel onderzoek naar heeft gedaan, is naar de verschillen tussen het implementeren van DSL implementaties (lexer, parser, checker en interpreter) van scratch (vanilla) of het gebruik van compiler generatoren.

Bij het Centrum Wiskunde en Informatica (CWI) is een onderzoek opgestart waarbij in deelonderzoeken verschillende compiler generatoren worden gebruikt om een implementatie van een DSL, Waebric, te maken. Dit onderzoek vormt een deelonderzoek van dit grotere onderzoek, waarbij een implementatie van Waebric met “Oslo” wordt vergeleken met een implementatie van scratch (vanilla) in C# met als doel inzicht te krijgen in de kwaliteitsverschillen tussen beide implementatiestrategieën.

Om inzicht te krijgen in de kwaliteitsverschillen zijn beide implementaties vergeleken op de aspecten functionaliteit, onderhoudbaarheid, performance en productiviteit. Uit dit onderzoek blijkt dat er geen significant verschil is tussen de inzet van de tool “Oslo” en het van scratch implementeren van een DSL in C#.

Abstract

The last decades a lot of research has been done into the subject of how and why using domain-specific languages (DSL’s). Most research was tailored to possible pitfalls when developing or using DSL’s. A relative uninvestigated area of research is differences between implementing a DSL (lexer, parser, checker and interpreter) from scratch or by using compiler generators.

At the Center for Mathematics and Computer Science (CWI) a research project has been started where in subprojects several compiler generators are used to implement the DSL Waebric. This research is part of this bigger research project, where an implementation of Waebric using Microsoft “Oslo” is compared with a from scratch (vanilla) implementation in C#. The goal of this research is to get insight in differences of quality between both implementation strategies.

To measure quality, both implementations are compared on functionality, maintainability, performance and productivity. This research points out that there’s no significant difference between the usage of “Oslo” and implementing a DSL from scratch with C#.

Voorwoord

Deze thesis is het resultaat van onderzoek dat is uitgevoerd als onderdeel van de eenjarige master Software Engineering aan de Universiteit van Amsterdam.

Ik wil graag Tijs van der Storm bedanken voor de begeleiding en de kritische feedback tijdens het project.

Daarnaast gaat mijn dank uit naar mijn begeleiders Dennis Mulder en Jeroen Quakernaat vanuit Microsoft Nederland die mij tijdens dit project hebben begeleid.

Bovendien wil ik Semantic Designs bedanken voor het gratis ter beschikking stellen van hun tool C# 2.0 Metrics voor het meten van de Halstead Effort.

Tenslotte wil ik mijn ouders bedanken voor de ondersteuning tijdens het project.

Jeroen van Lieshout, Haarlem, september 2009

Inhoudsopgave

1 Introductie

1.1 Aanleiding en context

In de loop der jaren is veel onderzoek gedaan naar het nut en de noodzaak van Domain Specific Languages (DSL’s) (Klint and Deursen 1998; Deursen, Klint et al. 2000; Mernik, Heering et al. 2005). Een DSL is een programmeertaal of specificatietaal, die is gericht op een specifiek (probleem)domein (Deursen, Klint et al. 2000). Voorbeelden van een (probleem)domein zijn verzekeringen, belastingen, documentopmaak, databases, etc. Concrete voorbeelden van DSL’s zijn LaTex, Risla (Arnold, Deursen et al. 1995), SQL en HTML.

Bij deze onderzoeken heeft men ook veel gekeken naar de mogelijke valkuilen bij het maken en gebruiken van DSL’s. Waar men echter nog niet zoveel onderzoek naar heeft gedaan, is naar de verschillen tussen het implementeren van DSL implementaties (lexer, parser, checker en interpreter) van scratch (vanilla) of het gebruik van compiler generatoren (Aho, Lam et al. 2007).

Bij het Centrum Wiskunde en Informatica (CWI) is een onderzoek opgestart waarbij in deelonderzoeken verschillende compiler generatoren worden gebruikt om een implementatie van een DSL, Waebric (van der Storm 2009), te maken. De implementatie bestaat uit het maken van een lexer, parser, checker en interpreter voor Waebric. Het doel hiervan is om kwaliteitsverschillen tussen implementatiestrategieën van DSL implementaties te bepalen. In dit masterproject, dat een deelonderzoek is van het grotere onderzoek, zullen Waebric implementaties in vanilla C# en Microsoft “Oslo” worden vergeleken door middel van benchmarking. De benchmark bestaat uit het vergelijken van een aantal softwarekwaliteitsaspecten (onderhoudbaarheid, efficiëntie en functionaliteit) en het meten van productiviteit.

1.2 Onderzoeksvragen

Binnen dit onderzoek worden twee implementatie strategieën met elkaar vergeleken/gebenchmarked. Één in Microsoft “Oslo” en één in vanilla C#. De hoofdonderzoeksvraag die in dit onderzoek wordt gesteld is:

OV1: Verhoogt de inzet van “Oslo” de kwaliteit van een DSL implementatie?

Om de hoofdonderzoeksvraag te kunnen beantwoorden, moeten de volgende deelvragen worden beantwoord:

OV1.1: Hoe verhoudt de onderhoudbaarheid van de vanilla C# implementatie zich tot die van de “Oslo” implementatie? OV1.2: Hoe verhoudt de performance van de vanilla C# implementatie zich tot die van de “Oslo” implementatie? OV1.3: Hoe verhoudt de functionaliteit van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?

OV1.4: Hoe verhoudt de productiviteit van het ontwikkelen van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?

1.3 Structuur van de thesis

De thesis is als volgt opgebouwd:

In hoofdstuk 2 worden de achtergronden van DSL’s, “Oslo” en Waebric beschreven.

Hoofdstuk 3 beschrijft de onderzoeksmethode. Hier worden de meetmethoden beschreven, die tijdens het experiment zijn toegepast en de bijbehorende hypotheses.

In hoofdstuk 4 worden de implementaties gedetailleerd toegelicht, inclusief ontwerpkeuzes die tijdens de implementatie zijn gemaakt.

Hoofdstuk 5 presenteert de resultaten van de uitgevoerde metingen.

In hoofdstuk 6 worden de resultaten geëvalueerd en wordt de validatie van het onderzoek beschreven.

In hoofdstuk 7 worden tenslotte de onderzoeksvragen beantwoord in de vorm van een conclusie en wordt mogelijk toekomstig onderzoek beschreven.

2 Achtergrond

2.1 DSL

Een domain-specific language (DSL) is een programmeertaal of uitvoerbare specificatietaal, die door middel van notaties en abstracties expressief vermogen biedt beperkt tot een specifiek probleemdomein (Deursen, Klint et al. 2000).

Er zijn verschillende redenen om DSL’s te gebruiken:

- Productiviteit ten opzichte van het gebruik van een General Purpose Language (GPL) neemt toe, omdat domeinspecifieke notaties en constructies in de taal kunnen worden gebruikt die in een GPL lastiger zijn te realiseren (Mernik, Heering et al. 2005).
- Omdat een DSL het toelaat om programma’s uit te drukken in de abstractie van het probleem domein, kunnen ook domeinexperts het begrijpen, valideren, wijzigen en zelfs zelf in de DSL programma’s ontwikkelen (Deursen, Klint et al. 2000).
- Een DSL voor een goed gekozen domein en met goede tools ontwikkeld kan de kosten voor het bouwen van nieuwe applicaties en het onderhouden van bestaande applicaties drastisch verminderen (Klint and Deursen 1998).

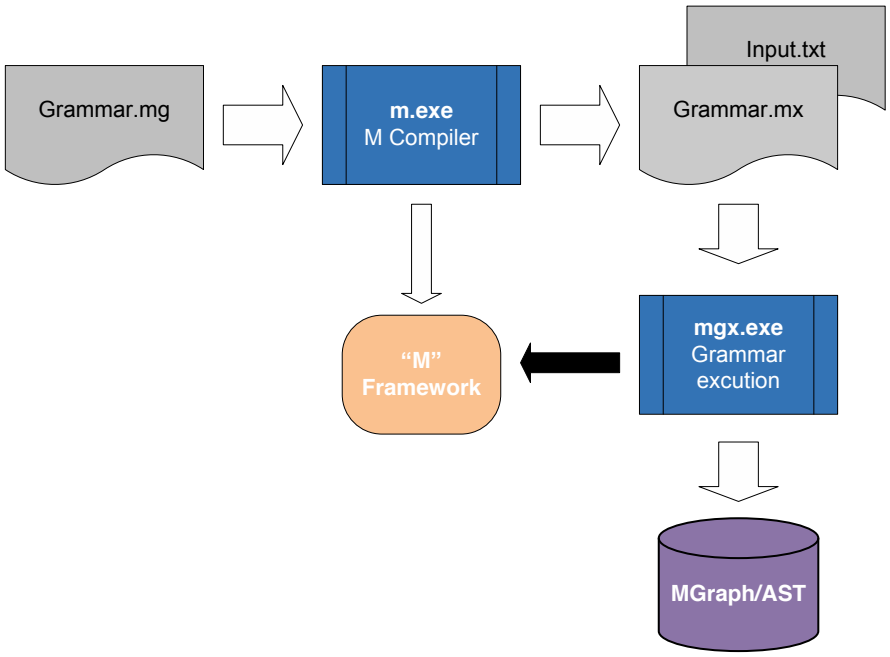
Er zijn ook nadelen verbonden aan het gebruik van DSL’s:

- De DSL moet eerst worden geïmplementeerd en hiervoor zijn mensen nodig met domeinkennis en kennis van het ontwikkelen van talen (Mernik, Heering et al. 2005). Dit is lastig omdat er weinig mensen zijn die kennis hebben van beide gebieden.
- De kosten voor het ontwerpen, implementeren en onderhouden van de DSL (Deursen, Klint et al. 2000).
- De moeilijkheid om een balans te vinden tussen GPL constructies en domein specifieke constructies (Deursen, Klint et al. 2000).

Het ontwikkelproces van DSL’s kan worden gefaciliteerd door het gebruik van een language development systeem of toolkit (Mernik, Heering et al. 2005). In dit onderzoek wordt gebruik gemaakt van Microsoft “Oslo” als language development systeem voor het implementeren van de DSL Waebric.

2.2 Microsoft “Oslo”

Momenteel is Microsoft bezig met het ontwikkelen van een nieuw modelleerplatform genaamd “Oslo” . “Oslo” is een modelleerplatform bestaande uit een aantal onderdelen, namelijk een modelleertaal met bijbehorend framework, genaamd “M” (MSDN 2009), een opslagplaats voor modellen, de Repository, en een visualisatietool Quadrant (Hermans 2009). De modelleertaal “M” bestaat uit een subtaal, genaamd MGrammar, die wordt gebruikt om een grammatica voor een DSL te specificeren. Met behulp van een MGrammar specificatie kan een stuk tekst worden geparseerd en worden omgezet in een MGraph. Een MGraph is de definitie voor een abstracte syntaxboom (AST) in “Oslo”. In Figuur 1 is schematisch het proces van specificatie een grammatica tot MGraph weergeven.

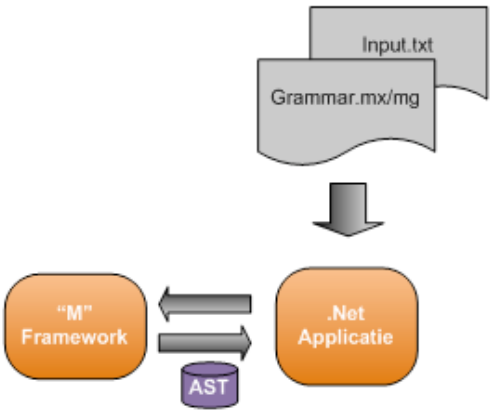


Figuur 1: Proces in “Oslo”: van specificatie van een grammatica tot AST

Zoals in Figuur 1 weergegeven, begint het proces met het ontwikkelen van een grammatica specificatie in MGrammar (.mg bestand). Door middel van de “M” compiler (m.exe) wordt deze grammatica gecompileerd tot een image (.mx bestand). Dit image bestaat uit een xml representatie van de grammatica. De compiler maakt hiervoor gebruik van het “M” Framework, een framework die een API biedt om grammatica’s geschreven in MGrammar te compileren en om gecompileerde grammatica’s te gebruiken om tijdens runtime parsers te creëren voor deze grammatica’s.

Met behulp van het gecompileerde image kan door middel van de MGrammar Executor (mgx.exe) een willekeurig bestand worden geparseerd volgens de grammatica. Hiervoor wordt opnieuw gebruik gemaakt van het “M” Framework. Het resultaat is, wanneer het parseren correct verloopt (input voldoet aan specificatie grammatica), een tekstuele AST. Eventueel worden foutmeldingen getoond wanneer de input niet voldoet aan de specificatie van de grammatica.

Het “M” Framework is ook te gebruiken vanuit .NET applicaties. In Figuur 2 is dit proces schematisch weergeven.



Figuur 2: Gebruik “M” Framework in .NET

Bij het gebruik van het “M” Framework moet door de .NET applicatie eerst de grammatica worden ingelezen. Dit kan zowel voorgecompileerde grammatica als MGrammar (niet voorgecompileerde grammatica) zijn. Door gebruik van het “M” Framework kan deze grammatica eventueel nog worden gecompileerd, wanneer dit nog niet is gebeurd. Daarna kan het “M” Framework worden aangeroepen om een parser instantie voor deze grammatica te creëren. Met deze instantie kan vanuit de .NET applicatie rechtstreeks input worden ingelezen en worden geparseerd. De AST kan vervolgens in de vorm van node objecten in de .NET applicatie worden gebruikt. In hoofdstuk 4 wordt dit proces uitgebreid beschreven.

In het experiment zal alleen gebruik worden gemaakt van MGrammar het bijbehorende “M” framework om MGraph’s/AST’s te produceren. De overige onderdelen van “Oslo” zijn niet specifiek gericht op het maken van textuele DSL’s en vallen daarom buiten de scope van dit onderzoek. Omdat “Oslo” enkel voorziet in een parser, zal de checker en interpreter gemaakt worden in C# met gebruik van het “M” Framework om de parser te kunnen gebruiken. Omdat “Oslo” een product in ontwikkeling is en per kwartaal een nieuwe versie wordt uitgebracht, zal tijdens het experiment alleen de publieke testversie uit mei 2009 worden gebruikt.

2.3 Waebric

Waebric is een taal bedoeld voor het genereren van XHTML opmaak (van der Storm 2009). De taal biedt dezelfde mogelijkheden als XHTML, maar met als toevoeging o.a. functiedeclaraties, beperkte control flow (if, if/else, each, etc), variabelen (lijsten, records, naturals, strings), built-in statements (echo, comment, cdata).

Hieronder volgt een concreet voorbeeld van de Waebric syntax:

```
module homepage

site
    index.html: home("Hello World!")
end

def home(msg)
    html {
        head title msg;
        body echo msg;
    }
end
```

Bovenstaand voorbeeld bevat een Waebric module, met daarbinnen een site definitie en een functiedefinitie. De site definitie is bedoeld om aan te geven welke XHTML bestanden moeten worden gegenereerd en welke functie daarvoor moet worden uitgevoerd. In dit geval moet index.html worden gegenereerd door de functie home aan te roepen met de parameter “Hello World!”. De index.html ziet er als volgt uit:

```
<html>
      <head>
            <title>Hello World!</title>
      </head>
      <body>Hello World!</body>
</html>
```

De structuur van de home functie uit het Waebric voorbeeld is goed te herkennen. De volgorde van html, head, title en body uit het Waebric voorbeeld komt overeen met de output. Bij de title tag is de msg variabele vervangen door de waarde van de variabele, namelijk *Hello World!*. Op de plaatsen waar de variabele msg stond is nu de daadwerkelijke waarde *Hello World!* voor in de plaats gekomen. Dit gebeurt op twee manieren in het voorbeeld. De eerste manier is de expressie gebruiken als inhoud voor een markup tag, in dit geval de title tag. De tweede manier is gebruik maken van de built-in operator echo om de expressie weg te schrijven als tekst.

Binnen Waebric zijn er een aantal typen statements voor control flow:

- If/Else statement:
  Met een if/else statement kan aan de hand van een predicaat (voorwaarde) een stuk code wel of niet worden uitgevoerd.
- Each statements:
  Met een each statement kunnen lijsten worden doorlopen.
- Let statements:
  Met een let statement kunnen lokale scopes (deel waarin variabelen en functies zichtbaar zijn) worden gedefinieerd met daarbinnen variabelen- en functiedeclaraties.

3 Onderzoeksmethode

Het onderzoek bestaat uit het vergelijken van twee implementatiestrategieën voor het implementeren van een DSL. Om dit te onderzoeken wordt een experiment uitgevoerd waarbij twee Waebric implementaties worden gemaakt (een vanilla C# implementatie en een met behulp van “Oslo”). Vervolgens vindt de vergelijking plaats voor de volgende attributen:

- Functionaliteit
- Onderhoudbaarheid
- Performance
- Productiviteit (ontwikkeltijd)

Per attribuut zal worden toegelicht welke metingen worden uitgevoerd om dit te kunnen vergelijken en wat met de metingen kan worden bepaald. Tevens zal worden aangegeven van welke hypothese wordt uitgegaan.

3.1 Functionaliteit

Met functionaliteit wordt aangeduid in hoeverre een Waebric implementatie voldoet aan de Waebric specificatie. Om dit te meten is een set van 104 Waebric testprogramma’s opgesteld door het CWI, ook wel acceptatietests genoemd. De acceptatietests testen alle onderdelen van de Waebric specificatie met uitzondering van CData statements en Site elementen. Per implementatie worden alle tests uitgevoerd en wordt gecontroleerd of de output overeenkomt met de output van de referentie-implementatie. Het percentage van de geslaagde acceptatietests drukt de mate van acceptatie uit. Hoe hoger het percentage van acceptatie, hoe beter de specificatie is nageleefd. Een lagere acceptatie kan duiden op notaties of structuren die moeilijk zijn uit te drukken in de gekozen implementatiestrategie.

Hypothese

De hypothese is dat zowel met behulp van “Oslo” als in vanilla C# de volledige Waebric specificatie kan worden gerealiseerd. Deze hypothese komt tot stand door het feit dat de taal “M” (de specificatietaal in “Oslo”) afgeleid is van EBNF (Garshol 2003). De specificatie van Waebric is vastgelegd in SDF (Heering, Hendriks et al. 1989), welke lijkt op EBNF. Daarom moet het mogelijk zijn om deze specificatie ook in “M” uit te kunnen drukken. De vanilla implementatie wordt in C# ontwikkeld, wat een taal is die is bedoeld voor generieke doeleinden, ook wel een general purpose language (GPL) genoemd. Deze bevindt zich op een lager abstractieniveau dan EBNF, waardoor dit geen beperking moet vormen voor het implementeren van de Waebric specificatie.

3.2 Onderhoudbaarheid

De onderhoudbaarheid zegt iets over hoe gemakkelijk het is om software aan te passen aan een gewijzigde omgeving of om fouten en/of performance te verbeteren (IEEE 1993). DSL´s evolueren in de tijd. Daarom is het belangrijk iets te weten over de onderhoudbaarheid van de lexer, parser, checker en interpreter per implementatiestrategie.

In tabel 1 zijn de metrieken voor onderhoudbaarheid weergegeven met de desbetreffende afkorting die in dit onderzoek worden gemeten. Per metriek zal worden toegelicht hoe deze wordt berekend en wat deze uitdrukt.

Afkorting	Metriek
NCLOC	Aantal regels code zonder commentaar en layout
MLOC	Aantal regels code in methodes
MCC	McCabe Complexity
E	Halstead Effort
MI	Maintainability Index
#M	Aantal methoden

Tabel 1: Metrieken onderhoudbaarheid

De metrieken zoals weergegeven in Tabel 1 zullen voor grammatica geschreven in MGrammar op een andere manier worden berekend dan voor C# code. Omdat MGrammar geen GPL is in tegenstelling tot C#, zullen deze metrieken voor grammatica worden berekend volgens de metrieken suite voor grammatica van (Power and Malloy 2004).

NCLOC

NCLOC staat voor Non-commented Lines Of Code, wat staat voor het aantal regels code zonder commentaar en layout en geeft hiermee inzicht in de grootte van de codebase. NCLOC geeft geen direct inzicht in de onderhoudbaarheid, maar er is een correlatie tussen NCLOC en onderhoudbaarheid (Rosenberg 1997).

MLOC

MLOC staat voor Method Lines Of Code, wat staat voor het aantal NCLOC dat zich bevindt in methodes. Voor MLOC geldt hetzelfde als voor NCLOC wat betreft indicatie voor onderhoudbaarheid. Deze metriek wordt niet gemeten voor grammatica geschreven in MGrammar, omdat deze taal geen methodes bevat.

McCabe Complexity

MCC staat voor McCabe Complexity (McCabe 1976). Bij MCC wordt het aantal lineair onafhankelijke paden (branches) in de code geteld. Hiermee wordt de complexiteit van de code gemeten. Hoe hoger de complexiteit, hoe moeilijker het is om deze te testen en hoe moeilijker het is om deze te begrijpen.
De MCC van een methode kan worden bepaald door te starten met het getal 1. Vervolgens moet voor iedere branch (if, for, foreach, while, do while, case, &&, || of ?), dit getal met 1 worden opgehoogd.

Onderstaand voorbeeld toont C# code met een McCabe complexiteit van 2:

```
public bool IsGreaterThan(int value1, int value2)
{
	if(value1 > value2)
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

In bovenstaand voorbeeld wordt voor de methode zelf een complexiteit van 1 gerekend, plus 1 voor de if branche wat een totale McCabe Complexity van 2 oplevert.

Bij grammatica kan de MCC worden berekend door te starten met het getal 1. Vervolgens moet bij iedere branch (*, ?, + of |) in de productieregels het getal met 1 worden verhoogd. Onderstaand voorbeeld toont een productieregel in MGrammar met een complexiteit van 2.

```
token Keyword = ”if” | ”else”;
```

In dit voorbeeld moet een complexiteit van 1 worden gerekend voor de productieregel, plus 1 voor de | (or) in de productie, wat een totale McCabe Complexity van 2 oplevert.

Halstead Effort

De Halstead Effort (Halstead 1977) geeft inzicht in de inspanning die moet worden geleverd om de applicatie te ontwikkelen en te onderhouden. Hoe lager de Halstead Effort is, hoe gemakkelijker het is om de applicatie te onderhouden. De Halstead effort wordt berekend volgens de volgende formule:

$$E = \frac{\mu_1 \, \eta_2 \, (\eta_1 + \eta_2) \, \log_2(\mu_1 + \mu_2)}{2\mu_2}$$

Waarbij geldt dat:

μ_1 = aantal unieke operatoren
μ_2 = aantal unieke operanden
η_1 = aantal operatoren
η_2 = aantal operanden

Een voorbeeld om toe te lichten hoe dit wordt berekend aan de hand van een stuk C# code:

```
public int testfunction()
{
	return 1-2;
}
```

In het voorbeeld worden als operatoren geteld: public,(), {}, return, -, in totaal dus 5 operatoren. De operanden bestaan uit: int, testfunction, 1, 2, ;., in totaal dus 5 operanden. In dit voorbeeld komen geen dubbele operanden of operatoren voor, dus het aantal unieke operatoren en het aantal unieke operanden is in dit geval ook 5. De Halstead effort kan op basis van deze waarden worden uitgerekend en is in dit geval 83,04.

Het bepalen van de Halstead Effort voor MGrammar, gaat op dezelfde wijze als bij code die is geschreven in een General Purpose Language (GPL), zoals C#. Alleen moeten de parameters anders worden uitgerekend:

μ_1 = aantal unieke operatoren
μ_2 = aantal unieke terminals + aantal unieke non-terminals
η_1 = aantal operatoren
η_2 = aantal terminals + aantal non-terminals

Als operatoren worden meegeteld (+,*,|,?,..,^). Opnieuw een voorbeeld om toe te lichten hoe dit wordt berekend bij MGrammar:

```
token Digit = '0'..'9';
token Numeric = Digit+;
syntax Main = Numeric;
```

In dit voorbeeld zijn drie producties. Een productie is een regel om enkele symbolen te herschrijven naar andere symbolen. Iedere productie is een Non-Terminal, met daarin Terminals en Non-Terminals. Er zijn 2 Terminals, namelijk 0 en 9. Terminals zijn elementen die niet kunnen worden herschreven door een productie. In dit voorbeeld tellen we in totaal 5 Non-Terminals, namelijk Digit, Numeric, Digit, Main en Numeric. Vervolgens moeten de operatoren worden geteld, in dit geval 2, namelijk .. en +. Nu moeten de parameters worden ingevuld volgens de eerder gegeven definitie:

μ_1 = 2 (.. en +)
μ_2 = 5 (2 unieke Terminals + 3 unieke Non-Terminals (aantal producties))
η_1 = 2 (.. en +)
η_2 = 7 (2 Terminals + 5 Non-Terminals)
Bij μ_2 wordt voor het aantal unieke Non-Terminals het aantal producties gebruikt, omdat iedere productie een Non-Terminal is.

Met deze parameters kan de Halstead Effort worden uitgerekend, in dit geval is deze 35,37.

Maintainability Index

MI staat voor Maintainability Index en geeft een goede indicatie in hoeverre een stuk software snel en gemakkelijk kan worden onderhouden (Oman and Hagemeister 1994). Hoe hoger de MI, hoe beter de onderhoudbaarheid is. De MI wordt berekend volgens de volgende formule (Welker 2001), verder MI3 genoemd:

$$MI3 = 171 - 5.2 * \ln(aveV) - 0.23 * aveV(g') - 16.2 * \ln(aveLOC)$$

Waarbij geldt dat:

$aveV = \frac{\eta_1+\eta_2 \log_2(\mu_1+\mu_2)}{Aantal\ \ met\ hoden} = \ gemiddelde\ Halstead\ Volume\ per\ methode$
$aveV(g') = gemiddelde\ MCC\ per\ methode$
$aveLOC = gemiddelde\ NCLOC\ per\ methode$

Bij het berekenen van de MI3 voor grammatica is de gebruikte formule hetzelfde, alleen hebben de parameters een andere betekenis:

$aveV = \frac{\eta_1+\eta_2 \log_2(\mu_1+\mu_2)}{Aantal\ \ Producties} = gemiddelde\ Halstead\ Volume\ per\ productie$
$aveV(g') = gemiddelde\ MCC\ per\ productie$
$aveLOC = \frac{(Terminals\ \ +Non\ Terminals\ )}{Aantal\ \ producties} = gemiddelde\ grootte\ per\ productie$

Hypothese

De hypothese is dat de Waebric implementatie in “Oslo” beter onderhoudbaar is dan de vanilla C# implementatie. Bij het gebruik van “Oslo” hoeft de lexer en de parser zelf niet te worden ontwikkeld, maar enkel de specificatie van de grammatica moet worden ontwikkeld. De grammatica is in tegenstelling tot de vanilla C# implementatie geschreven in een DSL, namelijk MGrammar. Code geschreven in een DSL kan onder voorwaarde van een goed gekozen domein, beter onderhoudbaar zijn dan code die is geschreven in een GPL, omdat de DSL zich op het juiste abstractieniveau bevindt (Klint and Deursen 1998).

De checker en interpreter worden in beide implementaties in C# ontwikkeld. Deze zullen qua onderhoudbaarheid in beide implementaties vergelijkbaar zijn, omdat deze dezelfde taak hebben en beiden geschreven zijn in dezelfde GPL.

3.3 Performance

Met performance wordt aangeduid hoe lang de executietijd is in milliseconden (ms) om een drietal Waebric testbestanden om te zetten naar XHTML. Deze bestanden verschillen qua grootte. Dit wordt voor beide implementatiestrategieën gemeten. Een hogere executietijd betekent dat het langer duurt om een Waebric bestand te parseren, checken en te interpreteren en levert dus een slechter resultaat op. De performance wordt gemeten per component (parser, checker en interpreter) om een nauwkeurig overzicht te krijgen in de verschillen en mogelijke bottlenecks. Iedere test zal 1000 keer worden uitgevoerd en de gemiddelde tijd per component zal worden gemeten.

Hypothese

De hypothese is dat de vanilla C# implementatie een betere performance (lagere executietijd) heeft dan de “Oslo” implementatie. Bij het gebruik van het “M” framework moet tijdens runtime de specificatie worden ingelezen en een parser instantie worden gecreëerd, voordat daadwerkelijk met het parseren wordt begonnen. De vanilla C# implementatie kent deze opstarttijd niet en kan onmiddellijk starten met het parseren. De “Oslo” parser maakt gebruik van het GLR parseeralgoritme en de vanilla C# parser van het LL parseeralgoritme. GLR parsers zijn noodzakelijkerwijs minder efficiënt dan LL parsers (Tomita 1991). De checker en interpreter zullen van de vanilla C# implementatie efficiënter zijn dan die van de “Oslo” implementatie. Dit omdat de “Oslo” checker en parser alleen door middel van loops de AST structuur kunnen doorlopen. Dit levert extra complexiteit op wat ten koste gaat van de efficiëntie.

3.4 Productiviteit

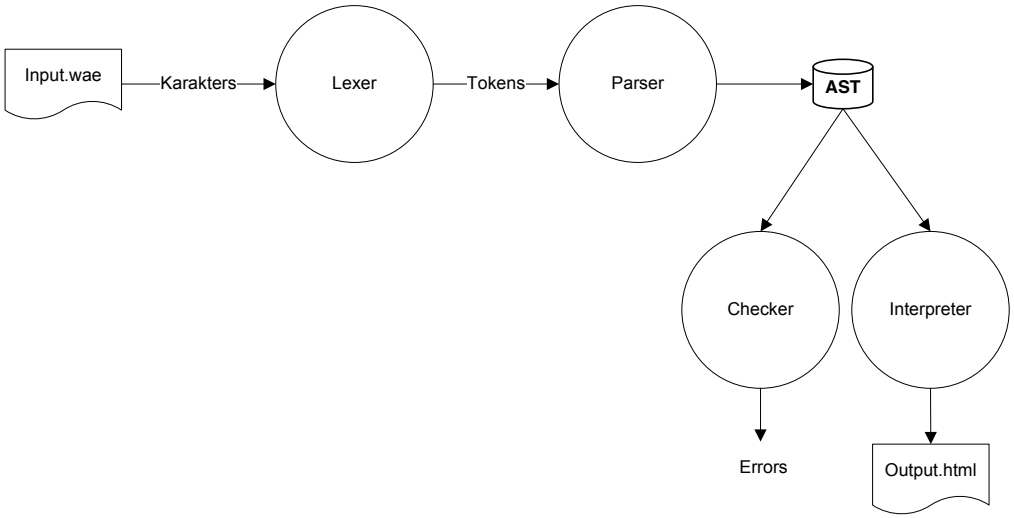
De productiviteit wordt gemeten door per component, per implementatiestrategie de ontwikkeltijd te meten in minuten. Een hogere productiviteit is een indicator dat de implementatiestrategie leidt tot een kortere ontwikkeltijd van de lexer, parser, checker en interpreter.

Hypothese

De hypothese is dat de productiviteit van de ontwikkeling van lexer, parser, checker en interpreter bij de ontwikkeling van de “Oslo” implementatie hoger is dan bij de implementatie van vanilla C#. Uit onderzoek is gebleken dat het gebruik van DSL’s een hogere productiviteit oplevert, dan bij inzet van GPL’s (Kieburtz, Bell et al. 1996). Aangezien in “Oslo” gebruik wordt gemaakt van de DSL “M”, zal dit naar verwachting ook hier gelden.

4 Implementatie

Het experiment bestaat uit het implementeren van twee Waebric implementaties. De Waebric implementaties bestaan uit een lexer, parser, checker en interpreter voor de taal Waebric. In Figuur 3 is het totale proces weergeven met de eerdergenoemde componenten.



Figuur 3: Proces: van Waebric input tot XHTML output

Het proces start met het inlezen van een bestand met daarin code geschreven in Waebric (.wae bestand). Deze worden karakter voor karakter ingelezen door de lexer en omgezet in tokens. Dit zijn gecategoriseerde verzamelingen van karakters (identifiers, naturals, keywords, whitespace, etc). Vervolgens leest de parser deze tokens in en haalt hier de essentiële informatie uit en creëert hiervan een boomstructuur in de vorm van een AST. Deze AST wordt door de checker gebruikt voor semantische controles en geeft eventueel een overzicht van gevonden fouten. De interpreter interpreteert de AST en maakt hiervan een XHTML boomstructuur, die wordt weggeschreven in een HTML bestand.

Per component zal worden beschreven hoe deze in beide implementaties werken en waarom deze zo werken.

4.1 Lexer

Een lexer, ook wel scanners of tokenizers genoemd, zet een stroom van karakters om in een stroom van tokens (Aho, Lam et al. 2007). Een token is een geclassificeerde verzameling van karakters. Hieronder een voorbeeld van tokens die in de taal Waebric voorkomen:

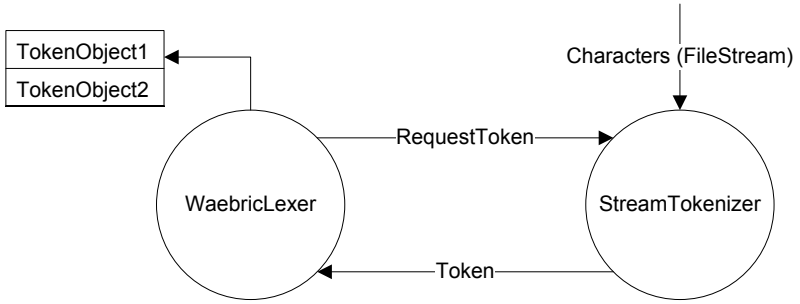
```
10
if
//commentaar token
```

In bovenstaand voorbeeld komen drie type tokens voor, namelijk een natural (numerieke waarde), een keyword (if is een Waebric keyword) en een single line commentaarblok.

Het gebruik van een lexer verkleint de complexiteit van een parser, omdat een parser geen rekening meer hoeft te houden met layout of commentaar. Daarnaast is het gemakkelijker om de lexer te vervangen voor een andere, voor bijvoorbeeld een ander formaat input (Aho, Lam et al. 2007).

Vanilla C# Lexer

De vanilla C# lexer leest een Waebric bestand karakter voor karakter in en categoriseert deze in tokens ook wel tokenizen genoemd. In Figuur 4 is schematisch het proces binnen de vanilla C# lexer weergegeven:



Figuur 4: Proces vanilla C# lexer

De vanilla C# lexer bestaat uit twee componenten, de WaebricLexer en een StreamTokenizer. De StreamTokenizer leest een bestand in (via een FileStream) en zet de karakters hieruit om in globale tokens (layout, comment, tekst, numeriek en symbolen). De WaebricLexer verzamelt tokens van de StreamTokenizer en maakt hier Waebric specifieke tokens van (keyword, identifier, number, text, symbol, waebricsymbol en embedding) in de vorm van een tokenobject. Deze tokenobjecten bestaan uit de waarde van het token (verzameling karakters), het type en de positie in het orginele bestand. Deze tokenobjecten worden opgeslagen in een lijst, zodat deze later kunnen worden gebruikt door de parser. De werking wordt aan de hand van een voorbeeld van Waebric input toegelicht:

```
module test
```

De WaebricLexer roept de StreamTokenizer aan om dit Waebric bestand in te lezen en om het eerste token te verzamelen. De StreamTokenizer leest een karakter in (in dit voorbeeld de letter `m`) en bepaalt op basis hiervan het type token. In dit geval betreft het een letter en is dus het tokentype tekst. Vervolgens worden één voor één karakters ingelezen, zo lang deze van hetzelfde type token zijn en wordt hiervan één token gemaakt. In dit geval bestaat dit token uit de karakterreeks `module`. Dit token wordt doorgestuurd naar de WaebricLexer.
De WaebricLexer gaat kijken naar het token om het Waebric specifieke type te bepalen. Voor tekst kan dit een keyword of een identifier zijn. In dit geval betreft het een keyword, namelijk module. Vervolgens maakt de WaebricLexer een tokenobject en voegt deze toe aan de lijst. Het proces herhaalt zich nu opnieuw.

Opnieuw wordt een token gevraagd aan de StreamTokenizer en weer wordt een karakter ingelezen. Deze keer betreft het een spatie, wat een layout token is. Vervolgens wordt het volgende karakter

ingelezen en nu wordt een letter gevonden, namelijk een `t`. Dit is geen layout token en daarom wordt nu enkel de spatie als token gezien en geretourneerd aan de WaebricLexer. De WaebricLexer filtert layout en comments, omdat deze voor het parseren niet van belang zijn en daarom wordt dit token genegeerd en wordt opnieuw een token gevraagd aan de StreamTokenizer. Dit proces herhaalt zich totdat het einde van het bestand is bereikt, of wanneer een onbekend type token wordt gevonden. In dit laatste geval wordt een foutmelding gegenereerd.

De reden waarom de WaebricLexer tokens vraagt is vanwege het feit dat bij een Text token, vooruit moet worden gekeken om het token type te bepalen. Een voorbeeld om dit toe te lichten:

```
”dit is tekst gescheiden door spaties, maar toch een Waebric token”
```

Bovenstaande tekst is een Text token. Text begint altijd met een quote en eindigt altijd met een quote. De StreamTokenizer ziet deze tekst niet als één token, maar als meerdere tokens (symbol, tekst, layout, tekst, layout, etc.). De WaebricLexer verzamelt na het detecteren van een quote alle token types, die in een Text token mogen voorkomen tot weer een quote wordt gevonden. Bij het detecteren van een niet toegestaan token of een onverwacht einde van het bestand resulteert dit in een foutmelding.

In enkele gevallen (escape karakters, comment statements en commentaar) komt het voor dat alleen vooruit kijken niet voldoende is om te bepalen welk token wordt gematched. Hieronder een voorbeeld om dit toe te lichten:

```
”//”
//
```

Bovenstaande karakterreeksen hebben een verschillende betekenis. Het eerste token is een Text token met twee slashes tussen quotes. Het tweede token is commentaar. Om te voorkomen dat de twee slashes tussen quotes als commentaar worden gezien, wordt bij het matchen van een Text token tijdelijk de detectie van commentaar uitgeschakeld.

Embeddings in vanilla C#

Waebric biedt ondersteuning voor zogenaamde embeddings. Dit zijn stukken tekst tussen quotes waartussen html markup en expressies kunnen worden gebruikt. Om het tijdens het parseren onderscheid te kunnen maken tussen Text en Embeddings is een apart EmbeddingToken gemaakt. Wanneer een embedding wordt gevonden tijdens het tokenizen, dan worden alle tokens verzameld tot aan het einde van de embedding. Vervolgens wordt dat geheel opnieuw getokenized en opgeslagen in een aparte lijst van tokens binnen een EmbeddingToken. De parser kan hierdoor de embedding als geheel token verwerken en hoeft dit niet speciaal meer te detecteren. In onderstaand voorbeeld is een voorbeeld van een embedding weergegeven:

```
”dit is tekst voor het em element<em ”tekst in het em element”>deze tekst
komt na het em element”
```

Bovenstaand voorbeeld wordt als een token getokenized met daarbinnen Text, Symbool (<), Identifier, Text, Symbool (>) en Text. De parser kan dit in een aparte lijst met tokens doorlopen.

“Oslo”

In “Oslo” wordt in MGrammar (Microsoft 2009) met behulp van token regels gedefinieerd welke tokens er zijn en uit welke karakters deze bestaan. In MGrammar kan men als volgt tokens definiëren:

```
token NatCon = ('0'..'9')+;
token IdCon = ('a'..'z' | 'A'..'Z')+;
token Space = '\u0020';
```

De token regels bestaan altijd uit het token keyword gevolgd door de naam van het token en de beschrijving na de = operator. In het voorbeeld worden drie type tokens gedefinieerd. Het token NatCon moet in dit geval altijd bestaan uit de cijfers 0 t/m 9 (range van karakters aangegeven met .. operator) en deze moeten één of meerdere keren achter elkaar voorkomen (aangegeven met + operator). Het token IdCon moet altijd bestaan uit de letters a t/m z of de letters A t/m Z en deze moeten eveneens één of meerdere keren achter elkaar voorkomen. Het derde token is het Space token. Deze moet in dit geval altijd bestaan uit het een spatie (aangegeven met een unicode notatie).

De “Oslo” lexer tokenized de input aan de hand van alle gedefinieerde token regels. De syntax regels wordt gebruik door de parser, zie voor een beschrijving hiervan paragraaf 4.2.

In een taal zoals Waebric komt het voor dat karakters in een bepaalde context een andere betekenis hebben. Zo kan de karakterreeks `”text”` een Text token zijn, maar in het geval dat het woord `comment` gescheiden door spaties ervoor staat betreft het een StrCon token.

Om dit onderscheid te kunnen maken zullen de tokens er als volgt uit moeten zien:

```
token Text = '"' t:(EscQuote | Text_Char)* '"' => t;
token StrCon = '"' s:Str_Char* '"' => s;
token CommentKey = "comment" Spaces t:StrCon Spaces ";" => t;
```

Door het definiëren van twee losse tokens die verschillend zijn kan de `comment ”text”` karakterreeks niet worden gematched met een Text token, maar enkel met een CommentKey. Een nadeel van deze oplossing is dat er nog rekening moet worden gehouden met layout karakters. Een andere oplossing is in deze versie van “Oslo” niet mogelijk, omdat de lexer onafhankelijk van de parser werkt. Zie ook “Samenwerking tussen Oslo lexer en parser” in paragraaf 4.2, voor een nadere beschrijving van de oorzaak van dit probleem.

4.2 Parser

Een parser leest de stroom van tokens die uit de lexer komen in en controleert deze qua syntax op correctheid. Wanneer fouten worden gevonden, dan wordt dit gemeld zodat de gebruiker weet wat fout is en waar. Wanneer de stroom van tokens qua syntax correct zijn, wordt een boomstructuur gemaakt van de relevante tokens. Dit wordt ook wel de abstracte syntax boom (AST) genoemd. De AST kan vervolgens voor andere doeleinden worden gebruikt, zoals semantiek controleren, interpreteren of compileren.

LL(k) parseren (vanilla C#)

De vanilla C# implementatie maakt gebruik van het zogenaamde LL(k) parseeralgoritme. LL staat voor Left-to-right en meest linkse afleiding, waarbij gebruik wordt gemaakt van een top-down benadering (beginnen bij het startpunt van de grammatica) (Aho, Lam et al. 2007). (k) staat voor het k aantal tokens dat vooruit wordt gelezen. Dit algoritme is gekozen, omdat deze relatief eenvoudig handmatig is te implementeren ten opzichte van algoritmen zoals LR en GLR (Aho, Lam et al. 2007). Om de werking toe te lichten een klein voorbeeld aan de hand van een grammatica:

(1) S → S + S
(2) S → a
(3) S → b

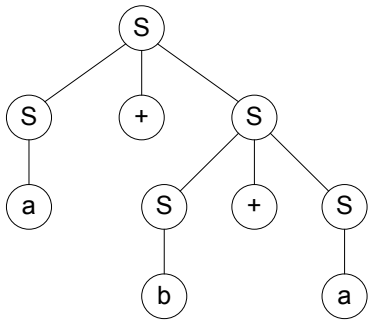
Bij het LL parseren gelden de volgende regels:

- Start bij het start symbool van de grammatica
- Vervang de meest linkse non-terminal als eerste

Als we deze regels toepassen op de string “a + b + a” en deze parseren volgens het LL algoritme dan gaat dit als volgt:

S → S + S (1)
→ a + S (2)
→ a + S + S (1)
→ a + b + S (3)
→ a + b + a (2)

In Figuur 5 is de bijbehorende parse boom weergeven.



Figuur 5: Parse boom van “a + b + a”

Als we kijken naar de manier van parseren dan zien we dus dat er van links naar rechts wordt gewerkt en dat steeds de meest linkse non-terminal wordt afgeleid.

Left Recursion

Bij het gebruik van het LL algoritme kan de parser niet omgaan met zogenaamde left recursion. Hier volgt een voorbeeld om het probleem met left recursion weer te geven:

Expr → Expr + Expr | Int | String

Met de bijbehorende C# code voor het parseren:

```
public void ParseExpr()
{
    ParseExpr();
    MatchToken('+');
    …
}
```

Het parseren van bovenstaande grammatica leidt in de code tot een oneindige loop. Dit probleem kan worden opgelost door gebruik te maken van een methode genaamd left factoring (Moore 2000). Bij left factoring wordt de productie opgesplitst in meerdere producties, waardoor de productie niet naar zichzelf wijst en dus geen sprake meer is van left recursion:

Expr → Int ExprRest | String ExprRest
ExprRest → ε | + Expr

In bovenstaand voorbeeld is de productie opgesplitst in twee producties. De eerste productie beschrijft een Expr, die kan bestaan uit een int of string en wordt gevolgd door een ExprRest. De RestExpr is leeg of bevat een + gevolgd door een expressie. De tweede productie verwijst weliswaar naar de eerste productie, maar de parser kan nooit meer in een oneindige loop komen.

Om dit nader toe te lichten volgt de C# code voor het parseren van deze herschreven grammatica:

```
public void ParseExpr()
{
    ParseType();
    ParseExprRest();
}
public void ParseExprRest()
{
    if(MatchToken('+')
    {
        ParseExpr();
    }
}
```

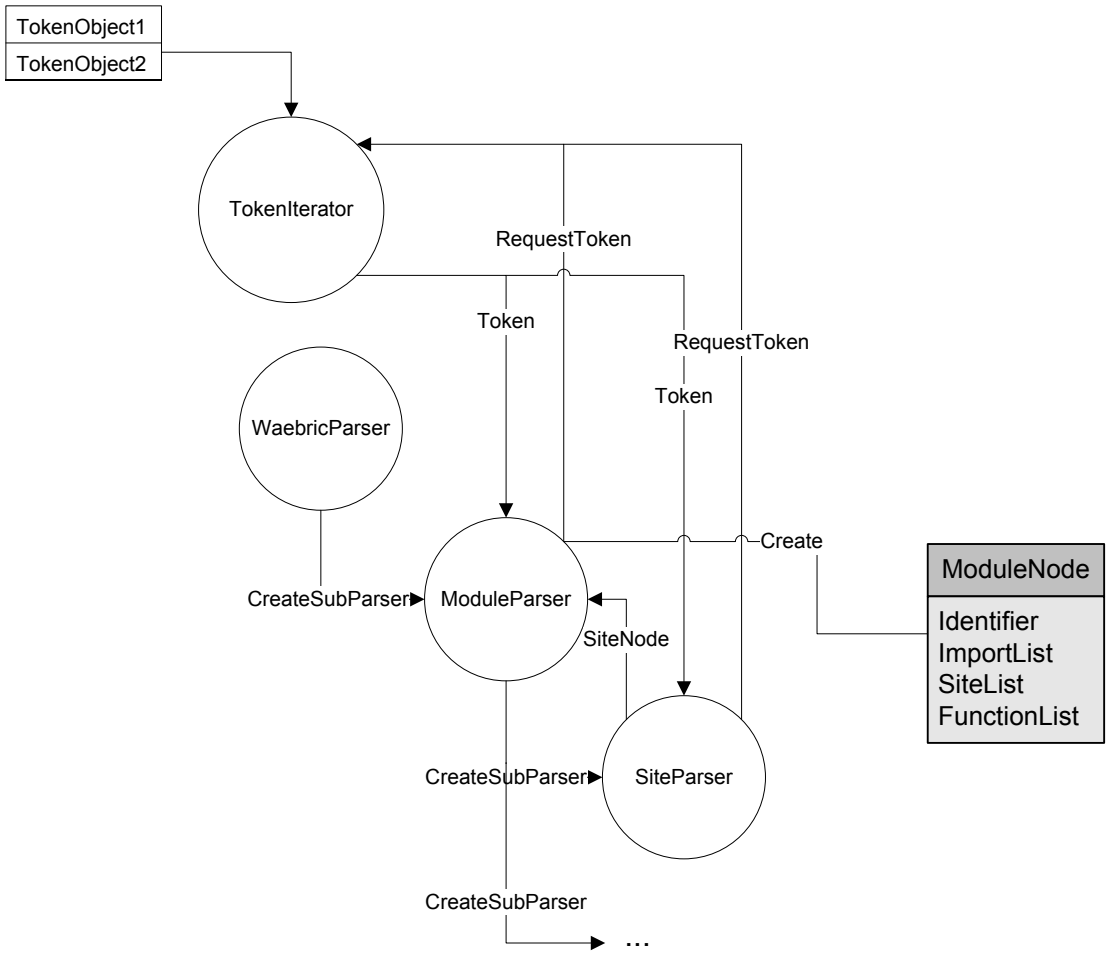
De parser parseert nu altijd eerst het type van de expressie en daarna pas het resterende deel van de expressie, waardoor deze niet meer in een oneindige loop kan komen.

GLR parseren (“Oslo”)

De “Oslo” parser maakt gebruik van het GLR parseeralgoritme. GLR staat voor algemeen, van links naar rechts en meest rechtse afleiding. Een GLR parser werkt bottom-up. Dit houdt in dat de parser vanuit de terminals naar boven toe de non-terminals (producties) probeert te matchen, totdat het startpunt van de grammatica is bereikt. Een voordeel van dit algoritme ten opzichte van het LL parseer algoritme is dat deze om kan gaan met meerdere typen grammatica’s. Zo is grammatica voor een LL parser ook geschikt voor een GLR, maar niet andersom. Zo kan GLR namelijk omgaan met left recursion. Ook ambiguïteit vormt geen probleem (meerdere keuzes). Dit komt omdat een GLR parser alle mogelijke varianten probeert, tot de variant wordt gevonden die een succesvol resultaat oplevert (Tomita 1991).

Werking vanilla C# parser

In Figuur 6 is het proces van de vanilla C# parser weergegeven.



Figuur 6: Proces vanilla C# parser

De parser bestaat zoals in Figuur 6 uit verschillende subparsers. Een subparser is een parser die een specifiek deel van de grammatica parseert. In totaal zijn er 8 subparsers voor modules, sites, functies, statements, predicaten, markup, expressies en embedding. Op deze manier kan de parser relatief eenvoudig worden uitgebreid door subparsers voor eventuele nieuwe grammatica toe te voegen.

Iedere subparser haalt de tokens op door gebruik te maken van een TokenIterator. De TokenIterator kan door de lijst van tokens itereren en biedt de mogelijkheid om vooruit te kijken in de lijst. Aan de hand van de tokens worden door de subparsers AST objecten aangemaakt en deze worden teruggeven aan de parser die de subparser heeft aangeroepen die vervolgens zorgt dat deze objecten op de juiste plek in de AST worden opgeslagen. Om het proces duidelijk te maken volgt een concreet voorbeeld aan de hand van de volgende lijst met tokens (tokens gescheiden door ;):

```
”module”;”voorbeeld”;”site”;”index.html”;”:”,”main”;”end”
```

Wanneer eerder genoemde lijst van tokens moet worden geparseerd zal de WaebricParser een ModuleParser creëren. Dit is altijd hetzelfde, omdat een Waebric bestand altijd uit één module bestaat. Vervolgens haalt de ModuleParser de eerste tokens op en creëert een ModuleNode (AST node). In deze node wordt de identifier opgeslagen, in dit geval is dat `voorbeeld`, het tweede token in de lijst met tokens. Volgens de Waebric grammatica kan een module sites, imports en functies bevatten. In dit geval wordt het volgende token opgehaald, dat het keyword site is. De ModuleParser creëert daarom een SiteParser en roept deze aan.

De SiteParser leest vervolgens alle tokens één voor één in totdat het einde van het siteblok is bereikt (end keyword). De relevante tokens, in dit geval de bestandsnaam en de functienaam, worden opgeslagen in een SiteNode (AST node) (niet weergegeven in figuur). De niet relevante tokens worden enkel gebruikt voor het herkennen van de structuur. De SiteNode wordt geretourneerd aan de ModuleParser, die deze node opslaat in de lijst van sites in de ModuleNode.

Werking “Oslo” parser

De “Oslo” parser maakt net zoals de lexer gebruik van MGrammar. In plaats van token regels, worden hiervoor syntax regels gebruikt. Om dit toe te lichten volgt een voorbeeld in MGrammar:

```
…
interleave Skippable = Whitespace | Comment;
syntax Main = m:Module => m;
syntax Module
    = ”module” m:ModuleId e:ModuleElement*
        => Module[m,valuesof(e)];
syntax ModuleId
    = item:IdCon
        => ModuleId[item]
    | item:IdCon "." list: ModuleId
        => ModuleId[item, valuesof(list)];
…
```

Het voorbeeld toont een klein deel van de Waebric grammatica in MGrammar. Met behulp van syntax regels, worden producties gedefinieerd die worden gebruikt door de parser om een AST te genereren. In deze producties kan worden gerefereerd aan token regels, andere syntax regels en karakters.

Een speciale productie die kan worden gebruikt is de interleave regel. De interleave regel wordt gebruikt om bepaalde tokens tijdens het parseren te negeren. In het voorbeeld worden whitespacekarakters en commentaarblokken door de parser genegeerd. In de syntaxregels hoeft hierdoor geen rekening meer te worden gehouden met whitespace en comments.

De syntax van een taal in MGrammar moet altijd beginnen met een syntax Main regel. Dit is het startpunt van de taal. Binnen syntax regels kan gebruik worden gemaakt van de => operator. Met behulp van deze operator kan de AST worden beschreven, dit wordt in MGrammar ook wel een reflectie

genoemd. Om de essentiële tokens uit een productie in een reflectie te kunnen gebruiken, kunnen deze worden geparametriseerd, zoals weergegeven in het voorbeeld.

Het is ook mogelijk om met behulp van syntaxregels lijsten te definiëren. De syntax regel van een ModuleId uit het voorbeeld is een voorbeeld van een lijstconstructie. Dit wordt gedaan door in de productie naar zichzelf te refereren. Concreet staat er: Een lijst bestaat uit een identifier of uit een identifier gevolgd door een punt en een ModuleId. In de reflectie wordt door middel van de valuesof operator de geneste AST structuur vereenvoudigd. Bij de input “a.b.c” wordt bij gebruik van de valuesof operator de inhoud van de AST node: `ModuleId[a,b,c]`. Zonder gebruik van de valuesof operator wordt de inhoud van de AST node: `ModuleId[a,ModuleId[b,ModuleId[c]]]`.

Om de werking van de syntax regels toe te lichten zal met behulp van de grammatica een regel Waebric worden geparseerd:

```
module test
```

De parser krijgt van de tokenizer deze input als drie tokens, namelijk een modulekeyword, whitespace en een identifier. Op basis van de syntax regels moeten deze tokens worden gematched. De token modulekeyword wordt gematched als een reeks van karakters in de vorm van ”module”. Vervolgens wordt de identifier gematched als een ModuleId. Het geheel matched weer met een Module. Omdat de parser nu bij het startpunt van de grammatica is uitgekomen is de parser klaar en ziet de AST er als volgt uit:

```
Module[
    ModuleId[
        ”test”
    ]
]
```

Samenwerking tussen “Oslo” lexer en parser

In “Oslo” worden token regels altijd als eerste verwerkt (door de lexer) en daarna worden de syntax regels (door de parser) verwerkt. De lexer zoekt vooraf altijd de langst mogelijke match. Indien meerdere tokens matchen met de input, dan zal altijd de langst mogelijke variant worden gematched. Om het probleem te beschrijven volgt hier een concreet voorbeeld in MGrammar en in Waebric:

```
token PathChar = ('\u0021'..'\u002D' | '\u002F'..'\u005A' |
'\u005E'..'\u007E');
token Filename = PathChar+ ^('/') '.' (Letter | Digit)+;

syntax Statement_No_Markup = …
                           | "if" "(" p:Predicate ")" ts:Statement
                              => IfStatement[p,TrueStatement[ts]]
                          …
syntax Predicate = …
                 | e:Expression "." t:Type "?"
                     => IsAPredicate[e, t]
                 …
```

Met deze grammatica specificatie proberen we het volgende stuk Waebric code te parseren:

```
if(variabele.string?)
{
   ...
}
```

Wanneer bovenstaand stuk Waebric code wordt geparseerd zal een parser error ontstaan. De lexer zal bovenstaand voorbeeld als volgt tokenizen (tokens gescheiden door komma’s:

```
”if”,Filename,”)”,”\n”,”{”…”}”
```

In plaats van een Filename dient de tokenizer het volgende te tokenizen:

```
”if”,”(”,identifier,”.”,identifier,”?”,”)”,”\n”,”{”…”}”
```

In het eerste voorbeeld is een Filename token een langere match dan de tokens, zoals weergegeven in het tweede voorbeeld. Dit probleem is in de huidige versie van “Oslo” niet op te lossen. Daarom is in de “Oslo” implementatie van Waebric gekozen om bestandsnamen met een / of ./ te laten beginnen.

Een oplossing voor dit probleem is mogelijk indien de lexer en parser in “Oslo” zouden samenwerken. In plaats van de langste match, zou de match die de parser vooruit helpt leidend moeten zijn. In het concrete voorbeeld zal dan nooit een Filename token worden gematched door de lexer, omdat de parser hier verder niets mee kan. Dit probleem is inmiddels onderkend door de ontwikkelaars van “Oslo” en zal in een toekomstige versie van “Oslo” zijn verholpen.

AST structuur in vanilla C#

De AST in de vanilla C# implementatie is een statisch getypeerde AST (Grimm 2007). Voor iedere non-terminal in de grammatica is een statische AST node gecreëerd. De verschillende subparsers creëren op basis van de tokens hiervan instanties die samen een hiërarchische AST structuur vormen. De reden dat gekozen is voor een statische AST is het feit dat hiermee typesafety wordt afgedwongen. In C# ziet een statische AST node er als volgt uit:

```
public class ModuleNode : ISyntaxNode()
{
      private String Identifier;
      private NodeList SiteList;
      private NodeList FunctionList;
      private NodeList ImportList;

      …
      public void SetIdentifier(String identifier)
      …
      public void AddSite(Site site)
      …
      public void AddFunction(FunctionDefinition function)
      …
      public void AddImport(Import import)
      …

}
```

In bovenstaand voorbeeld is de structuur van de Waebric grammatica te herkennen. Zie eerdere voorbeelden van MGrammar waarbij deze structuur ook voorkomt. In dit voorbeeld zijn de non-terminals als class members terug te vinden. Voor iedere member is vervolgens een set/get of add/getmethode (in het geval van een lijst) gemaakt. Alle nodes overerven van een ISyntaxNode interface om een uniforme manier van het doorlopen van de AST aan te kunnen bieden. Dit is in paragraaf 4.3 beschreven.

AST structuur in “Oslo”

De AST structuur van “Oslo” is een dynamisch getypeerde AST (Grimm 2007). Deze AST bestaat uit één type node. De node bevat twee members, namelijk een Brand en een AtomicValue. Het brand member van de node bevat een string representatie van het type node. Wanneer een node bijvoorbeeld van het type Module is, bevat het Brand member de waarde “Module”. Het AtomicValue bevat een string representatie van de waarde van de node, indien deze een waarde heeft. In paragraaf 4.3 is beschreven hoe nodes uit de “Oslo” parser kunnen worden gebruikt in C# code.

“M” Runtime en C#

De lexer en parser van “Oslo” zijn ondergebracht in een zogenaamde “M” Runtime. De runtime kan vanuit C# worden aangeroepen om een parser voor een grammatica gedefinieerd in MGrammar te creëren. Hier volgt een voorbeeld om duidelijk te maken hoe de “Oslo” parser in C# moet worden aangeroepen:

```
using Microsoft.M;
using System.Dataflow;
…

MImage grammarImage = new MImage(”pathtomxfile”);
DynamicParser parser = new
grammarImage.ParserFactories[”Waebric.Waebric”].Create();
Parser.GraphBuilder = new NodeGraphBuilder();
Node rootNode = Parser.Parse(”pathtoinput”, ErrorReporter.Standard);
…
```

In bovenstaand voorbeeld wordt eerst een image ingelezen van een voorgecompileerd MGrammar bestand. Met behulp van dit image wordt door een parserfactory een parser aangemaakt voor de taal Waebric. Vervolgens moet een NodeGraphBuilder worden gespecificeerd om de MGraph/AST om te zetten in nodes. Vervolgens kan met de parse methode van de parser een willekeurig bestand worden geparseerd. Eventuele foutmeldingen worden via een errorreporter afgehandeld.
Om ervoor te zorgen dat het image en het parserobject niet voor ieder bestand (in het geval van Waebric imports) opnieuw moeten worden gecreëerd, is gekozen om de parser instantie in een singleton (Gamma, Helm et al. 1994) klasse onder te brengen.

Imports in Waebric

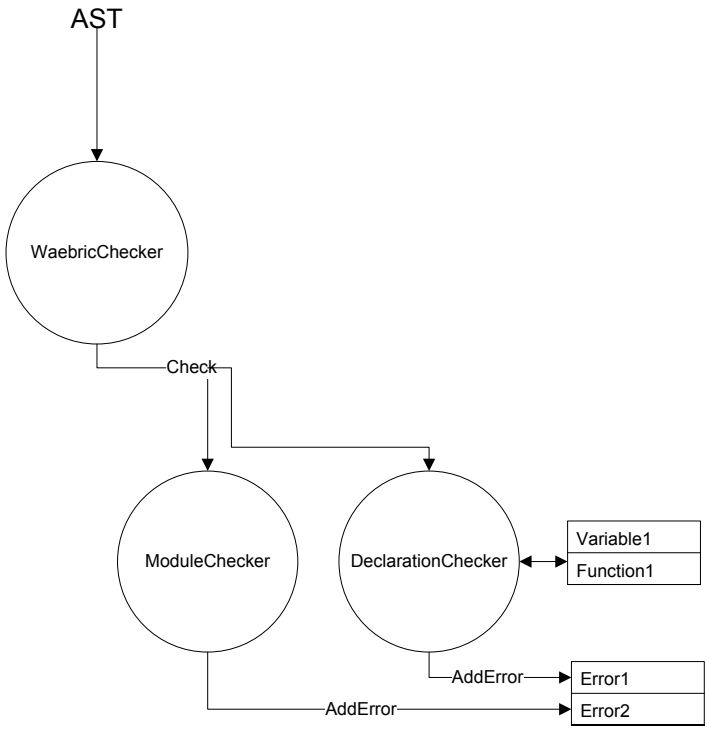
In de taal Waebric is het mogelijk om externe Waebric bestanden te importeren en functies aan te roepen die gedefinieerd zijn in deze externe bestanden. In beide implementaties kunnen de parsers zelf hiermee geen rekening houden. De parsers herkennen alleen import blokken en slaan deze als een node op in de AST. Na het parseren worden eerst de import nodes in de AST nagelopen en wordt de parser eventueel opnieuw aangeroepen om andere Waebric bestanden te parseren. Alle AST’s worden daarna opgeslagen in een cache, zodat de checker en de interpreter direct over alle benodigde AST’s kunnen beschikken.

4.3 Checker

De checker is een component, die een aantal semantische controles uitvoert op de AST. Het gaat hierbij om de volgende controles:

- Controle op dubbele functiedeclaraties
- Controle op functieaanroepen met een verkeerd aantal parameters
- Controle op niet bestaande modules
- Controle op functieaanroepen naar niet gedefinieerde functies
- Controle op verwijzingen naar niet gedefinieerde variabelen

In Figuur 7 is het proces van de checker te zien. Beide implementaties werken hetzelfde, alleen de manier waarop de AST wordt doorlopen is verschillend, waarover later meer.



Figuur 7: Proces checker

De Checker bestaat uit drie componenten, de WaebricChecker, de ModuleChecker en de DeclarationChecker. De WaebricChecker leest de AST in en roept achtereenvolgens de ModuleChecker en de DeclarationChecker aan. De ModuleChecker doorloopt de AST en controleert of alle Waebric imports bestaan. Indien een import niet bestaat wordt een foutmelding aan een lijst met foutmeldingen toegevoegd. De DeclarationChecker doorloopt eveneens de AST en controleert of alle variabelen- en functiedeclaraties correct zijn en maakt hiervoor gebruik van een tabel om deze tijdelijk op te slaan (zie SymbolTable). Fouten worden hierbij eveneens aan de lijst met foutmeldingen toegevoegd. Deze fouten

worden aan de gebruiker ter kennisgeving getoond, maar voorkomen niet dat er geen output wordt gegenereerd door de interpreter.

AST Treewalking vanilla C#

In de vanilla C# implementatie wordt zoals eerder vermeld gebruik gemaakt van een statisch getypeerde AST bestaande uit nodeobjecten. Om de AST te doorlopen wordt gebruikt gemaakt van het Visitor Pattern (Gamma, Helm et al. 1994).

Het visitor pattern bestaat uit twee onderdelen, de visitor en objecten die de visitor bezoekt, in dit geval AST nodes. De visitor is een klasse die alle AST objecten bezoekt en hierop een actie wil uitvoeren, in dit geval een checker of interpreter. Om een visitor de mogelijkheid te bieden om alle nodes te bezoeken moet iedere node een AcceptVisitor methode implementeren:

```
public class Module : ISyntaxNode
{
	…
	public void AcceptVisitor(ISyntaxNodeVistor visitor)
	{
		visitor.Visit(this);
	}
	…
}
```

In de AcceptVisitor methode wordt de Visit methode van het specifieke nodetype aangeroepen. De visitor implementeert voor alle nodetypen één Visit methode waarin de acties voor dit specifieke nodetype worden uitgevoerd:

```
public class ModuleChecker : ISyntaxNodeVisitor
{
	…
	public void Visit(Module module)
	{
		…
		Module.GetIdentifier().AcceptVisitor(this);
		…
	}
	…
}
```

In bovenstaand voorbeeld is de Visit methode van een Waebric module weergegeven. In dit geval wordt een ModuleId opgehaald en wordt hiervan de AcceptVisitor aangeroepen om de specieke ModuleId node te bezoeken. Voordeel van deze manier van werken is dat de AST op een type safe wijze kan worden doorlopen (Grimm 2007).

AST Treewalking “Oslo”

De “Oslo” implementatie maakt gebruik van een dynamisch getypeerde AST, zoals eerder beschreven. Deze objecten kunnen niet worden doorlopen door middel van het Visitor Pattern, omdat de nodeklasse niet uitbreidbaar is door middel van overerving. In C# wordt daarom de nodestructuur door middel van loops doorlopen:

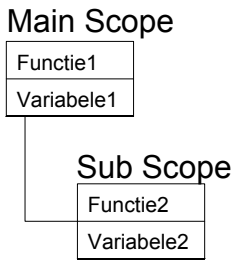
```
public void VisitChildNodes(Node node)
{
      NodeCollection childs = node.ViewAllNodes()
      foreach(Node child in childs)
      {
            switch(child.Brand.Text)
            {
                  case ”FunctionDef”:
                  …
                  case ”EachStatement”:
                  …
            }

      }
}
```

Een nadeel van deze methode is dat dit extra complexiteit in de C# code introduceert. Bovendien moeten alle nodetypen hardcoded in de code worden vastgelegd. Tenslotte kan de AST niet type safe worden doorlopen.

SymbolTable

Om tijdens het doorlopen van de AST bij te kunnen houden welke variabelen en functies zijn gedefinieerd, wordt een tabel bijgehouden met variabelen en functies, de zogenaamde SymbolTable (zoals ook weergegeven in Figuur 7). De SymbolTable is een dictionary of map, met daarin variabelen met de bijbehorende expressie en functies met de bijbehorende referentie naar de functienode uit de AST.

Binnen Waebric komt beperkt scoping voor. Scoping is het bereik van variabelen en functies. Om in een SymbolTable hiermee rekening te houden is ervoor gekozen voor iedere scope een eigen SymbolTable aan te maken die verwijst naar een parent SymbolTable. In Figuur 8 is dit schematisch weergegeven.



Figuur 8: Scoping

In Figuur 8 is sprake van twee scopes. In de Sub Scope kunnen zowel de functies en variabelen van zowel de Main Scope als de SubScope worden aangeroepen of gebruikt. In de Main Scope zijn alleen de functies en variabelen van de Main Scope zichtbaar. Wanneer een variabele/functie worden gebruikt in een “lagere” scope, die dezelfde naam heeft als een variabele/functie in een “hogere” scope, dan kan alleen de variabele/functie uit de “lagere” scope worden gebruikt. Dit wordt ook wel function/variable hiding genoemd.

LetStatement

Waebric kent zogenaamde LetStatements. Met behulp van letstatements kunnen functies en variabelen worden gedeclareerd binnen een bepaalde scope. Om de werking inzichtelijk te maken volgt een voorbeeld in Waebric:

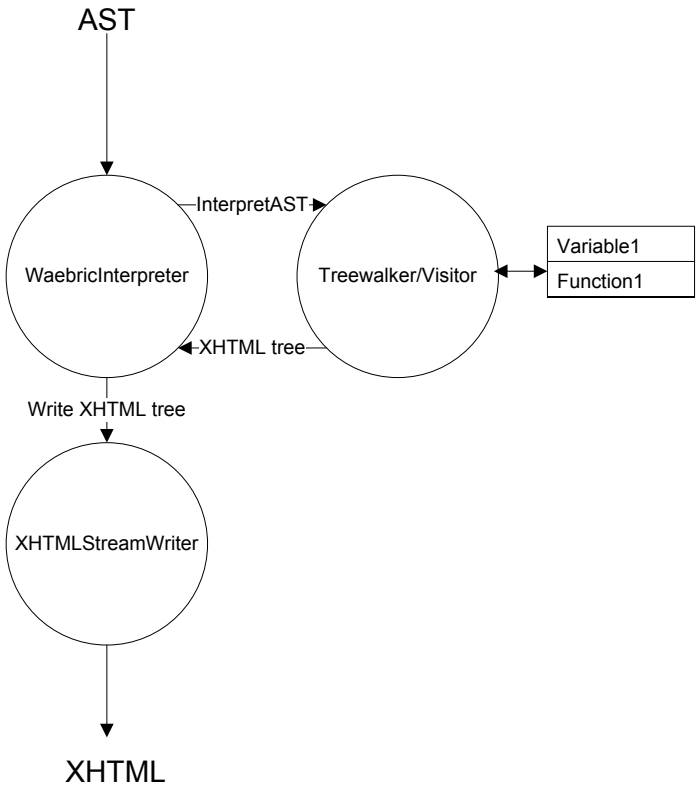
```
def examplefunction
    let
        function1(text) = echo text;
        function2() = function1(”tekst”);
    in
        function2;
    end
end
```

Een LetStatement bestaat uit twee blokken. In het eerste blok kunnen functies en variabelen worden toegewezen, ook wel assignments genoemd. Speciaal aan deze assignments is dat iedere assignment een eigen scope kent. Zo kan in het tweede assignment worden gerefereerd aan het eerste assignment, zoals ook in dit voorbeeld weergegeven. Om de variabelen en functies binnen LetStatement semantisch te kunnen controleren, wordt voor iedere scope een nieuwe SymbolTable aangemaakt.

In LetStatements is het mogelijk om functies te definiëren. De parser slaat deze echter niet als een functie op in de AST. Daarom wordt tijdens het checken en interpreteren de assignmentnode in de AST omgezet in een functienode. Daardoor kan deze op dezelfde manier worden opgeslagen in een SymbolTable als ieder andere functie in een Waebric programma.

4.4 Interpreter

De interpreter is een component, die de AST doorloopt volgens de control flow. Hierbij wordt een XHTML boomstructuur opgebouwd, die wordt weggeschreven naar een XHTML bestand. In Figuur 9 is het proces van de interpreter weergegeven. Beide implementaties werken net zoals bij de checker hetzelfde, alleen de manier van het doorlopen van de AST is verschillend.



Figuur 9: Proces interpreter

De WaebricInterpreter leest de AST in en geeft opdracht aan de Treewalker/Visitor om de AST te interpreteren. De Treewalker/Visitor doorloopt de boom en maakt hierbij SymbolTables aan om de actuele waarden van de functies en variabelen bij te houden en om rekening te houden met scoping. Tijdens het doorlopen van de boom, wordt een XHTML boomstructuur opgebouwd. Na het doorlopen van de AST wordt deze vervolgens door een XHTMLStreamWriter weggeschreven naar een XHTML bestand.

Yield statement

Waebric kent zogenaamde YieldStatements. Met behulp van een YieldStatement kunnen html markup en statements gemakkelijk worden herbruikt. Om de werking van een YieldStatement inzichtelijk te maken volgt een concreet voorbeeld:

```
def home(msg)
    layout(msg) echo msg;
end

def layout(title)
    html {
        head title title;
        body yield;
    }
end
```

In bovenstaand voorbeeld resulteert het gebruik van het YieldStatement in een functiecall naar de desbetreffende layout functie de markups en statements na de functiecall, die deel uitmaken van hetzelfde statement, worden uitgevoerd op de plaats van het YieldStatement. In dit geval betekent het dat op de plek van het YieldStatement, het statement `echo msg;` wordt uitgevoerd. In beide implementaties wordt dit gedaan door gebruik te maken van een stack, waarop de statements worden gepushed op het moment dat een functie met een YieldStatement wordt aangeroepen. Omdat het statement dat op de stack wordt gepushed zich in een andere scope bevindt binnen het Waebric programma, wordt ook de SymbolTable op de stack gepushed. Bij het bereiken van het YieldStatement wordt het statement en de SymbolTable van de stack gepopped en uitgevoerd op die plaats.

Wanneer in eerder genoemd voorbeeld de functie home wordt aangeroepen met de parameter “Hello World!”, dan levert dit de volgende XHTML op:

```
<html>
    <head>
        <title>Hello World!</title>
    </head>
    <body>Hello World!</body>
</html>
```

XHTML genereren

Om van de AST structuur een XHTML bestand te kunnen maken is een XHTML boomstructuur nodig. XHTML is afgeleid van XML. In C# zou voor het maken van een XHTML boomstructuur gebruik kunnen worden gemaakt van bij C# meegeleverde XML bibliotheken. Karakter escaping vormt echter een probleem om van deze bibliotheken gebruik te kunnen maken.

Een Waebric voorbeeld ter toelichting:

```
dit is tekst met een   no break space en een & karakter
```

Het voorbeeld bevat zowel escaped ( ) en niet escaped (&) karakters door elkaar, wat correcte Waebric is. De C# XML bibliotheken escapen altijd deze karakters om correcte XML te produceren met het volgende resultaat:

```
dit is tekst met een &amp;nbsp; no break space en een &amp; karakter
```

Bovenstaande tekst is niet de gewenste output, omdat de escapekarakters opnieuw zijn geëscaped. Alleen het losse & karakter moet worden geëscaped en niet & karakters die onderdeel zijn van een escapekarakter. De correcte output is:

```
dit is tekst met een   no break space en een &amp; karakter
```

Omdat de werking van de XML bibliotheken niet aangepast kan worden is besloten om voor beide implementaties een gemeenschappelijke manier van XHTML boomstructuren te creëren. Hierbij wordt geen escaping in de boomstructuur uitgevoerd. De escaping vindt pas plaats, waar nodig, tijdens het wegschrijven van de boomstructuur naar een XHTML bestand. Hiervoor is een XHTMLElement klasse gemaakt, waarmee een XHTML boomstructuur kan worden gecreëerd. Tevens is een XHTMLStreamWriter gemaakt, waarmee deze boomstructuur naar een bestand kan worden weggeschreven. Deze maakt hierbij gebruik van een bij C# meegeleverde XHTMLWriter, voor het creëren van XHTML bestanden.

5 Resultaten

In dit hoofdstuk worden de resultaten besproken van de metingen die zijn verricht aan beide implementaties. Hierbij wordt alleen ingegaan op opvallende resultaten. De volledige resultaten zijn opgenomen in de appendices.

5.1 Functionaliteit

Alle acceptatietests zijn succesvol (100% score) doorlopen bij beide implementaties. Hierbij moet worden opgemerkt dat door de acceptatietests geen CData statements en Site definities van Waebric worden getest. De site definities zijn niet volledig geïmplementeerd in “Oslo”, zoals vermeld in paragraaf 4.2. Het resultaat zou dus bij een volledige acceptatietest bij de implementatie“Oslo” lager liggen.

Voor een overzicht van de resultaten van de acceptatietests zie Bijlage A: Resultaten acceptatietests.

5.2 Onderhoudbaarheid

De NCLOC, MLOC en E zijn gemeten door gebruik te maken van de C# 2.0 Metrics tool van Semantic Designs. De MCC en #M zijn berekend met de tool SourceMonitor[1]. De MI3 is handmatig berekend met behulp van de parameters uit de C# 2.0 Metrics tool, omdat de tool MI volgens een andere formule uitrekent.

In Tabel 2 zijn de resultaten van de metingen van de vanilla C# implementatie weergeven. In de tabel zijn het aantal regels code zonder commentaar (NCLOC), aantal regels code in methoden (MLOC), gemiddelde McCabe Complexity per methode (MCC), Halstead Effort (E), Maintainability Index (MI3) en aantal methoden (#M) weergegeven.

Metric	Lexer Core	Lexer Token	Parser Core	Parser AST	Checker	Interpreter	Utils
NCLOC	588	151	1404	3357	298	1256	404
MLOC	513	105	1237	2348	197	1130	248
MCC	3,21	1,37	3,12	1,07	2,14	2,99	2,04
E	1.304.337	34.516	1.645.219	692.661	134.731	3.474.242	224.344
MI3	109	126	106	133	116	108	116
#M	42	19	91	535	22	85	27

Tabel 2: Metingen onderhoudbaarheid vanillaC# implementatie

De metingen zijn verdeeld over de componenten lexer, parser, checker en interpreter. De lexer en parser zijn daarbij opgesplitst in twee delen, een deel met de code voor het lexen en parsen (core) en een deel met code voor tokens en de AST. Bovendien is een utils component opgenomen. Hierin bevindt zich de cache voor AST’s, de SymbolTable en het startpunt van de applicatie (main methode). De utils component is hetzelfde geïmplementeerd in beide implementaties, alleen de typering van de AST is verschillend.

[1] De C# Metrics tool van Semantic Designs bevat een fout in de berekening van McCabe Complexity. Dit is na mailverkeer met Semantic Designs bevestigd. Daarom is de McCabe Complexity gemeten door gebruik te maken van de tool SourceMonitor.

Opvallende waarden zijn te vinden bij de lexer core, parser core en interpreter. Al deze componenten hebben een hoge Halstead Effort (> 1 miljoen) en een lage MI3 waarde.

De verklaring voor de hoge Halstead Effort is dat deze componenten complexer (bevatten dus meer operatoren en operanden) zijn (MCC is gemiddeld ≈3 of hoger) dan de overige componenten in combinatie met het aantal NCLOC. Een hogere NCLOC resulteert in meer operanden en operatoren en dus een hoger effort. Echter een regel code met een hoge MCC heeft meer operatoren en operanden dan een regel code met een lage MCC. Zo heeft de parser AST component een hoog aantal NCLOC (3357), maar een lage MCC (1,07) dat resulteert in een veel lagere effort in vergelijking tot de parser core component (NCLOC 1404, MCC 3,12).

De Maintainability Index (MI3) is opvallend lager bij de lexer core, parser core en interpreter dan bij de overige componenten. Dit heeft voornamelijk te maken met de Halstead Effort, de NCLOC en het aantal methoden. De MI3 wordt berekend door drie componenten, het gemiddelde Halstead Volume (subcomponent van Halstead Effort) per methode, het gemiddelde MCC per methode en het gemiddelde NCLOC per methode. De componenten Halstead Volume en NCLOC tellen hierbij het zwaarste mee. Voor ieder component geldt: Hoe hoger de waarde, hoe lager de MI3. Een hoger aantal NCLOC per methode levert dus een lagere MI3 op. Het aantal methoden is sterk van invloed op de MI3 waarde, omdat wordt gerekend met gemiddelden per methode.

Bij de lexer core, parser core en interpreter component is sprake van een hoge Halstead Effort (dus een hoog Halstead Volume) en een hoge NCLOC ten opzichte van het aantal methoden. Dit resulteert in een lagere MI3 waarde.

De “Oslo” implementatie is op dezelfde manier gemeten als de C# implementatie, behalve de grammatica. Deze is handmatig gemeten (volgens de methode beschreven in paragraaf 3.2), omdat geen tools beschikbaar zijn die dit van MGrammar kunnen meten. De resultaten van de metingen zijn weergegeven in Tabel 3. In de tabel zijn het aantal regels code zonder commentaar (NCLOC), aantal regels code in methoden (MLOC), gemiddelde McCabe Complexity per methode (MCC), Halstead Effort (E), Maintainability Index (MI3) en aantal methoden weergegeven.

Metric	Grammar (“M”)	Parser (C#)	Checker (C#)	Interpreter (C#)	Utils (C#)
NCLOC	335	51	394	1576	415
MLOC	-	32	303	1463	266
MCC	2,11	1,25	2,54	3,82	2,08
E	50.873	11.348	450.861	6.117.914	270.731
MI3	134	119	110	103	114
#M	-	4	26	89	28

Tabel 3: Metingen onderhoudbaarheid "Oslo" implementatie

De componenten zijn op andere wijze ingedeeld dan bij de vanilla C# implementatie. Dit omdat de AST klasse door “Oslo” wordt geleverd. De parser component bestaat in deze implementatie enkel uit één klasse voor het aanroepen van het “M” Framework.

Bij de “Oslo” implementatie zijn bij de interpreter component opvallende waarden te zien. De Halstead Effort bedraagt meer dan 6 miljoen en een in verhouding hoge MCC van 3,82. In vergelijking met de

vanilla C# interpreter is het verschil aanzienlijk. Een verklaring hiervoor is de inefficiënte methode van het doorlopen van de AST. Dit verschil is in mindere mate ook zichtbaar bij de checker componenten van beide implementaties. Dit resulteert bij zowel de “Oslo” checker en interpreter in een lagere MI3 (110 en 103) dan bij de vanilla C# checker en interpreter (116 en 108).

De grammar en parser component vallen op door de lage Halstead Effort en hoge MI3 waarde. Bij de grammar is dit te verklaren door het feit dat MGrammar veel minder type operatoren kent dan C#. Daarnaast is ook het aantal typen operanden in MGrammar beperkt. Dit resulteert in een lager effort en dus uiteindelijk ook in een hogere MI3 waarde. De parser component heeft een lage Halstead Effort, omdat deze enkel het “M” Framework hoeft aan te roepen en zelf geen parseertaken uitvoert.

De grammar en parser componenten van de “Oslo” implementatie hebben in vergelijking tot de lexer en parser componenten van de vanilla C# implementatie een lagere Halstead Effort en een hogere MI3 waarde. Het grote verschil zit in het feit dat bij de vanilla C# implementatie de lexer en de parser moeten worden ontwikkeld en bij het gebruik van “Oslo” enkel de grammatica en de code voor het aanroepen van het “M” Framework.

Voor een compleet overzicht van de meetresultaten zie Bijlage B: Resultaten metingen onderhoudbaarheid**.**

5.3 Performance

De performance wordt gemeten door een serie tests waarbij Waebric bestanden 1000 keer worden geparseerd, gecheckt en geïnterpreteerd. Per component van de implementaties wordt gemeten hoeveel tijd deze verbruikt. De tijd wordt op twee manieren gemeten. De eerste manier van meten is wallclock time (verder kloktijd genoemd). Hierbij wordt op basis van de kloktijd bepaald hoe lang een component heeft gedraaid. Bij de tweede manier wordt alleen de processortijd gemeten, de tijd waarbij de component daadwerkelijk op de processor heeft gedraaid. Het verschil tussen beide meetmethoden is dat bij kloktijd geen rekening wordt gehouden met het feit dat het besturingssysteem andere processen op de processor kan laten draaien. Bij processortijd is hier wel rekening mee gehouden en wordt deze tijd niet meegerekend.

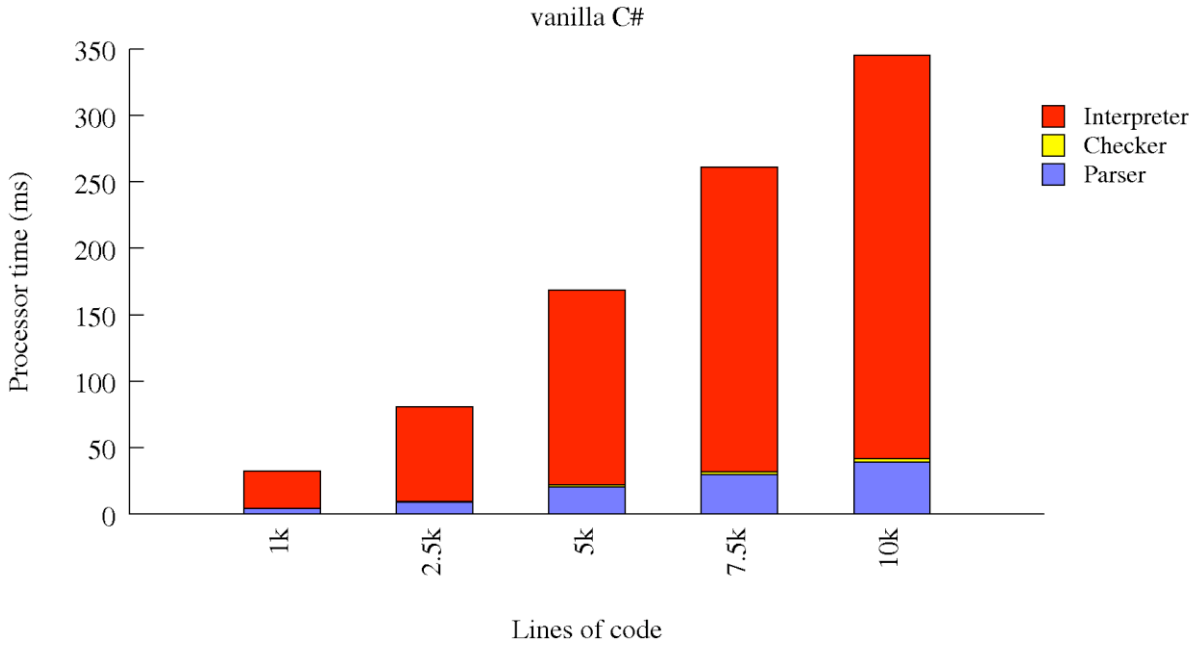
In totaal zijn twee testseries uitgevoerd. De eerste serie bestaat uit een drietal Waebric bestanden met verschillende groottes en verschillende complexiteit. De tweede serie bestaat uit Waebric bestanden met verschillende groottes en gelijke complexiteit, zodat beter inzicht kan worden verkregen in de schaalbaarheid.

Component	**Menus (54 LOC)**		**LDTA (269 LOC)**		**Lava (967 LOC)**	
	Vanilla C#	**“Oslo”**	**Vanilla C#**	**“Oslo”**	**Vanilla C#**	**“Oslo”**
GrammarLoader (ms)	-	547,05	-	552,50	-	550,68
Parser (ms)	0,45	9,15	1,96	154,77	8,59	318,27
Checker (ms)	0,24	0,31	1,56	1,99	8,10	641,11
Interpreter (ms)	2,37	4,08	19,18	27,98	50,03	375,10

Tabel 4: Performance testserie 1 (Processortijd)

In Tabel 4 is het resultaat van de eerste testserie weergeven, waarbij processortijd is gemeten. De meting is opgedeeld in vier componenten, namelijk grammarloader, parser, checker en interpreter. De grammarloader is alleen van toepassing op de “Oslo” implementatie, omdat daarbij iedere keer de gecompileerde MGrammar moet worden ingelezen en een parser instantie moet worden gecreëerd.

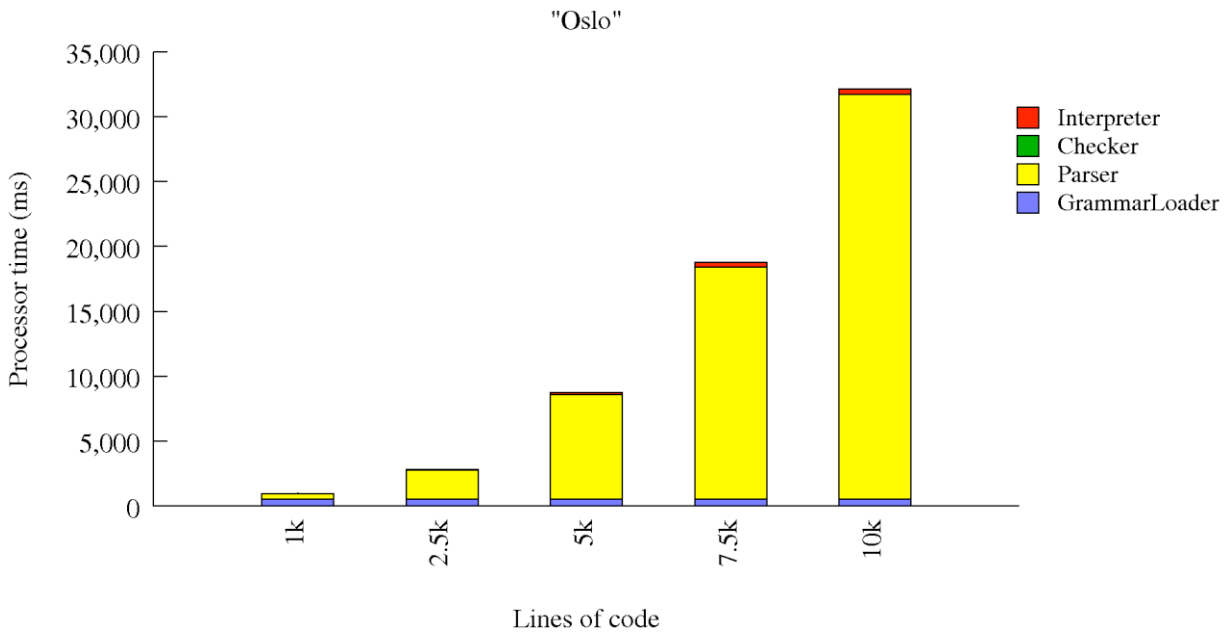
De metingen laten zien dat het laden van de grammatica een vrij constant resultaat oplevert (±550 ms). De verschillen zijn echter groter bij de parser, checker en interpreter. Opvallend is het grote verschil tussen de vanilla C# parser en de “Oslo” parser. Bij de grotere en complexere LDTA en Lava bestanden loopt het verschil op tot meer dan 300 milliseconden. Een verklaring hiervoor is het complexere parseeralgoritme van “Oslo” (GLR) ten opzichte van het gebruikte parseeralgoritme (LL) in vanilla C#. Het verschil tussen de Checker en Interpreter tussen beide implementaties is bij de grotere bestanden ook aanzienlijk groter. Een verklaring hiervoor is de minder efficiënte wijze van het doorlopen van de AST in de “Oslo” implementatie.



Figuur 10: Performance testserie 2 vanilla C# (Processortijd)

In Figuur 10 is het resultaat van de tweede testserie voor vanilla C# te zien, waarbij is gemeten in processortijd. In totaal zijn vijf Waebric bestanden met verschillende oplopende groottes en met gelijke complexiteit getest. Daardoor wordt beter zichtbaar hoe de componenten schalen.

Bij de vanilla C# implementatie schalen de checker en parser duidelijk lineair. De interpreter laat vanaf 2.5k lines of code een duidelijk lineaire verhouding zien. Opvallend is dat de interpreter aanzienlijk meer tijd nodig heeft dan de parser.



Figuur 11: Performance testserie 2 "Oslo" (Processortijd)

In Figuur 11 zijn de testresultaten van de tweede testserie voor de “Oslo” implementatie te zien. Hierbij is eveneens gemeten in processortijd.

Het meest opvallende aan de grafiek is de toename van de parseertijd. Dit loopt exponentieel op ten opzichte van het aantal lines of code. De verschillen in schaalbaarheid tussen de vanilla C# en “Oslo” implementatie zijn groot. Ter vergelijking: Bij 10k LOC Waebric code duurt het proces (tokenizen, parseren, checken en interpreteren) 345 ms (0,34 s) ten opzichte van 32127,90 ms (32,12 s). Dit is een enorm verschil. Een groot aandeel hierin is de parser van “Oslo”. Een verklaring hiervoor is het gebruikte parseeralgoritme.

De checker is niet te zichtbaar op de grafiek in Figuur 11, omdat deze in verhouding tot de parser weinig tijd verbruikt. Aan de waarden in Tabel 5 is te zien dat de checker van de “Oslo” implementatie lineair schaalt. De interpreter van de “Oslo” implementatie schaalt duidelijk niet volgens een patroon.

Component	1k		2.5k		5k		7.5k		10k	
	vanilla C#	"Oslo"	vanilla C#	"Oslo"	vanilla C#	"Oslo"	vanilla C#	"Oslo"	vanilla C#	"Oslo"
Grammar Loader	0	508,93	0	542,01	0	541,45	0	549,50	0	552,84
Parser	4,02	429,14	8,80	2205,29	20,58	8039,62	29,96	17880,15	38,92	31154,59
Checker	0,37	1,45	0,94	3,55	1,47	7,36	1,89	10,72	2,86	14,60
Interpreter	28,07	34,64	70,95	86,00	146,31	174,22	229,20	325,65	303,68	405,87
Totaal	32,46	974,16	80,69	2836,85	168,36	8762,65	261,05	18766,02	345,46	32127,9

Tabel 5: Performance testserie 2 (Processortijd)

Uit de testresultaten komt naar voren dat de performance van alle onderdelen bij “Oslo” lager is. De grammarloader heeft een vrij constante nadelige invloed hierop. De “Oslo” parser is significant trager dan de vanilla C# parser en dit verslechtert naar mate de hoeveelheid te parseren Waebric code toeneemt. De checker en interpreter van de “Oslo” implementatie zijn eveneens trager dan de vanilla C# checker en interpreter. Een verklaring hiervoor is de inefficiënte manier van het doorlopen van de AST in de “Oslo” implementatie.

In Tabel 6 zijn de resultaten weergegeven van de metingen in wallclock time.

Component	1k		2.5k		5k		7.5k		10k	
	vanilla C#	"Oslo"	vanilla C#	"Oslo"	vanilla C#	"Oslo"	vanilla C#	"Oslo"	vanilla C#	"Oslo"
Grammar Loader	0	541,36	0	552,52	0	551,89	0	555,59	0	564,00
Parser	4,11	438,98	9,70	2374,32	21,89	8643,29	30,78	18985,77	40,26	33405,32
Checker	0,38	1,58	0,93	3,91	1,39	7,80	2,11	11,32	2,93	16,90
Interpreter	32,88	42,10	78,17	95,86	156,67	210,55	243,19	375,85	321,35	430,37
Totaal	37,37	1024,02	88,80	3026,61	179,95	9413,53	276,08	19928,53	364,54	34416,59

Tabel 6: Performance testserie 2 (Kloktijd)

De resultaten uit Tabel 6 laten zien dat de kloktijden langer zijn dan de processortijden uit Tabel 5. Dit komt omdat in de kloktijd geen rekening wordt gehouden met de scheduling van het besturingssysteem. Deze tijden variëren per testrun, omdat dit sterk afhankelijk is van het aantal processen dat op het testsysteem draait. Hoe meer processen gelijktijdig draaien, hoe vaker het proces wordt onderbroken door andere processen.

Alle tests zijn uitgevoerd op een systeem met de in Tabel 7 weergegeven configuratie.

Processor:	Intel Core 2 T7400
Geheugen:	4GB DDR2 Dual Channel
OS:	Windows 7 RTM
.NET versie	3.5 SP1

Tabel 7: Configuratie testsysteem

De volledige testresultaten inclusief resultaten met kloktijd zijn terug te vinden in Bijlage C: Performance meetresultaten.

5.4 Productiviteit

De productiviteit is gemeten door de ontwikkeltijd van de verschillende componenten te meten. De ontwikkeltijd van de grammatica, lexer en parser zijn gerekend als een enkel parser component. Dit omdat bij “Oslo” geen scheiding is tussen parser en lexer. De componenten utils en interpreter bevatten herbruikbare componenten, die in beide implementaties zijn gebruikt. Voor de component utils betreft dit de cache voor AST’s en de symboltable. Voor de component interpreter betreft het de XHTML writer. In beide implementaties zijn deze daarom meegeteld als ontwikkeltijd.

In Tabel 8 zijn de resultaten van de productiviteitsmeting weergegeven.

Component	Vanilla C#	“Oslo”
Parser	123:40	39:50
Checker	13:30	13:45
Interpreter	48:35	39:10
Utils	3:45	3:45
Totaal	189:30	96:30

Tabel 8: Productiviteitsmeting (uren:minuten)

De meting laat zien dat de vanilla C# implementatie ongeveer twee keer zoveel ontwikkeltijd heeft gekost als de “Oslo” implementatie. Een verklaring hiervoor is het feit dat in de vanilla C# implementatie de lexer en parser moeten worden ontwikkeld, terwijl bij het gebruik van “Oslo” enkel de grammatica hiervoor moet worden ontwikkeld en de code om het “M” framework aan te roepen.

In Bijlage D: Resultaten productiviteitsmetin zijn de volledige resultaten van de productiviteitsmeting opgenomen.

6 Evaluatie en discussie

Dit hoofdstuk beschrijft de evaluatie van de resultaten en de validatie van het onderzoek.

6.1 Evaluatie

Per attribuut worden de verschillen tussen beide implementaties geëvalueerd.

Functionaliteit

De resultaten van de acceptatietests laten zien dat beide implementaties geen verschillen vertonen qua functionaliteit. Beiden scoren namelijk 100% bij de acceptatietests. In de acceptatietests worden niet alle Waebric elementen getest, namelijk CData en Site elementen. De “Oslo” implementatie ondersteunt geen Site elementen volgens de specificatie, door problemen in de samenwerking tussen de lexer en parser. CData elementen zijn niet getest, dus mogelijk zit hier in beide of één van beide implementaties nog een bug. Bovendien garanderen de acceptatietests niet dat er geen bugs in beide implementaties aanwezig zijn. De acceptatietests zijn vrij specifiek en deze geven dus alleen een indicatie of de specificatie in grote lijnen wordt nageleefd. Dit is in beide implementaties het geval.

Wanneer enkel naar de resultaten van de acceptatietests wordt gekeken, dan bevestigt dit de hypothese. Bij de “Oslo” implementatie is sprake van een onverwacht probleem bij Site elementen, maar dit is door eerder genoemde reden niet zichtbaar in de resultaten van de acceptatietests. Bij grammatica’s van andere DSL’s kan dit mogelijk ook een probleem vormen. Het betreft hierbij grammatica’s waarvan syntax regels overlappen met token regels. Token regels matchen in “Oslo” altijd eerder dan syntax regels, waardoor de syntax regels niet meer worden gematched door de parser. In de praktijk is dit probleem soms te omzeilen door de syntax iets aan te passen, waardoor er geen overlap meer is. Nadeel hiervan is dat bestanden geschreven in de grammatica, mogelijk moeten worden aangepast. Relatief gezien is de functionaliteit van de “Oslo” implementatie dus iets lager dan die van de vanilla C# implementatie.

Onderhoudbaarheid

De resultaten van de metingen voor onderhoudbaarheid laten vooral bij de Halstead Effort grote verschillen zien tussen beide implementaties. De Halstead Effort geeft een indicatie hoe lastig het is om software te begrijpen en onderhouden (Halstead 1977). Er zijn geen absolute waarden die aangeven of een stuk code wel of niet goed onderhoudbaar is, maar deze waarden geven samen met andere metrieken, zoals MI3 een indicatie. De verschillen in Halstead Effort en MI3 tussen de componenten van beide implementaties zijn in een aantal gevallen vrij fors. Dit is een duidelijke indicatie van verschillen in onderhoudbaarheid tussen de componenten.

De lexer en parser componenten van de vanilla C# implementatie zijn in vergelijking tot de grammar en parser van de “Oslo” implementatie minder goed onderhoudbaar. Er zijn grote verschillen qua effort en MI3 waarden gemeten. Dit komt door het feit dat bij het gebruik van “Oslo” niet de parser en lexer moet worden ontwikkeld, maar enkel de grammatica. De grammatica heeft ten opzichte van de C# code van de parser en lexer in de vanilla C# implementatie een veel lagere Halstead Effort (ruim 50.000 versus > 1 miljoen). Ook zijn de MI3 waarden van de grammatica fors hoger dan van de parser en lexer code in de vanilla C# implementatie (109 en 106 versus 134). Ook de omvang van de hoeveelheid code verschilt

behoorlijk. De grammatica bestaat uit 335 regels grammatica en 50 regels C# code voor het aanroepen van het “M” Framework. De lexer en parser in de vanilla C# implementatie bestaan uit bijna 2000 regels C# code.

De checker en interpreter van de vanilla C# implementatie zijn in vergelijking tot de checker en interpreter van de “Oslo” implementatie beter onderhoudbaar. Dit komt door het feit dat de manier van het doorlopen van de AST nodes in de “Oslo” implementatie op een inefficiënte en minder onderhoudbare wijze plaatsvindt. De vanilla C# implementatie maakt gebruik van een statisch getypeerde AST die door middel van het Visitor Pattern wordt doorlopen. De “Oslo” implementatie maakt gebruik van een dynamisch getypeerde AST die niet uitbreidbaar is en alleen door middel van looping kan worden doorlopen. Deze manier levert extra complexiteit en meer code op, waardoor de onderhoudbaarheid afneemt. De resultaten laten dit ook duidelijk zien bij de interpreter. De Halstead effort is daar aanzienlijk hoger bij de “Oslo” implementatie (6,1 miljoen) dan bij de vanilla C# implementatie (3,4 miljoen). De MI3 verschilt in mindere mate (108 versus 103).

Bij de checkers is hetzelfde beeld bij de Halstead Effort zichtbaar, maar in mindere mate (134731 versus 250861). Dit komt door het feit dat de checkers minder intensief gebruik maken van de AST in vergelijking tot de interpreters. De MI3 laat hier echter een groter verschil zien, namelijk 121 voor de vanilla C# checker versus 110 voor de “Oslo”checker.

De resultaten bevestigen deels de hypothese. Daar werd gesteld dat de “Oslo” implementatie beter onderhoudbaar zou zijn dan bij de vanilla C# implementatie, omdat de lexer en de parser bij gebruik van “Oslo” niet worden ontwikkeld, maar enkel de grammatica. Dit deel van de hypothese wordt bevestigd door de resultaten. De verschillen tussen de checkers en interpreters zijn veel groter dan verwacht. Verwacht werd dat dit nauwelijks zou verschillen, omdat de taken hetzelfde zijn en in beide implementaties zijn ontwikkeld in C#. De manier waarop de AST wordt doorlopen is van grote invloed op de onderhoudbaarheid, waarbij de vanilla C# implementatie in het voordeel is.

Performance

De resultaten van de performancetests laten grote verschillen tussen beide implementaties zien. In het bijzonder de tests, waarbij gevarieerd wordt in de hoeveelheid Waebric code, bevestigen dit. De performancetests coveren slechts een klein deel van de Waebric grammatica. Niettemin zijn de verschillen zodanig groot dat deze toch significant zijn. De tests met Waebric bestanden die variëren in grootte en complexiteit bevestigen dit, omdat de verschillen hier ook zichtbaar zijn.

Wanneer de resultaten van de metingen tussen beide implementaties worden bezien, valt op dat de “Oslo” implementatie structureel trager is dan de vanilla C# implementatie. Bij grotere Waebric bestanden vormt voornamelijk de parser de bottleneck. De parseertijd van de “Oslo” implementatie neemt exponentieel toe ten opzichte van de grootte van het te parseren bestand. Bij bestanden van duizend regels code duurt het parseren een halve seconde, maar bij 10.000 regels code is dit al ruim 30 seconden.

De checker en interpreter van de “Oslo” implementatie zijn ook structureel trager dan de checker en interpreter van de vanilla C# implementatie. Hierbij zijn de verschillen een stuk kleiner. Bij 10.000 regels code is het verschil tussen de checkers ruim 12 milliseconden en bij de interpreters ruim 100 milliseconden bij de metingen van processortijd.

De invloed van het laden van de grammatica bij de “Oslo” implementatie is redelijk constant. Gemiddeld duurt dit 550 milliseconden. Een deel hiervan wordt veroorzaakt door I/O bij het inlezen van het grammaticabestand. Een ander deel wordt veroorzaakt door het creëren van de parser instantie. Deze tijd is afhankelijk van de grootte en complexiteit van de grammatica. Een kleinere DSL zou dus een kleinere laadtijd van grammatica moeten opleveren.

De hypothese wordt door de resultaten bevestigd. De verschillen bij het parseren zijn veel groter dan verwacht. De vanilla C# parser schaalt lineair bij gelijke complexiteit, terwijl de “Oslo” parser exponentieel schaalt.

Productiviteit

De resultaten laten zien dat de ontwikkeltijd bij de “Oslo” implementatie korter is. Dit verschil is alleen significant zichtbaar bij het ontwikkelen van de lexer en parser in vergelijking tot de grammatica. Bij de checker en interpreter is dit verschil veel kleiner en dus ook minder betrouwbaar. De implementaties zijn na elkaar ontwikkeld. Daardoor kan het ontwikkelen van een interpreter eenvoudiger zijn, omdat hier al sprake is van enige ervaring. Bij het ontwikkelen van de parser is dit effect niet zo sterk, omdat deze op verschillende abstractieniveaus zijn geïmplementeerd. Het verschil is bij de parser dusdanig groot (ruim 40 uur versus ruim 139 uur) dat hier echt sprake is van een verschil in productiviteit, in het voordeel van de “Oslo” implementatie.

De hypothese wordt door de resultaten deels bevestigd. De hypothese stelde dat de “Oslo” implementatie in kortere tijd kan worden gerealiseerd. Bij de realisatie van de parser kan dit op basis van de meetresultaten worden bevestigd. Bij de checker en interpreter is geen significant verschil gemeten.

6.2 Validatie

Representativiteit van Waebric

De representativiteit van Waebric als DSL is niet nader onderzocht. De aanname in dit onderzoek is dat Waebric representatief is, omdat deze is gebaseerd op een andere DSL, namelijk XHTML, uitgebreid met meer complexiteit en functionaliteit.

Invloed onderhoudbaarheid implementaties

Om te voorkomen dat de implementaties op een “verkeerde” manier worden geïmplementeerd is gebruik gemaakt van code reviews. Hierbij is door een C# expert vanuit Microsoft en een externe “Oslo” expert gekeken naar de code en gekeken of C# en “Oslo” op de goede manier is toegepast. Bij deze code reviews is gekeken of C# en “Oslo” op een juiste wijze zijn ingezet en niet onnodig veel complexiteit in de code is geïntroduceerd. Het resultaat van deze code reviews is dat geen onnodige complexiteit in de code is geïntroduceerd in zowel C# als “Oslo”.

Validatie meetgegevens Halstead Effort

De C# 2.0 Metrics tool wordt gebruikt voor het meten van Halstead Effort. Deelcomponenten van de Halstead Effort zijn het aantal operatoren en operanden. De documentatie van de tool beschrijft niet expliciet de definitie van de operatoren en operanden. De definitie van de tool is dat alle tokens in de C# code, die acties specificeren operatoren zijn en dat alle overige tokens operanden zijn. Om dit te controleren zijn twee tests uitgevoerd met stukken C# code en deze zijn door de tool geanalyseerd.

Om de validatie van de tool toe te lichten, volgt een voorbeeld met C# code:

```
using System;

namespace testing
{
    public class test001
    {
        public int test()
        {
            return (x + a) * ( y + a);
        }
    }
}
```

Het voorbeeld is geen semantisch correcte C# code, maar om de werking van de tool te analyseren vormt dit geen probleem. Wanneer bovenstaande code wordt geanalyseerd door de tool dan levert dit de volgende resultaten op:

μ_1 (Aantal unieke operatoren)	9
μ_2 (Aantal unieke operanden)	9
η_1 (Aantal operatoren)	15
η_2 (Aantal operanden)	11

Tabel 9: Resultaten test tool

De testresultaten kunnen worden gecontroleerd aan de hand van de definitie van operatoren en operanden. Alle namen in de C# code zijn operanden, want deze specificeren geen actie. In het voorbeeld zijn dit 8 operanden (System, testing, test001, test, x, a, y, a). Typenamen specificeren ook geen actie, dus worden ook als operand meegeteld. In dit voorbeeld is int het enige type en levert dus in totaal 9 operanden op. Het karakter ; specificeert eveneens geen actie. In het voorbeeld komen op twee plaatsen ; karakters voor, dat een totaal van 11 operanden oplevert. Dit komt overeen met het berekende aantal operanden door de tool.

De operatoren specificeren acties in de C# code. Alle C# keywords die geen type zijn, specificeren acties en worden als operator meegeteld. In dit voorbeeld zijn dit in totaal 6 keywords (using, namespace, public, class, public, return). Verder worden blokken ({}), haakjes (()) en operatoren op expressies meegeteld (+,-, *, \, etc.). Dit levert in totaal 9 operatoren op ({}, {}, (), {}, (), +, *, (), +). In totaal zijn dit 15 operatoren dat overeenkomt met het berekende aantal operatoren door de tool.

Bij het aantal unieke operatoren en operanden worden alle operatoren en operanden eenmalig geteld. Dit levert in totaal 9 unieke operanden en operatoren op, dat overeenkomt met de resultaten van de tool. Op basis van deze test is in dit onderzoek aangenomen dat de tool werkt volgens de beschrijving en correcte resultaten berekent.

Validatie productiviteitsmeting

De productiviteit is gemeten door het meten van de ontwikkeltijd van beide implementaties. Deze ontwikkeltijd is subjectief, aangezien dit afhankelijk is van programmeerervaring, voorkennis over compilerbouw en leercurve tijdens het ontwikkelen. Dit onderzoek is uitgevoerd zonder kennis van programmeren in C# en zeer beperkte kennis van compilerbouw. Op basis hiervan is in dit onderzoek aangenomen dat de resultaten van de productiviteitsmeting beperkt betrouwbaar zijn. Alleen zeer grote verschillen (meer dan 20 uur verschil) zijn meegenomen in de evaluatie.

7 Conclusie en toekomst

Dit hoofdstuk interpreteert de resultaten van dit onderzoek en geeft antwoorden op de onderzoeksvragen in de vorm van conclusies. Tot slot zal worden ingegaan op mogelijk toekomstig onderzoek.

7.1 Conclusie en discussie

OV1: Verhoogt de inzet van “Oslo” de kwaliteit van een DSL implementatie?
OV1.1: Hoe verhoudt de onderhoudbaarheid van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?
OV1.2: Hoe verhoudt de performance van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?
OV1.3: Hoe verhoudt de functionaliteit van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?
OV1.4: Hoe verhoudt de productiviteit van het ontwikkelen van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?

OV1.1: Hoe verhoudt de onderhoudbaarheid van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?

De onderhoudbaarheid is gemeten door van beide implementaties een set van software metrieken te meten. Uit deze metingen blijkt dat bij gebruik van “Oslo” alleen de onderhoudbaarheid van de parser aantoonbaar verbetert. De onderhoudbaarheid van zowel de checker als de interpreter is bij het gebruik van “Oslo” iets verslechterd ten opzichte van de vanilla C# implementatie. Dit leidt tot de conclusie dat “Oslo” alleen bij het ontwikkelen van een parser voor een DSL een aantoonbare verbeterde onderhoudbaarheid geeft.

OV1.2: Hoe verhoudt de performance van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?

Van beide implementaties is de performance gemeten door zeven Waebric bestanden met verschillende grootte en complexiteit te testen. Daarbij is de executietijd gemeten per component. Op basis van de resultaten kan worden geconcludeerd dat de “Oslo” parser significant trager is dan de vanilla C# parser. Deze conclusie geldt ook voor de checker en interpreter van de “Oslo” implementatie, maar in aanzienlijk mindere mate ten opzichte van de parser.

OV1.3: Hoe verhoudt de functionaliteit van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?

De functionaliteit is gemeten door het uitvoeren van acceptatietests bij beide implementaties. Beide implementaties zijn geslaagd voor al deze acceptatietests, maar de acceptatietests coveren niet de volledige Waebric grammatica. In één specifiek geval, waarbij token en syntax regels elkaar overlappen levert dit problemen op. Het probleem kan worden omzeild door het aanpassen van de grammatica van de DSL. Dit knelpunt kan zich ook mogelijk manifesteren in grammatica’s van andere DSL’s en het staat niet zonder meer vast dat dit knelpunt altijd kan worden opgelost door de grammatica van de DSL aan te passen.

OV1.4: Hoe verhoudt de productiviteit van het ontwikkelen van de vanilla C# implementatie zich tot die van de “Oslo” implementatie?

De productiviteit is gemeten door het meten van de ontwikkeltijd van beide implementaties. De resultaten van deze productiviteitsmeting in dit onderzoek bevestigen in relatie tot algemeen onderzoek dat de ontwikkeling van de parser bij het gebruik van een DSL (“M” in dit onderzoek) korter is dan bij het gebruik van een GPL (C# in dit onderzoek). Bij de checker en interpreter is geen significant verschil in productiviteit gemeten.

OV1: Verhoogt de inzet van “Oslo” de kwaliteit van een DSL implementatie?

Op basis van de antwoorden op de vier onderzoeksvragen, kan niet zonder meer worden gesteld dat de inzet van “Oslo” de kwaliteit van een DSL implementatie verhoogt.

7.2 Toekomst

In dit onderzoek zijn vier attributen onderzocht voor twee verschillende implementatiestrategieën voor het implementeren van een DSL. Het is aan te bevelen om aanvullend onderzoek te verichtten naar de voor- en nadelen bij het gebruik van compiler generatoren voor de constructie van een DSL. Hierbij is het ook wenselijk om meer attributen (bijvoorbeeld gebruiksvriendelijkheid en betrouwbaarheid) te betrekken in het onderzoek. Mogelijk kan dit ingebed worden in het grotere onderzoek bij het CWI.

In dit onderzoek is de tool “Oslo” van Microsoft gebruikt. Deze tool bevindt zich momenteel in een ontwikkelstadium en is dus nog niet volledig uitontwikkeld. Een nieuwe vergelijking in de toekomst kan mogelijk andere resultaten opleveren. Vooral bij de attributen functionaliteit en performance zou dit mogelijk in de toekomst tot andere uitkomsten kunnen leiden, omdat aan deze aspecten nog wordt gewerkt.

Referenties

Aho, A. V., M. S. Lam, et al. (2007). Compilers: Principles, Techniques and Tools 2th edition, Addison Wesley.

Arnold, B. R. T., A. v. Deursen, et al. (1995). "An Algebraic Specification of a Language for Describing Financial Products."

Deursen, A. v., P. Klint, et al. (2000). "Domain Specific Languages: An Annotated Bibbliography." ACM Sigplan Notices **35**: 26-36.

Gamma, E., R. Helm, et al. (1994). Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley.

Garshol, L. M. (2003). "BNF and EBNF: What are they and how do they work?" Retrieved 7-26-2009, from http://tur-www1.massey.ac.nz/~kahawick/159331/BNF+EBNF-Garshol.pdf.

Grimm, R. (2007). "Declarative Syntax Tree Engineering."

Halstead, M. H. (1977). Elements of Software Science, Elsevier.

Heering, J., P. R. H. Hendriks, et al. (1989). "The syntax definition formalism SDF - reference manual." SIGPLAN Notices.

Hermans, F. (2009). Oslo, het nieuwe modelleerplatform van Microsoft. .NET Magazine.

IEEE (1993). Glossary of Software Engineering Terminology. Software Engineering Standards Collection.

Kieburtz, R. B., J. M. Bell, et al. (1996). "A Software Engineering Experiment in Software Component Generation." International Conference on Software Engineering(0270-5257): 542.

Klint, P. and A. v. Deursen (1998). "Little Languages: Little maintenance?" Journal of Software Maintenance **10**: 75-92.

McCabe, T. J. (1976). "A Complexity Measure." IEEE Transactions on software engineering **SE-2**(4).

Mernik, M., J. Heering, et al. (2005). "When and how to develop domain-specific languages." ACM computing surveys **37**(4): 316-344.

Microsoft. (2009, May 6, 2009). "The "Oslo" Modeling Language Specification." Retrieved 6-1-2009, from http://download.microsoft.com/download/D/A/B/DAB9E2D8-3A27-4BA7-BE66-8600EE4E33B0/M_Language_Specification.pdf.

Moore, R. C. (2000). "Removing Left Recursion from Context-Free Grammars." 6th Applied Natural Language Processing Conference: 249-255.

MSDN. (2009). ""M" Programming Guide." Retrieved 6-1-2009, from http://msdn.microsoft.com/en-us/library/dd129568(VS.85).aspx.

Oman, P. and J. Hagemeister (1994). "Construction and testing of polynomials predicting software maintainability." Journal of Systems and Software **24**(3): 251-266.

Power, J. F. and B. A. Malloy (2004). "A metrics suite for gammar-based software." Software Maintenance and Evolution: Research and Practice **16**(6): 405-426.

Rosenberg, J. (1997). "Some Misconceptions About Lines of Code." Fourth International Software Metrics Symposium (METRICS'97): 137.

Tomita, M. (1991). Generalized LR parsing, Kluwer.

van der Storm, T. (2009). "Waebric: a Little Language for Markup Generation."

Welker, K. D. (2001). "The Software Maintainability Index Revisited." The Journal of Defense Software Engineering.

Bijlage A: Resultaten acceptatietests

Deze bijlage bevat het complete overzicht van de acceptatietests.

In Tabel 10 zijn de testresultaten van de acceptatietests voor de vanilla C# implementatie weergegeven. De resultaten bestaan uit de uitgevoerde tests met bestandsnaam, eventuele errors en warnings die zijn opgetreden bij het uitvoeren van de test met de melding of dit volgens specificatie is en de status van de XHTML output.

Test	Errors/Warnings	XHTML Output
imp.wae		OK
cycle.wae		OK
trans.wae		OK
transcyclic.wae		OK
test001.wae		OK
test002.wae		OK
test003.wae		OK
test004.wae		OK
test005.wae		OK
test006.wae		OK
test007.wae		OK
test008.wae		OK
test009.wae		OK
test010.wae		OK
test011.wae		OK
test012.wae		OK
test013.wae		OK
test014.wae		OK
test015.wae		OK
test016.wae		OK
test017.wae		OK
test018.wae		OK
test019.wae		OK
test020.wae		OK
test021.wae		OK
test022.wae		OK
test023.wae		OK
test024.wae		OK
test025.wae		OK
test026.wae		OK
test027.wae		OK

test028.wae		OK
test029.wae		OK
test030.wae		OK
test031.wae		OK
test032.wae		OK
test033.wae		OK
test034.wae		OK
test035.wae		OK
test036.wae		OK
test037.wae		OK
test038.wae		OK
test039.wae		OK
test040.wae		OK
test041.wae	Function f called with wrong number of arguments/OK	OK
test042.wae	Function f called with wrong number of arguments/OK	OK
test043.wae		OK
test044.wae		OK
test045.wae		OK
test046.wae		OK
test047.wae		OK
test048.wae		OK
test049.wae		OK
test050.wae		OK
test051.wae	Variable a used but not declared/OK	OK
test052.wae		OK
test053.wae	Variable b used but not declared/OK	OK
test054.wae	Variable b used but not declared/OK	OK
test055.wae		OK
test056.wae		OK
test057.wae		OK
test058.wae		OK
test059.wae		OK
test060.wae		OK
test061.wae		OK
test062.wae		OK
test063.wae		OK
test064.wae	Function f called with wrong number of arguments/OK	OK
test065.wae	Function f called with wrong number of arguments/OK	OK
test066.wae		OK
test067.wae	Function f called with wrong number of arguments/OK	OK

test068.wae		OK
test069.wae	Function f called with wrong number of arguments/OK	OK
test070.wae		OK
test071.wae		OK
test072.wae		OK
test073.wae		OK
test074.wae		OK
test075.wae		OK
test076.wae		OK
test077.wae		OK
test078.wae	Function g called with wrong number of arguments/OK	OK
test079.wae	Variable a used but not declared/OK	OK
test080.wae	Variable a used but not declared/OK	OK
test081.wae		OK
test082.wae		OK
test083.wae		OK
test084.wae		OK
test085.wae		OK
test086.wae		OK
test087.wae	Unexpected token found: end at line 8/OK	OK
test088.wae		OK
test089.wae		OK
test090.wae		OK
test091.wae		OK
test092.wae		OK
test093.wae		OK
test094.wae		OK
test095.wae		OK
test096.wae		OK
test097.wae		OK
test098.wae	Function g called with wrong number of arguments/OK	OK
test099.wae		OK
test100.wae		OK

Tabel 10: Resultaten acceptatietests vanilla C# implementatie

In Tabel 11 zijn de resultaten van de acceptatietests weergeven van de “Oslo” implementatie.

Test	Errors/Warnings	XHTML Output
imp.wae		OK
cycle.wae		OK
trans.wae		OK
transcyclic.wae		OK
test001.wae		OK
test002.wae		OK
test003.wae		OK
test004.wae		OK
test005.wae		OK
test006.wae		OK
test007.wae		OK
test008.wae		OK
test009.wae		OK
test010.wae		OK
test011.wae		OK
test012.wae		OK
test013.wae		OK
test014.wae		OK
test015.wae		OK
test016.wae		OK
test017.wae		OK
test018.wae		OK
test019.wae		OK
test020.wae		OK
test021.wae		OK
test022.wae		OK
test023.wae		OK
test024.wae		OK
test025.wae		OK
test026.wae		OK
test027.wae		OK
test028.wae		OK
test029.wae		OK
test030.wae		OK
test031.wae		OK
test032.wae		OK
test033.wae		OK

test034.wae		OK
test035.wae		OK
test036.wae		OK
test037.wae		OK
test038.wae		OK
test039.wae		OK
test040.wae		OK
test041.wae	Function f called with wrong number of arguments	OK
test042.wae	Function f called with wrong number of arguments	OK
test043.wae		OK
test044.wae		OK
test045.wae		OK
test046.wae		OK
test047.wae		OK
test048.wae		OK
test049.wae		OK
test050.wae		OK
test051.wae	Variable a used but not declared	OK
test052.wae		OK
test053.wae	Variable b used but not declared	OK
test054.wae	Variable b used but not declared	OK
test055.wae		OK
test056.wae		OK
test057.wae		OK
test058.wae		OK
test059.wae		OK
test060.wae		OK
test061.wae		OK
test062.wae		OK
test063.wae		OK
test064.wae	Function f called with wrong number of arguments	OK
test065.wae	Function f called with wrong number of arguments	OK
test066.wae		OK
test067.wae	Function f called with wrong number of arguments	OK
test068.wae		OK
test069.wae	Function f called with wrong number of arguments	OK
test070.wae		OK
test071.wae		OK
test072.wae		OK
test073.wae		OK

test074.wae		OK
test075.wae		OK
test076.wae		OK
test077.wae		OK
test078.wae	Function g called with wrong number of arguments	OK
test079.wae	Variable a used but not declared	OK
test080.wae	Variable a used but not declared	OK
test081.wae		OK
test082.wae		OK
test083.wae		OK
test084.wae		OK
test085.wae		OK
test086.wae		OK
test087.wae	..\..\test087.wae(7,1): error 5007: Token EndKeyword with text "end" unexpected.	OK
test088.wae	..\..\test088.wae(7,9): error 5007: Token EchoKeyword with text "echo" unexpected.	OK
test089.wae	..\..\test089.wae(6,20): error 5003: "/>\";\ne" is invalid for token Filename. Character "e" was unexpected. "." expected. ..\..\test089.wae(6,20): error 5003: "/>\";\ne" is invalid for token Filename. Character "e" was unexpected. "." expected. ..\..\test089.wae(8,1): error 5006: Reached end of buffer.	OK
test090.wae		OK
test091.wae		OK
test092.wae		OK
test093.wae		OK
test094.wae		OK
test095.wae		OK
test096.wae		OK
test097.wae		OK
test098.wae	Function g called with wrong number of arguments	OK
test099.wae		OK
test100.wae		OK

Tabel 11: Resultaten acceptatietests "Oslo" implementatie

Het eindresultaat is weergegeven in Tabel 12.

Vanilla C# implementatie	“Oslo” implementatie
100% (104/104)	100% (104/104)

Tabel 12: Eindresultaat acceptatietests

Bijlage B: Resultaten metingen onderhoudbaarheid

In Tabel 13 zijn de resultaten van de metingen voor onderhoudbaarheid weergegeven van de vanilla C# implementatie. De legenda van afkortingen is weergegeven in Tabel 16.

Metric	LexerCore	LexerToken	ParserCore	PaserAST	Checker	Interpreter	Utils
NCLOC	588	151	1404	3357	298	1256	404
MLOC	513	105	1237	2348	197	1130	248
MCC	3,21	1,37	3,12	1,07	2,14	2,99	2,04
E	1304337	34516	1645219	692661	134731	3474242	224344
V	16897	1821	43000	43253	5524	40571	6651
MI3	109	126	106	133	121	108	116
#M	42	19	91	535	27	85	27
AMS	12	6	14	4	7	13	9
#C	3	3	11	80	8	6	4
Avg-E	31056	1817	18079	1295	4990	40873	8309

Tabel 13: Resultaten metingen onderhoudbaarheid vanilla C# implementatie

De metingen van “Oslo” zijn opgesplitst in twee afzonderlijke metingen, namelijk de grammatica die is weergegeven in Tabel 14 en de C# code die is weergegeven in Tabel 15. De legende van afkortingen is weergegeven in Tabel 16.

Metriek	Grammar
NCLOG	335
WMC	257
MCC	2,11
E	50873
MI3	117
#Productions	122
AVS	4,25
V	6033
u1	7
u2	215
n1	256
n2	518
#T	222
#NT	296
Decision Points	135
Non-MCC operators	121
#OP	256
#OB	401
#UT	93

Tabel 14: Resultaten metingen onderhoudbaarheid grammatica “Oslo” implementatie

Metric	Parser	Checker	Interpreter	Utils
NCLOC	51	394	1576	415
MLOC	32	303	1463	266
MCC	1,25	2,54	3,82	2,08
V	11348	450861	6117914	270731
E	119	110	103	114
MI3	4	26	89	28
#M	8	12	16	10
AMS	1	8	6	4
#C	51	394	1576	415
Avg-E	2837	17341	68741	9669

Tabel 15: Resultaten metingen onderhoudbaarheid C# code "Oslo" implementatie

Afkorting	Betekenis
NCLOC	Non commented Lines of code
MLOC	NLOC in methodes
MCC	McCabe Complexity
WMC	Weighted Methods per Class
E	Halstead Effort
MI3	Maintainability Index
#M	Aantal methoden
AMS	Gemiddeld aantal NCLOC per methode
#C	Aantal klassen
Avg-E	Gemiddelde halstead effort per methode
V	Halstead Volume
n1	Operator Occurence
n2	Operarand Occurence
u1	Aantal unieke operatoren
u2	Aantal unieke operanden
AVS	Gemiddelde Right Hand Size
#T	Aantal terminals
#NT	Aantal non-terminals
Decision Points	Aantal {|,^,+,*,?}
Non-MCC operators	Aantal {..,=>}
#OP	Aantal operatoren
#OB	Aantal operanden
#UT	Aantal unieke terminals

Tabel 16: Legenda afkortingen metingen onderhoudbaarheid

Bijlage C: Performance meetresultaten

In Tabel 17 en Tabel 18 zijn de resultaten van de performance metingen weergegeven in processortijd.

	Menus		LDTA		Lava	
Component	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**
GrammarLoader (ms)	0	547,05	0	552,5	0	550,68
Parser (ms)	0,45	9,15	1,96	154,77	8,59	318,27
Checker (ms)	0,24	0,31	1,56	1,99	8,10	641,11
Interpreter (ms)	2,37	4,08	19,18	27,98	50,03	375,10
Totaal (ms)	3,06	560,59	22,7	737,24	66,72	1885,16

Tabel 17: Resultaten performancemetingen deel 1 processortijd

Component	**1k**		**2.5k**		**5k**		**7.5k**		**10k**	
	vanilla C#	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**
Grammar Loader (ms)	0	508,93	0	542,01	0	541,45	0	549,50	0	552,84
Parser (ms)	4,02	429,14	8,80	2205,29	20,58	8039,62	29,96	17880,15	38,92	31154,59
Checker (ms)	0,37	1,45	0,94	3,55	1,47	7,36	1,89	10,72	2,86	14,60
Interpreter (ms)	28,07	34,64	70,95	86,00	146,31	174,22	229,20	325,65	303,68	405,87
Totaal (ms)	32,46	974,16	80,69	2836,85	168,36	8762,65	261,05	18766,02	345,46	32127,9

Tabel 18: Resultaten performancemetingen deel 2 processortijd

In Tabel 19 en Tabel 20 zijn de resultaten weergegeven van de performancemetingen in kloktijd.

	Menus		LDTA		Lava	
Component	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**
GrammarLoader (ms)	0	575,02	0	563,84	0	556,31
Parser (ms)	0,48	10,47	2,75	170,03	8,47	337,72
Checker (ms)	0,18	0,34	1,5	2,36	8,92	651,43
Interpreter (ms)	4,14	6,04	24	33,78	85,50	446,32
Totaal (ms)	4,8	591,87	28,25	770,01	102,89	1991,78

Tabel 19: Resultaten performancemetingen deel 1 kloktijd

Component	**1k**		**2.5k**		**5k**		**7.5k**		**10k**	
	vanilla C#	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**	**vanilla C#**	**"Oslo"**
Grammar Loader (ms)	0	541,36	0	552,52	0	551,89	0	555,59	0	564,00
Parser (ms)	4,11	438,98	9,70	2374,32	21,89	8643,29	30,78	18985,77	40,26	33405,32
Checker (ms)	0,38	1,58	0,93	3,91	1,39	7,80	2,11	11,32	2,93	16,90
Interpreter (ms)	32,88	42,10	78,17	95,86	156,67	210,55	243,19	375,85	321,35	430,37
Totaal (ms)	37,37	1024,02	88,8	3026,61	179,95	9413,53	276,08	19928,53	364,54	34416,59

Tabel 20: Resultaten performancemetingen deel 2 kloktijd

Bijlage D: Resultaten productiviteitsmeting

In Tabel 21 zijn de metingen van de ontwikkeltijd van de vanilla C# implementatie weergegeven.

Datum:	Begintijd:	Eindtijd:	Tijd:	Module:
18-5-2009	13:30	14:15	0:45	Lexer
18-5-2009	14:30	15:15	0:45	Lexer
18-5-2009	16:10	16:30	0:20	Lexer
20-5-2009	9:50	11:20	1:30	Lexer
20-5-2009	13:15	15:10	1:55	Lexer
20-5-2009	16:00	16:15	0:15	Lexer
25-5-2009	9:30	11:00	1:30	Lexer
25-5-2009	11:15	12:00	0:45	Lexer
25-5-2009	13:45	16:30	2:45	Lexer
26-5-2009	9:30	10:00	0:30	Lexer
26-5-2009	11:00	12:15	1:15	Lexer
26-5-2009	16:00	17:00	1:00	Lexer
27-5-2009	11:00	12:00	1:00	Lexer
27-5-2009	13:30	14:45	1:15	Lexer
29-5-2009	8:45	9:30	0:45	Parser
2-6-2009	10:15	11:30	1:15	Parser
2-6-2009	13:15	15:15	2:00	Parser
2-6-2009	16:00	17:00	1:00	Parser
3-6-2009	10:15	12:45	2:30	Parser
8-6-2009	7:15	9:45	2:30	Parser
8-6-2009	10:00	12:00	2:00	Parser
8-6-2009	12:45	14:00	1:15	Parser
8-6-2009	14:30	15:30	1:00	Parser
9-6-2009	8:30	12:30	4:00	Lexer
9-6-2009	13:15	14:30	1:15	Lexer
9-6-2009	15:00	16:15	1:15	Lexer
10-6-2009	8:30	10:00	1:30	Lexer
10-6-2009	10:30	12:15	1:45	Lexer
10-6-2009	13:45	15:30	1:45	Parser
10-6-2009	15:45	16:30	0:45	Parser
11-6-2009	9:00	10:30	1:30	Parser
11-6-2009	11:15	13:00	1:45	Parser
11-6-2009	14:00	15:00	1:00	Parser
11-6-2009	16:15	16:30	0:15	Parser
15-6-2009	9:30	11:50	2:20	Parser

15-6-2009	13:00	15:20	2:20	Parser
15-6-2009	16:00	16:50	0:50	Parser
16-6-2009	9:30	11:30	2:00	Parser
16-6-2009	12:15	12:40	0:25	Parser
16-6-2009	13:30	14:30	1:00	Parser
16-6-2009	14:45	17:10	2:25	Parser
17-6-2009	10:00	12:10	2:10	Parser
17-6-2009	12:35	13:00	0:25	Parser
17-6-2009	14:00	15:15	1:15	Parser
18-6-2009	9:30	10:50	1:20	Parser
18-6-2009	11:00	12:30	1:30	Parser
18-6-2009	13:30	15:50	2:20	Parser
18-6-2009	16:00	16:20	0:20	Parser
18-6-2009	16:35	17:05	0:30	Parser
19-6-2009	9:30	12:30	3:00	Parser
19-6-2009	15:10	15:50	0:40	Parser
22-6-2009	9:45	11:15	1:30	Parser
22-6-2009	11:20	12:20	1:00	Parser
22-6-2009	12:30	13:00	0:30	Parser
22-6-2009	13:45	15:00	1:15	Parser
22-6-2009	15:10	16:45	1:35	Parser
23-6-2009	9:45	11:00	1:15	Parser
23-6-2009	11:15	13:00	1:45	Parser
23-6-2009	13:45	15:15	1:30	Parser
23-6-2009	15:40	17:10	1:30	Parser
24-6-2009	9:30	11:00	1:30	Parser
24-6-2009	11:15	13:20	2:05	Parser
24-6-2009	14:20	14:50	0:30	Parser
25-6-2009	11:00	13:00	2:00	Lexer
25-6-2009	14:15	15:00	0:45	Lexer
25-6-2009	15:25	16:15	0:50	Lexer
25-6-2009	16:15	18:00	1:45	Parser
25-6-2009	19:00	20:00	1:00	Lexer
27-6-2009	10:50	11:20	0:30	Lexer
27-6-2009	12:15	12:30	0:15	Lexer
27-6-2009	13:30	14:50	1:20	Lexer
27-6-2009	15:55	17:15	1:20	Parser
28-6-2009	12:20	13:30	1:10	Parser
28-6-2009	13:50	15:30	1:40	Parser
28-6-2009	16:10	17:10	1:00	Parser

29-6-2009	10:20	10:50	0:30	Parser
30-6-2009	10:50	11:20	0:30	Checker
30-6-2009	13:45	14:45	1:00	Utils
1-7-2009	9:15	10:55	1:40	Utils
1-7-2009	11:00	11:15	0:15	Utils
1-7-2009	11:30	12:00	0:30	Parser
1-7-2009	12:15	12:55	0:40	Parser
1-7-2009	14:00	14:50	0:50	Utils
1-7-2009	15:15	16:15	1:00	Checker
2-7-2009	10:15	11:15	1:00	Checker
2-7-2009	11:30	12:30	1:00	Parser
2-7-2009	13:45	16:15	2:30	Parser
4-7-2009	10:45	12:00	1:15	Parser
4-7-2009	13:50	15:15	1:25	Parser
4-7-2009	15:45	16:25	0:40	Parser
4-7-2009	16:40	17:35	0:55	Parser
5-7-2009	12:15	13:15	1:00	Parser
6-7-2009	10:10	10:40	0:30	Parser
6-7-2009	11:15	12:30	1:15	Parser
6-7-2009	13:35	14:15	0:40	Parser
7-7-2009	9:45	10:45	1:00	Parser
7-7-2009	11:00	12:45	1:45	Parser
7-7-2009	14:00	16:00	2:00	Parser
7-7-2009	16:25	17:00	0:35	Parser
8-7-2009	8:50	10:35	1:45	Checker
8-7-2009	11:00	12:40	1:40	Checker
8-7-2009	13:50	14:50	1:00	Checker
8-7-2009	15:10	16:00	0:50	Checker
9-7-2009	8:45	10:00	1:15	Checker
9-7-2009	10:15	12:30	2:15	Checker
9-7-2009	13:50	15:35	1:45	Checker
10-7-2009	9:30	11:50	2:20	Parser
10-7-2009	13:30	15:50	2:20	Parser
10-7-2009	16:50	17:20	0:30	Checker
13-7-2009	8:00	9:20	1:20	Interpreter
13-7-2009	9:40	10:40	1:00	Interpreter
13-7-2009	10:55	11:30	0:35	Interpreter
13-7-2009	11:45	13:00	1:15	Interpreter
13-7-2009	13:55	15:30	1:35	Interpreter
15-7-2009	12:30	13:10	0:40	Interpreter

15-7-2009	13:45	14:10	0:25	Interpreter
15-7-2009	14:35	15:00	0:25	Interpreter
16-7-2009	9:30	10:30	1:00	Interpreter
16-7-2009	11:00	12:15	1:15	Interpreter
16-7-2009	13:45	15:30	1:45	Interpreter
17-7-2009	9:45	10:45	1:00	Interpreter
17-7-2009	11:20	12:00	0:40	Interpreter
17-7-2009	12:20	12:45	0:25	Interpreter
17-7-2009	14:00	15:15	1:15	Interpreter
17-7-2009	15:45	17:00	1:15	Interpreter
20-7-2009	7:45	10:10	2:25	Interpreter
20-7-2009	10:30	11:40	1:10	Interpreter
20-7-2009	12:00	13:00	1:00	Interpreter
20-7-2009	13:45	15:00	1:15	Interpreter
21-7-2009	8:15	10:00	1:45	Interpreter
21-7-2009	10:30	12:00	1:30	Interpreter
21-7-2009	12:15	13:00	0:45	Interpreter
21-7-2009	14:10	16:00	1:50	Interpreter
22-7-2009	8:15	11:15	3:00	Interpreter
22-7-2009	11:30	12:50	1:20	Interpreter
22-7-2009	13:40	15:10	1:30	Interpreter
22-7-2009	15:30	17:00	1:30	Interpreter
23-7-2009	8:10	10:30	2:20	Interpreter
23-7-2009	10:45	12:20	1:35	Interpreter
23-7-2009	12:35	13:00	0:25	Interpreter
23-7-2009	14:00	17:20	3:20	Interpreter
23-7-2009	17:30	18:10	0:40	Interpreter
18-8-2009	10:00	10:45	0:45	Interpreter
2-9-2009	14:00	16:00	2:00	Interpreter
2-9-2009	16:20	19:00	2:40	Interpreter

Tabel 21: Resultaten meting ontwikkeltijd vanilla C# implementatie

In Tabel 22 zijn de metingen van de ontwikkeltijd van de “Oslo” implementatie weergegeven.

Datum:	Begintijd:	Eindtijd:	Tijd:	Module:
24-7-2009	8:15	10:40	2:25	Grammar
24-7-2009	11:15	12:10	0:55	Grammar
24-7-2009	13:15	14:20	1:05	Grammar
24-7-2009	14:40	16:50	2:10	Grammar
27-7-2009	8:15	10:15	2:00	Grammar
27-7-2009	10:30	11:50	1:20	Grammar
27-7-2009	12:00	13:00	1:00	Grammar
27-7-2009	13:30	15:15	1:45	Grammar
29-7-2009	8:45	10:40	1:55	Grammar
29-7-2009	11:00	12:30	1:30	Grammar
29-7-2009	12:45	13:10	0:25	Grammar
29-7-2009	14:00	16:05	2:05	Grammar
30-7-2009	9:15	10:40	1:25	Grammar
30-7-2009	11:15	12:00	0:45	Grammar
30-7-2009	13:00	15:20	2:20	Grammar
30-7-2009	16:25	16:50	0:25	Grammar
31-7-2009	9:45	11:15	1:30	Grammar
31-7-2009	11:25	13:00	1:35	Grammar
31-7-2009	14:00	15:15	1:15	Grammar
3-8-2009	10:45	12:00	1:15	Grammar
3-8-2009	13:45	14:30	0:45	Grammar
4-8-2009	14:15	16:00	1:45	Grammar
5-8-2009	10:15	11:35	1:20	Grammar
5-8-2009	11:50	12:15	0:25	Grammar
5-8-2009	14:00	15:30	1:30	Grammar
6-8-2009	9:15	11:30	2:15	Grammar
6-8-2009	13:30	15:00	1:30	Checker
7-8-2009	11:00	12:15	1:15	Parser
7-8-2009	13:15	14:00	0:45	Parser
7-8-2009	14:00	14:45	0:45	Checker
10-8-2009	9:15	10:45	1:30	Checker
10-8-2009	11:15	12:50	1:35	Checker
10-8-2009	13:15	16:15	3:00	Checker
11-8-2009	9:45	11:00	1:15	Checker
11-8-2009	11:30	12:35	1:05	Checker
11-8-2009	12:40	13:00	0:20	Checker
11-8-2009	14:00	14:45	0:45	Checker

11-8-2009	15:10	17:10	2:00	Checker
12-8-2009	10:20	11:15	0:55	Interpreter
12-8-2009	11:45	13:00	1:15	Interpreter
12-8-2009	14:00	17:00	3:00	Interpreter
13-8-2009	9:45	13:00	3:15	Interpreter
13-8-2009	13:45	16:15	2:30	Interpreter
14-8-2009	9:30	10:15	0:45	Interpreter
14-8-2009	10:20	13:00	2:40	Interpreter
14-8-2009	14:00	16:20	2:20	Interpreter
17-8-2009	9:15	10:45	1:30	Interpreter
17-8-2009	11:10	13:00	1:50	Interpreter
17-8-2009	13:55	16:15	2:20	Interpreter
17-8-2009	16:30	17:00	0:30	Interpreter
18-8-2009	11:00	11:45	0:45	Parser
2-9-2009	9:00	13:00	4:00	Interpreter
16-7-2009	9:30	10:30	1:00	Interpreter
16-7-2009	11:00	12:15	1:15	Interpreter
16-7-2009	13:45	15:30	1:45	Interpreter
23-7-2009	8:10	10:30	2:20	Interpreter
23-7-2009	10:45	12:20	1:35	Interpreter
23-7-2009	12:35	13:00	0:25	Interpreter
23-7-2009	14:00	17:20	3:20	Interpreter
23-7-2009	17:30	18:10	0:40	Interpreter
30-6-2009	13:45	14:45	1:00	Utils
1-7-2009	9:15	10:55	1:40	Utils
1-7-2009	11:00	11:15	0:15	Utils
1-7-2009	14:00	14:50	0:50	Utils

Tabel 22: Resultaten meting ontwikkeltijd "Oslo" implementatie