

Implementering av en Lua parser

Oskar Schöldström

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informations- och medieteknik
Identifikationsnummer:	1234
Författare:	Oskar Schöldström
Arbetets namn:	Implementering av en Lua parser
Handledare (Arcada):	...
Uppdragsgivare:	
Sammandrag:	
a a	
Nyckelord:	parser, javascript, lua, lexer, tolk
Sidantal:	31
Språk:	Svenska
Datum för godkännande:	

DEGREE THESIS	
Arcada	
Degree Programme:	Information and Media Technology
Identification number:	1234
Author:	Oskar Schöldström
Title:	Implementing a Lua parser
Supervisor (Arcada):	...
Commissioned by:	
Abstract:	
a a	
Keywords:	parser, javascript, lua, lexer, lexical analysis, tokenizer, scanner, lexical analyzer, syntactic analysis
Number of pages:	31
Language:	Swedish
Date of acceptance:	

OPINNÄYTE	
Arcada	
Koulutusohjelma:	Informaatio- ja mediatekniikka
Tunnistenumero:	1234
Tekijä:	Oskar Schöldström
Työn nimi:	Implementing a Lua parser
Työn ohjaaja (Arcada):	...
Toimeksiantaja:	
Tiivistelmä: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas molestie, nisi sed consectetur tincidunt, ante libero euismod nulla, eget commodo ligula purus eu nibh. Ut enim risus, congue sed viverra ut, tempor vel velit. Quisque consectetur tincidunt scelerisque. Donec nec mollis leo. Donec laoreet purus a massa ultrices egestas. Ut quis neque nisi. In hac habitasse platea dictumst. Suspendisse suscipit, ante eget venenatis molestie, elit nulla aliquam ligula, sit amet consectetur tortor magna eu ipsum. Pellentesque tempor tincidunt arcu, eget ultrices lorem commodo a.	
Avainsanat:	parser, javascript, lua, lexer, lexical analysis, tokenizer, scanner, lexical analyzer, syntactic analysis
Sivumäärä:	31
Kieli:	Ruotsi
Hyväksymispäivämäärä:	

INNEHÅLL

1	Inledning	8
1.1	Syfte och målsättning	8
1.2	Utförande	8
2	Språk teori	9
2.1	Backus-Naur Form	9
2.2	Chomskyhierarkin	10
2.2.1	Reguljär grammatik	11
2.2.2	Kontextfri grammatik	12
3	Parsning	13
3.1	Lexikal analys	14
3.2	Syntastisk analys	14
3.3	Parser typer	15
3.3.1	LL parser	15
3.3.2	LR parser	15
3.3.3	Recursive descent parser	15
3.4	Tekniker	16
3.4.1	Left factoring	16
3.4.2	Prediction	16
4	Implementation	17
4.1	Lua	17
4.1.1	Funktionalitet	17
4.1.2	Datastrukturer	17
4.1.3	Grammatik	17
4.2	Design	17
4.3	Modifiering av grammatik	17
4.3.1	Lexer	17
4.3.2	Parser	18
4.3.3	Uttrycks parser	18
4.3.4	Resultat träd	18
4.4	JavaScript kodmönster	19
4.4.1	Imperativt paradigm	19
4.4.2	Modul laddare	19
5	Prestanda optimering	20
5.1	Insamling av data	20
5.2	Analysering av data för V8	20
5.2.1	Sampling	20
5.3	Optimerings tekniker för V8	20
5.3.1	Undvikandet av bailouts	21

5.3.2	<i>Inlining</i>	22
5.3.3	<i>Teckenkoder istället för reguljära uttryck</i>	23
5.3.4	<i>Optimering av sträng funktioner</i>	23
5.3.5	<i>Garbage Collection</i>	24
5.4	Resultat	25
6	Kvalitets analys	27
6.1	Mätning	27
6.1.1	<i>Test</i>	27
6.1.2	<i>Kodtäckning</i>	27
6.1.3	<i>Kod komplexitet</i>	27
6.1.4	<i>Benchmark</i>	27
6.2	Automation	27
6.2.1	<i>Kontinuerling integration</i>	27
6.2.2	<i>Git hook</i>	27
6.3	Jämförelse med liknande parsers	27
7	Diskussion	29
	Källor	30
	Bilaga - Källkod	31

TABELLER

FIGURER

Figur 1. De fyra nivåerna i Chomskyhierarkin	10
Figur 2. EBNF grammatik för ett nummer.	12
Figur 3. BNF grammatik för en kalkylator.	12
Figur 4. Översikt av komponenterna i en kompilator.	13
Figur 5. Den syntastiska uppdelningen av en kod-sats	14
Figur 6. Abstrakt syntax träd som representation av en parsnings analys.	18
Figur 7. Översikt av de mest prestanda krävande funktionera	20
Figur 8. Förändring av funktionen isUnary från icke-optimerbar till optimerbar. . .	22
Figur 9. Procedur förändring från sträng sammanfogning till sträng utskärning. . .	24
Figur 10. Optimerad träd implementation av en vanlig switch-sats för strängar. . . .	24
Figur 11. Översikt av de mest prestanda påverkande kod commitsen	26
Figur 12. Jämförelse av parser implementationen och andra liknande lösningar . . .	28

1 INLEDNING

1.1 Syfte och målsättning

1.2 Utförande

2 SPRÅK TEORI

För att en dator skall ha möjlighet att förstå innebördet i ett uttryck krävs det att uttrycket är uppbyggt med en konsekvent utformning av såväl dess syntax och dess semantik. Detta krav existerar inte i de naturliga språk så som svenska utan är unikt för en specifik typ av språk, de formella språken. Formella språk är uppbyggda enligt matematiska regler som definierar sitt alfabet och hur det kan kombineras för att skapa uttryck. Teorin härstammar från språkvetenskap men har idag en stor betydelse inom datavetenskap eftersom det utnyttjas för att konstruera programmeringsspråk (Scott 2009 s. 41).

2.1 Backus-Naur Form

Inom datavetenskap är syntax den kombination av tecken som är giltig för att skapa ett uttryck. Uttryckets funktion kan vara av flera former, t.ex. kan det vara en del av ett flöde som skapar ett datorprogram, eller enbart format för att uttrycka konfigurationer. Processen av att läsa denna syntax och granska om den är giltig kallas för syntaktisk analys eller parsning.

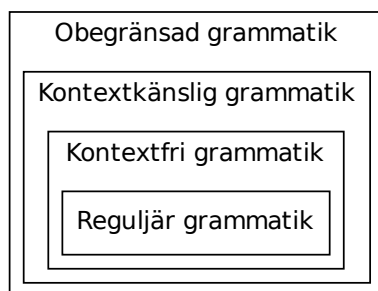
När man beskriver syntaxen av ett språk använder man sig av ett metaspråk för att definiera de syntaktiska regler man är tillåten att använda. Det finns ett flertal metaspråk men ett av de vanligaste inom programmeringsspråk är Backus-Naur Form (BNF) (Grune & Jacobs 2008 s. 27). Notationen är uppbyggd enligt produktionsregler som var för sig definierar en sammansättning av terminaler och icke-terminaler tillåtna för sin regel. Terminaler är ett begränsat alternativ teckensträngar, som kan liknas till ord i ett språk. Icke-terminaler är serier av terminaler som utgör en giltig produktionsregel, eller en språklig mening. Ytterligare existerar vissa symboler för att uttrycka vilken typ av sammansättningsfunktion som är tillåten.

BNF i sig existerar dessutom i flera varianter var vissa är mer strikta och ämnade till att läsas av maskiner, medan vissa försöker visualisera elementen för en mänsklig läsare. Extended BNF (EBNF) som är en utökad variant av den ursprungliga BNF notationen skriver

icke-terminaler inom vinkelparentes i formen $\langle regel \rangle$. En produktionsregels namn, som är en icke-terminal, skrivs längst till vänster följt av symbolerna $::=$ samt själva regeln. Terminalerna skrivs med fet stil och sammansättningarna skrivs med normal stil samt regelns specifika syntax. De vanliga funktionerna är alternering, som skrivs med ett lodrätt streck ($|$) mellan alternativen, repetition som skrivs med en klammerparentes ($\{\dots\}$) omkring uttrycket och slutligen valfrihet som skrivs med en hakparentes ($[\dots]$) runt uttrycket (Grune & Jacobs 2008 s. 28). Dessa är de viktigaste elementen i BNF men det existerar även övriga funktioner för bl.a. bekvämlighet och läsbarhet.

2.2 Chomskyhierarkin

När man skriver grammatiken till ett språk beaktar man alltid vilka typer av regler man vill tillåta i specifikationen. Utgående från valet man gör kommer grammatiken och därmed också språket tillhöra en av fyra språkliga delmängder som sträcker sig från simpel till komplicerad (Grune & Jacobs 2008 s. 19).



Figur 1. De fyra nivåerna i Chomskyhierarkin

Denna indelning kallas för Chomskyhierarkin och används bl.a. för att ta reda på vilken typ av automat som krävs för att läsa språket.

Delmängderna börjar från den simplaste typen, reguljär grammatik. Den reguljära grammatiken är sedan en delmängd av den kontextfria grammatiken som i sin tur är delmängd till den kontextkänsliga grammatiken. Slutligen tillhör alla de tidigare nämnda även den obegränsade grammatiken som kan beskriva alla grammatiker vilka accepteras av en Turingmaskin.

De två huvudsakliga grupperna i dagens programmeringsspråk är dock de två minsta, re-

guljära grammatiker samt kontextfria grammatiker (Scott 2009 s. 100). Dessa är möjliga att skriva både för hand och av maskiner och har därför blivit mycket vanliga i design av programmeringsspråk. Majoriteten av språk använder sig av en kontextfri syntax grammatik. Ett undantag är C++ vars uttryck inte kan definieras enbart utgående från syntaxen utan kräver också en semantisk analys av ett flertal uttryck. På grund av detta är C++ grammatiken kontextkänslig och därmed också svår att parsa. I flera fall har parser implementationer valt att ignorera mångtydigheterna på grund av dess komplexitet och dess lilla användning (Thomas 2005 s. 2).

2.2.1 Reguljär grammatik

Den minsta delmängden i Chomskyhierarkin är reguljär grammatik och kan uttryckas enbart mha. reglerna *sammanfogning*, *alternering* och *repetition*. I programmeringsspråk används ofta en reguljär grammatik för att identifiera tokens och går att läsa med en ändlig automat (Scott 2009 s. 100).

För att beskriva alla variationer av ett nummer i en kalkylator kan man använda sig av EBNF grammatiken i figur 2. Ett nummer definieras som alterneringen av ett heltal och ett reellt tal. Ett heltal måste bestå av minst en siffra medan ett reellt tal kan bestå av antingen ett heltal samt en exponent eller ett decimaltal och en valfri exponent. Detta innebär att uttrycken $0.14E-2$ och 3 är giltiga medan uttrycket $222e$ inte är giltigt eftersom en exponent måste avslutas med ett heltal. Dessutom måste man tänka på att ett giltigt decimaltal inte nödvändigtvis behöver börja med en siffra utan kan börja med en punkt, dock måste det antingen börja med en siffra eller avslutas med en siffra eftersom ett uttryck enbart innehållande en punkt inte kan räknas som giltigt. Alla dessa regler kan bli komplicerade att hålla reda på och därför underlättar det att arbeta med BNF notationer för att inte mista giltiga uttryck.

$$\begin{aligned}
\langle \text{nummer} \rangle &::= \langle \text{heltal} \rangle \mid \langle \text{reellt tal} \rangle \\
\langle \text{heltal} \rangle &::= \langle \text{siffra} \rangle \{ \langle \text{siffra} \rangle \} \\
\langle \text{reellt tal} \rangle &::= \langle \text{heltal} \rangle \langle \text{exponent} \rangle \mid \langle \text{decimaltal} \rangle [\langle \text{exponent} \rangle] \\
\langle \text{decimaltal} \rangle &::= \langle \text{siffra} \rangle (\cdot \langle \text{siffra} \rangle \mid \langle \text{siffra} \rangle \cdot) \langle \text{siffra} \rangle \\
\langle \text{exponent} \rangle &::= (\mathbf{e} \mid \mathbf{E}) [+ \mid -] \langle \text{heltal} \rangle \\
\langle \text{siffra} \rangle &::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}
\end{aligned}$$

Figur 2. EBNF grammatik för ett nummer.

2.2.2 Kontextfri grammatik

Tillåter man ytterligare *rekursion* som en giltig regel är grammatiken inte längre reguljär, utan klassas som en kontextfri grammatik och måste läsas av parser (Scott 2009 s. 100). Rekursion innebär att en produktionsregel kan innehålla sig själv som en icke-terminal i regel definitionen. Denna funktionalitet är användbar när ett uttryck skall vara flexibelt, exempelvis i en aritmetisk kalkylator var ett uttryck kan bestå av en siffra, en matematisk operation samt en oändlig uppsättning av dessa.

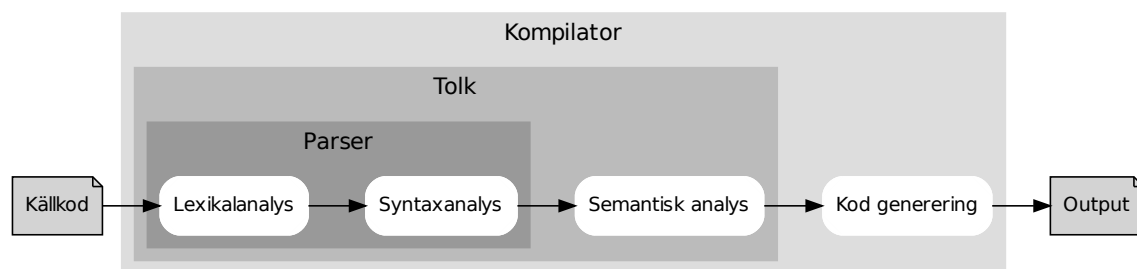
I figur 3 har vi ett exempel på en kalkylator som kan uttrycka alla dessa funktionaliteter genom att rekursivt hänvisa till sig själv och därmed tillåta uttryck så som $1 + (2 / 7) * -3$.

$$\begin{aligned}
\langle \text{uttryck} \rangle &::= \langle \text{nummer} \rangle \mid (- \mid +) \langle \text{uttryck} \rangle \mid (\langle \text{uttryck} \rangle) \\
&\quad \mid \langle \text{uttryck} \rangle \langle \text{operator} \rangle \langle \text{uttryck} \rangle \\
\langle \text{operator} \rangle &::= + \mid - \mid * \mid /
\end{aligned}$$

Figur 3. BNF grammatik för en kalkylator.

3 PARSNING

Implementeringen av ett programmeringsspråk finns i flera varianter och en vanlig sådan är kompilatorn. Denna implementation läser indata och producerar sedan ett körbart program utgående från de instruktioner den fått. Själva processen av kompilering består av flera faser och komponenter. Den första komponenten är en parser som läser indata och konstruerar en fördefinierad struktur av de regler som givits. Vid detta skede bryr sig programmet inte ännu om vad som skall göras utan den försöker enbart identifiera de olika reglerna och granska att dess syntax är korrekt. Parsern körs normalt som en komponent inne i en tolk vars funktion i sin tur är att förstå och tolka innebördet hos en regel. När parsern är klar med sin analys returnerar den strukturen till tolken som i sin tur returnerar sin omformning av strukturen tillbaka till huvudkomponenten, kompilatorn. Kompilatorn fungerar som en översättare som slutligen genererar den kod som datorns processor kan förstå. En översikt av dessa komponenter och dess funktioner kan hittas i figur 4 (Parr 2010 s. 16).



Figur 4. Översikt av komponenterna i en kompilator.

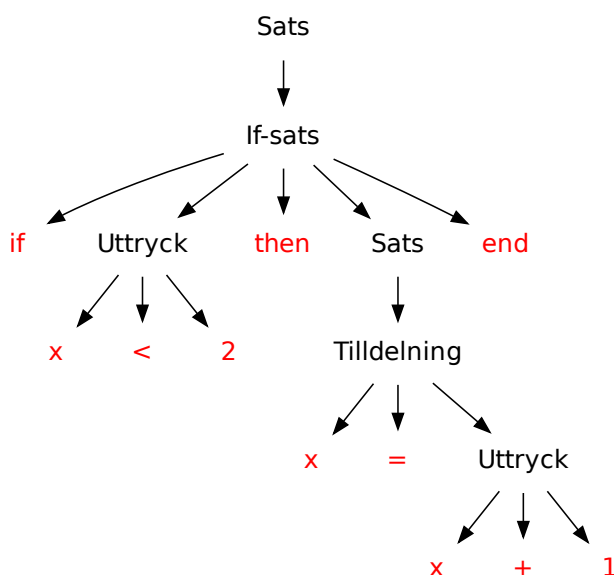
Parsnings processen kan delas upp i två skilda faser, först en s.k. lexikal analys som identifierar lexikala element, mer kända som tokens. Tokens är identifierbara teckensträngar med speciell betydelse. De kategoriseras enligt typer så som nyckelord, konstanter, parametrar osv. (Aho et al. 2006 s. 6).

Den andra fasen som sker efter att tokens är identifierade är den syntaktiska analysen var elementen sammansätts till helhets uttryck som utgör sin specifika funktion.

3.1 Lexikal analys

Eftersom elementen i en lexikal analys kan beskrivas med en reguljär grammatik använder man sig ofta av en ändlig automat för att läsa den. Denna typ av automat brukar man kalla för lexer. Automaten börjar i ett specifikt start läge var den väntar på ett tecken att läsa. När ett tecken läses går den genom en serie alternativa lösningar den har för att minska mängden slutgiltiga lösningar. När därpå följande tecken läses in fortsätter den att härleda sig vidare tills den når en slutgiltig lösning, eller alternativt inte känner igen elementet och kastar ett felmeddelande. När lösningen är hittad skickar lexern elementet tillbaka till parsern och återgår till sitt utgångsläge för att vänta på nästa tecken. På detta sätt kan lexern, som har en effektiv algoritm, kasta bort onödig information så som mellanslag och kommentarer för att sedan ge uttryckets riktiga element vidare till parsern som nu enkelt vet om en teckensträng är en nyckelordsterminal eller ett nummer (Scott 2009 s. 51).

3.2 Syntastisk analys



Figur 5. Den syntastiska uppdelningen av en kod-sats

3.3 Parser typer

Som det nämnts tidigare använder sig de flesta parsers sig av en lexer för att göra en lexikal analys av de tokens som ges som indata. När parsern får dessa tokens påbörjar den en syntaktisk analys som kan implementeras på diverse olika sätt. Huvudsakligen existerar de i två typer. Antingen härleder de regler från vänster och kallas då LL parsers eller så härleder de regler från höger och kallas LR parsers (Scott 2009 s. 67).

3.3.1 LL parser

LL parsers kan man skriva förhand eftersom de i allmänhet följer en människas tankegång. De börjar med att gissa sig till en rot-regel, likt en ändlig automat, och förväntar sig sedan att löv-reglerna skall passa in en efter en. Om det misslyckas kan den gå till nästa alternativa regel och fortsätter fram tills den bevisat sin gissning. Denna typ av strategi kallas även för en uppifrån-och-ner strategi.

3.3.2 LR parser

LR parsers genereras huvudsakligen av maskiner eftersom de är svårare att visualisera. Dessa parsers börjar med att läsa löv-reglerna, alltså de minsta identifierbara reglerna och ansluter dem sedan till varandra fram till det att den nått en slutgiltig rot-regel. Detta kallas även för en nerifrån-och-upp strategi.

3.3.3 Recursive descent parser

...

3.4 Tekniker

3.4.1 Left factoring

Hur gör man left factoring

3.4.2 Prediction

Hur fungerar en predictive parser.

4 IMPLEMENTATION

4.1 Lua

Vad är Lua och hur utmärks det.

4.1.1 Funktionalitet

Vilka funktioner har lua.

4.1.2 Datastrukturer

Beskriv hur allt är tables och hur det kan användas till OOP.

4.1.3 Grammatik

Beskriv luas grammatik och dess möjliga problem. Hänvisa till bilaga.

4.2 Design

LL(1)

4.3 Modifiering av grammatik

Hur modifierar vi grammatiken för en LL(1) parser

4.3.1 Lexer

...

4.3.2 Parser

Beskriv hur parsern skiljer sig från luas egna.

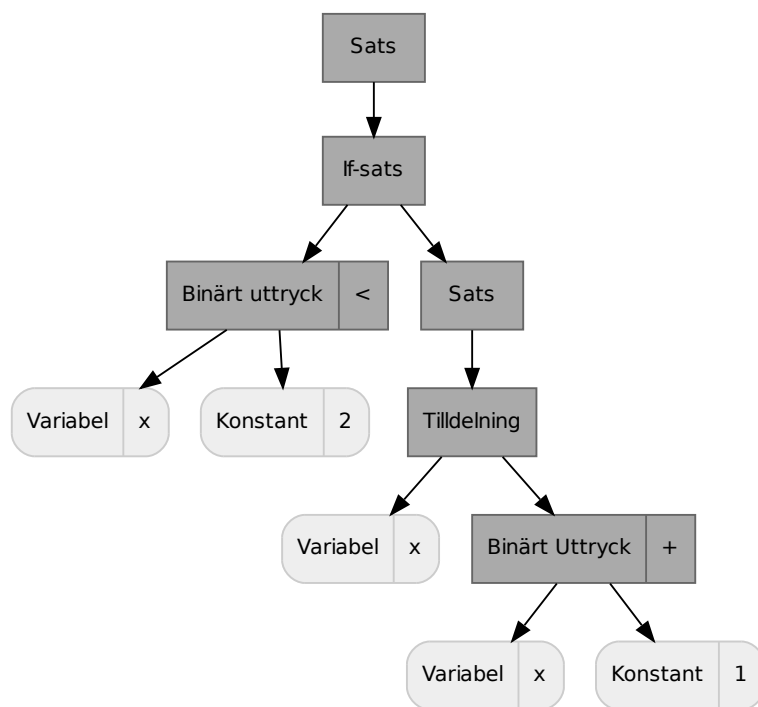
4.3.3 Uttrycks parser

Likna parsern till Luas egna.

4.3.4 Resultat träd

Beskriv Mozilla Parser API, och funktionaliteten med separat AST funktioner.

Vad är AST?



Figur 6. Abstrakt syntax träd som representation av en parsnings analys.

4.4 JavaScript kodmönster

4.4.1 Imperativt paradigm

OOP vs Imperativt.

4.4.2 Modul laddare

Require.JS, AMD, Node.JS, Browsers, vad det kan användas till.

5 PRESTANDA OPTIMERING

Beskriv varför JavaScript behöver micro optimering.

5.1 Insamling av data

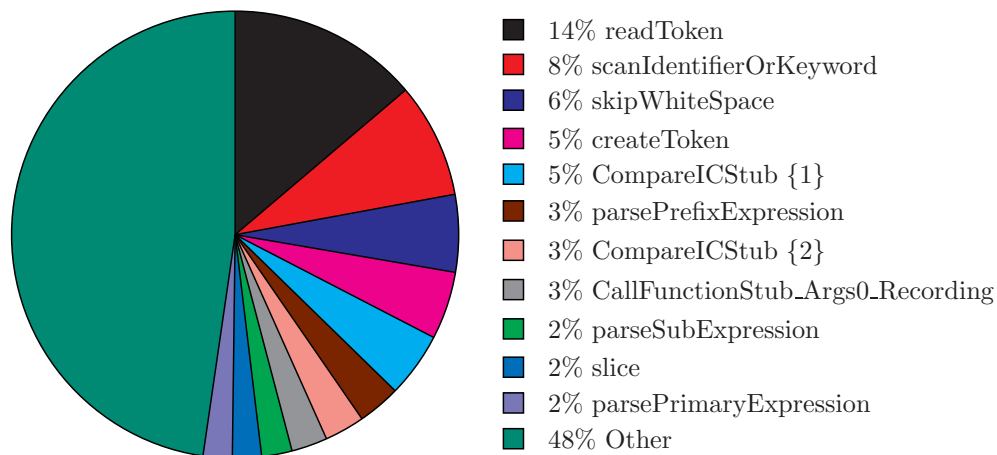
Beskriv processen för test samt sample standard deviation.

5.2 Analysering av data för V8

Beskriv varför v8.

5.2.1 Sampling

Beskriv samplings verktyget och hur datat läses.



Figur 7. Översikt av de mest prestanda krävande funktionerna

5.3 Optimerings tekniker för V8

Javascript motorn V8 innehåller en optimerings kompilator kallad Crankshaft, vars syfte är att identifiera kod som körs mycket för att sedan spendera sin tid på att optimera dom istället för de mindre väsentliga funktionerna. Detta fungerar genom att baskompi-

lern börjar med att kompilera koden varefter en runtime profiler övervakar körningen av programmet och identifierar den kod som körs aktivt. När de väsentliga programdelarna är identifierade optimeras och omkompileras de ivrigt av optimerings kompilatorn. Innebördet med att de omkompileras ivrigt är att det ytterligare finns en komponent som kan återgå till baskompilatorns version om det visar sig att koden slutligen inte var optimerbar (Millikin & Schneider 2010). Denna åtgärd kallas för deoptimization och inträffar aktivt i körningen av javascript eftersom språket är dynamiskt och en deoptimization sker varje gång en datatyp förändras. Ytterligare existerar det vissa specifika strukturer som kompilatorn inte kan optimera, exempelvis try/catch-satser och vissa typer av switch-satser. Även här görs en deoptimization men den kallas nu för en bailout. (@TODO terminologi?)

V8 och därmed även Node.JS har diverse kommandorads flaggor som hjälper identifiera funktioner var detta sker samt ge information om varför det sker.

5.3.1 Undvikandet av bailouts

Trots att bailouts möjliggör en allmän prestanda förbättring hos V8 är det vettigt att undvika de bailouts som görs eftersom de vid varje exekvering kommer återupprepa sig och processen med att återgå till den ursprungliga kompilerings versionen kostar en del prestanda.

Kommandorads flaggan *-trace_bailout* matar ut en lista på funktioner samt orsaken till varför en bailout gjorts. Genom att exekvera en parsning med flaggan aktiverad går det att identifiera två funktioner som gör en bailout, bägge på grund av ooptimerbara switch statements var en icke-literal label användts.

Funktionerna som orsakat dessa bailouts är *isUnary()* samt *parsePrimaryExpression()* som bägge består av en switch-sats som utför olika operationer beroende på om en given token är lika med ett visst variabelvärde. Detta är ett enkelt problem att lösa eftersom bailouten inte sker för if-sats motsvarigheten och därför kan koden optimeras enligt figur 8.

<pre>function isUnary(token) { switch (token.type) { case Tokens.Punctuator: return ~'#-'.indexOf(token.value); case Tokens.Keyword: return token.value === 'not'; } return false; }</pre> <p>a) Funktion före</p>	<pre>function isUnary(token) { if (token.type === Tokens.Punctuator) return ~'#-'.indexOf(token.value); if (token.type === Tokens.Keyword) return token.value === 'not'; }</pre> <p>b) Funktion efter</p>
--	---

Figur 8. Förändring av funktionen `isUnary` från icke-optimerbar till optimerbar.

5.3.2 Inlining

@TODO expandera kanske? Hur översätta literaler?

Crankshaft använder sig även av optimeringstekniken funktion inlining. Detta innebär att kompilatorn ersätter ett funktionsanrop med själva funktionsinnehållet och sparar därmed kostnaden av själva funktions anropet och dess returnering. Dessutom möjliggör det ytterligare optimeringstekniker eftersom koden nu är mer kompakt och introducerar satser som möjligen kan kombineras med omliggande kod.

V8 har dock vissa begränsningar för vilka funktioner som går att inlinea. Bland annat får källkoden inte överstiga 600 bytes (hur skall jag referera till v8 koden?) och funktioner kan inte heller innehålla array literaler. I V8 motorn som kommer med Node.JS v0.8 går det inte heller att inlinea funktioner som innehåller objekt literaler.

Lexer implementationen som gjorts använde till att börja med två separata funktioner som skapade och avslutade token noder. Eftersom noderna var uppgjorda av objekt kunde dessa inte inlineas och därmed introducerades en funktionskallelse extra för varje token som hittades under en parsning. På grund av jag inte ansåg detta påverka läsligheten eller komplexiteten av koden ansåg jag det vara värt att expandera skapandet och avslutandet av noderna i varje enskild token funktion.

5.3.3 Teckenkoder istället för reguljära uttryck

Med användning av samplings verktyget i V8 är det möjligt att få en överblick av vilka funktioner och operationer som är mest aktiva under en exekvering. Genom att analysera detta resultat uppgavs det att 39.2% av en hel exekvering går åt till reguljära uttryck som används för att identifiera whitespace och giltiga identifierare.

Eftersom dessa operationer ansvarade för en extremt stor del av parsningsprocessen omvandlades de till binära operationer som jämför teckenkoder istället. Orsaken till att teckenkoder används istället för string literals är att en kompilator måste omvandla tecken till siffror för att göra jämförelser, denna operation visar sig ha en betydande prestanda skillnad i äldre V8 versioner samt övriga webbläsare (hur ska dessa refereras? <http://jsperf.com/charcodeat-vs-string-comparison/2>).

5.3.4 Optimering av sträng funktioner

Efter att de tidigare optimeringarna genomförts sticker nu 2 huvudsakliga funktioner ut i samplings analysen. StringAddStub som är en maskinkods metod för att sammanfoga strängar och CompareICStub som i en metod för att jämföra strängar tar plats 3 och 4 med 8.3% respektive 7.9% av parsningsprocessen.

StringAddStub orsakas av att lexern sammanfoga token värden för kommentarer, nummer och identifierare genom att iterera över varje tecken så länge som en avgränsare inte hittas. Denna procedur går att förenkla genom att istället räkna var värdet börjar och sedan iterera fram till avgränsaren och vid det skedet skära ut värdet från dess start position till positionen var avgränsaren enligt exempel figur 9.

Vid vidare analys av sampling informationen gick det att utläsa att den huvudsakliga orsaken för den långsamma CompareICStub operation var en funktion som inlineats i uttrycks parsern. Funktionen ansvarar för att ange prioriteten av en given binära operator utgående från en switch-sats med sträng jämförelser. Eftersom funktionen körs för varje uttryck som hittas valde jag att speciellt uppoffra läsligheten med att istället jämföra teckenko-

```

var value = '';
while (isIdentPart(input.charCodeAt(index)))
    value += input[index++];
a) Procedur före

var start = index;
while (isIdentPart(input.charCodeAt(index)))
    index++;
var value = input.slice(start, index);
b) Procedur efter

```

Figur 9. Procedur förändring från sträng sammanfogning till sträng utskärning.

der i formen av ett träd enligt figur 10. Efter att optimeringen gjorts validerade jag ännu funktionen var tillräckligt liten för att bli inlined, vilket den visade sig vara.

```

function binaryPrecedence(operator) {
    var char = operator.charCodeAt(0)
        , length = operator.length;

    if (1 === length) {
        switch (char) {
            case 94: return 10; // ^
            case 42: case 47: case 37: return 7; // * / %
            case 43: case 45: return 6; // + -
            case 60: case 62: return 3; // < >
        }
    } else if (2 === length) {
        switch (char) {
            case 46: return 5; // ..
            case 60: case 62: case 61: case 126: return 3; // <= >= == ~=
            case 111: return 1; // or
        }
    } else if (97 === char && 'and' === operator) return 2;
    return 0;
}

```

Figur 10. Optimerad träd implementation av en vanlig switch-sats för strängar.

5.3.5 Garbage Collection

V8 består av två typer av garbage collectors som den väljer mellan att använda under en exekvering. Den första är en scavanger som arbetar snabbt men enbart på indirekt skapa-

de objekt, dvs. objekt utan nyckelordet `new`. Den andra typen av collector använder sig av en mark-sweep algoritm som även granskar indirekt skapade objekt men är betydligt långsammare (Ager 2010).

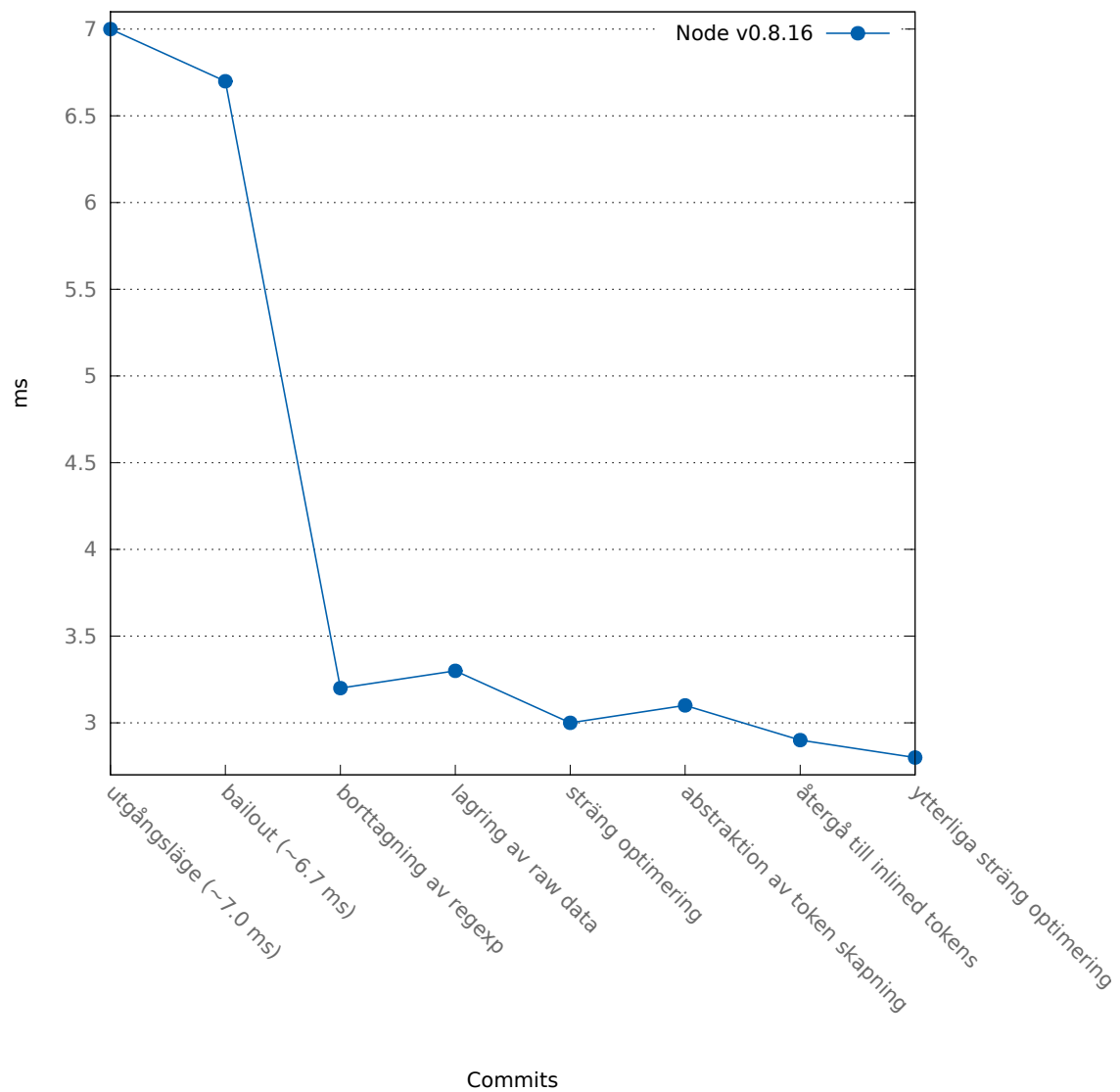
Eftersom parsern implementerats med hjälp av scopes och funktioner istället för med ett objektorienterat paradigm är all allokering indirekt och därmed finns det ingen oro om att minne skall läcka så länge som mer avancerade closure användningar inte används. När en funktion körts färdigt blir alla variabler som definierats inne i funktionen utom räckhåll för programmet och därmed kommer de tas bort vid nästa scavanger runda. På grund av att minnet knyts med hjälp av scopes är det därför viktigt att undvika globala variabler samt se till att hela programmet är knytet till ett eget scope som utomstående objekt inte har tillgång till.

Med hjälp av kommandorads flagga `-trace_gc` kan vi se vilken typ av garbage collection som görs samt hur länge den tagit.

Utgående från resultatet ser vi att det sker 215 scavange collections som alla tar under 1ms att utföra, samt en mark-sweep som utförs tidigt i exekveringen och kräver 7ms tid. Utgående från detta kan det konstateras att implementationen inte har läcker minne och att scavange collectorn kan upprätthålla programmet utan mer än en inledande mark-sweep.

5.4 Resultat

Beskriv vart vi kommit och varför vi inte kan optimera mer.



Figur 11. Översikt av de mest prestanda påverkande kod commitsen

6 KVALITETS ANALYS

6.1 Mätning

6.1.1 Test

6.1.2 Kodtäckning

6.1.3 Kod komplexitet

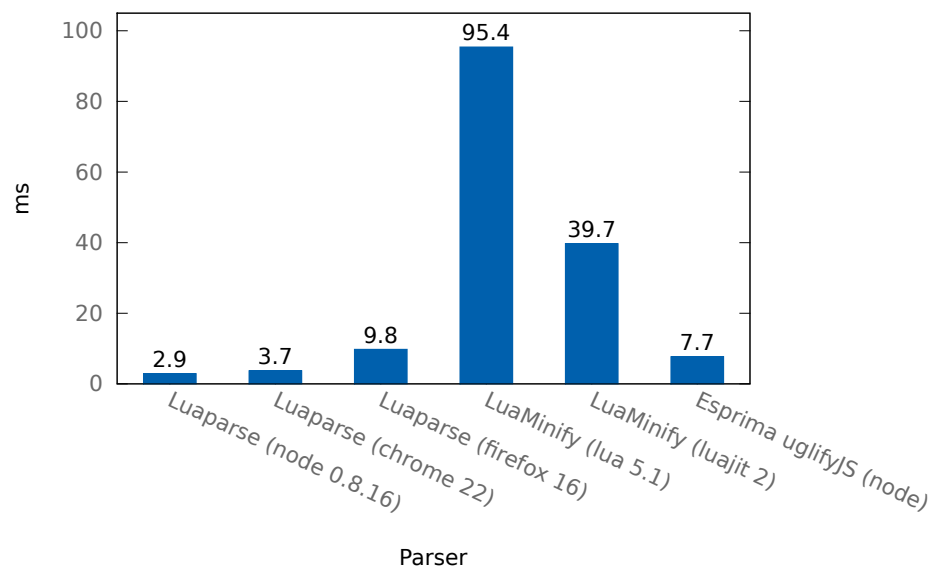
6.1.4 Benchmark

6.2 Automation

6.2.1 Kontinuerling integration

6.2.2 Git hook

6.3 Jämförelse med liknande parsers



Figur 12. Jämförelse av parser implementationen och andra liknande lösningar

7 DISKUSSION

...

KÄLLOR

- Ager, Mads. 2010, *The V8 JavaScript Engine: Design, Implementation, Testing and Benchmarking*, doktorsavhandling, Odense: University of Southern Denmark.
Tillgänglig: http://websrv0a.sdu.dk/ups/SCM/slides/lecture_03_mads_ager.pdf,
Hämtad: 8.1.2013.
- Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi & Ullman, Jeffrey D. 2006, *Compilers: Principles, Techniques & Tools*, 2 uppl., Prentice Hall, 1000 s.
- Grune, Dick & Jacobs, Cerial J.H. 2008, *Parsing Techniques: A Practical Guide*, 2 uppl., New York: Springer, 662 s.
- Millikin, Kevin & Schneider, Florian. 2010, A new Crankshaft for V8, *The Chromium Blog*. Tillgänglig: <http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html>,
Hämtad: 8.1.2013.
- Parr, Terence. 2010, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, Pragmatic Bookshelf, 374 s.
- Scott, Michael L. 2009, *Programing Language Pragmatics*, 3 uppl., Burlington: Morgan Kauffmann Publishers, 910 s.
- Thomas, Rudolf. 2005, *A Parser of the C++ Programming Language*, Examensarbete, Prague: Charles University, Faculty of Mathematics and Physics.

BILAGA - LUA GRAMMATIK

$\langle chunk \rangle$	$::= \langle block \rangle$
$\langle block \rangle$	$::= \{ \langle stat \rangle \} [\langle retstat \rangle]$
$\langle stat \rangle$	$::= ;$ $\langle varlist \rangle = \langle explist \rangle$ $\langle functioncall \rangle$ $\langle label \rangle$ break goto Name do $\langle block \rangle$ end while $\langle exp \rangle$ do $\langle block \rangle$ end repeat $\langle block \rangle$ until $\langle exp \rangle$ if $\langle exp \rangle$ then $\langle block \rangle$ { elseif $\langle exp \rangle$ then $\langle block \rangle$ } [else $\langle block \rangle$] end for Name = $\langle exp \rangle$, $\langle exp \rangle$ [, $\langle exp \rangle$] do $\langle block \rangle$ end for $\langle namelist \rangle$ in $\langle explist \rangle$ do $\langle block \rangle$ end function funcname $\langle funcbody \rangle$ local $\langle function \rangle$ Name $\langle funcbody \rangle$ local $\langle namelist \rangle$ [= $\langle explist \rangle$]
$\langle retstat \rangle$	$::= \textbf{return} [\langle explist \rangle] [;]$
$\langle label \rangle$	$::= :: \text{Name} ::$
$\langle funcname \rangle$	$::= \text{Name} \{ . \text{Name} \} [: \text{Name}]$
$\langle varlist \rangle$	$::= \langle var \rangle \{ , \langle var \rangle \}$
$\langle var \rangle$	$::= \text{Name} \mid \langle prefixexp \rangle [\langle exp \rangle] \mid \langle prefixexp \rangle . \text{Name}$
$\langle namelist \rangle$	$::= \text{Name} \{ , \text{Name} \}$
$\langle explist \rangle$	$::= \langle exp \rangle \{ , \langle exp \rangle \}$
$\langle exp \rangle$	$::= \textbf{nil} \mid \textbf{false} \mid \textbf{true} \mid \text{Number} \mid \text{String} \mid \dots \mid \langle functiondef \rangle \mid \langle prefixexp \rangle$ $\langle tableconstructor \rangle \mid \langle exp \rangle \langle binop \rangle \langle exp \rangle \mid \langle unop \rangle \langle exp \rangle$
$\langle prefixexp \rangle$	$::= \langle var \rangle \mid \langle functioncall \rangle \mid (\langle exp \rangle)$
$\langle functioncall \rangle$	$::= \langle prefixexp \rangle \langle args \rangle \mid \langle prefixexp \rangle : \text{Name} \langle args \rangle$
$\langle args \rangle$	$::= ([\langle explist \rangle]) \mid \langle tableconstructor \rangle \mid \text{String}$
$\langle functiondef \rangle$	$::= \textbf{function} \langle funcbody \rangle$
$\langle funcbody \rangle$	$::= ([\langle parlist \rangle]) \langle block \rangle \textbf{end}$
$\langle parlist \rangle$	$::= \langle namelist \rangle [, \dots] \mid \dots$
$\langle tableconstructor \rangle$	$::= \{ [\langle fieldlist \rangle] \}$
$\langle fieldlist \rangle$	$::= \langle field \rangle \{ \langle fieldsep \rangle \langle field \rangle \} [\langle fieldsep \rangle]$
$\langle field \rangle$	$::= [\langle exp \rangle] = \langle exp \rangle \mid \text{Name} = \langle exp \rangle \mid \langle exp \rangle$
$\langle fieldsep \rangle$	$::= , \mid ;$
$\langle binop \rangle$	$::= + \mid - \mid * \mid / \mid ^ \mid \% \mid .. \mid < \mid <= \mid > \mid >= \mid == \mid ~= \mid \textbf{and} \mid \textbf{or}$
$\langle unop \rangle$	$::= - \mid \textbf{not} \mid \#$