Implementering av en Lua-parser

Oskar Schöldström

Examensarbete
Informationsteknik
2013

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informations- och medieteknik
Identifikationsnummer:	4052
Författare:	Oskar Schöldström
Arbetets namn:	Implementering av en Lua-parser
Handledare (Arcada):	Göran Pulkkis
Uppdragsgivare:	

Sammandrag:
Avsikten med detta examensarbete är att redogöra för hur parsningen av ett programmeringsspråk fungerar samt att undersöka varför vissa kompilatorer implementerar en handskriven parser medan andra använder en maskingenererad parser. Examensarbetet består av en teoretisk grund samt en praktisk implementation av en handskriven Lua-parser. Den teoretiska delen beskriver hur programmeringsspråk är uppbyggda samt ger en överblick av hur en parser implementeras. Teknikerna som tas upp fokuserar huvudsakligen på handskrivna parsers. Den praktiska implementationen är programmerad i JavaScript för att kunna användas som ett analyseringsverktyg i en nätbaserad programkodsredigerare. Problemen som påträffas under implementationen behandlas med hjälp av tekniker presenterade i teorin. Slutligen genomgår parserimplementationen en prestandaoptimering för JavaScript-motorn V8. Resultatet jämförs med en maskingenererad parser samt med en handskriven parser implementerad i Lua. Slutsatsen är att handskrivna parsers kan uppnå högre prestanda samt ökad flexibilitet men kräver avsevärt mera tid att implementera än en maskingenererad parser.

Nyckelord:	Parsning, Rekursivt nedstigande parser, Syntaktisk analys, Lexikal analys, Metaspråk, JavaScript, Lua
Sidantal:	73
Språk:	Svenska
Datum för godkännande:	25.04.2013

DEGREE THESIS	
Arcada	
Degree Programme:	Information and Media Technology
Identification number:	4052
Author:	Oskar Schöldström
Title:	Implementing a Lua parser
Supervisor (Arcada):	Göran Pulkkis
Commissioned by:	

Abstract:
The purpose of this thesis is to describe the process of parsing a programming language and investigate why some compilers implement their own handwritten parser while others use a machine generated parser. The thesis consists of a theoretical foundation and a practical implementation of a handwritten Lua parser. The theoretical part gives an overview of how programming languages are built and how a parser is implemented. The techniques presented focus primarily on handwritten parsers. The practical parser implementation is written in JavaScript so that it can be used as an analysis tool within online code editors. The problems encountered during the implementation process are solved with the help of techniques presented in the theoretical part. At the end the parser undergoes a performance optimization for the V8 JavaScript engine. The achieved results are compared to a machine generated parser as well as to a handwritten parser implemented in Lua. The conclusion is that a handwritten parser can achieve higher performance and an increased flexibility but is significantly more time consuming to implement than a machine generated parser.

Keywords:	Parsing, Recursive descent parser, Syntactic analysis, Lexical analysis, Metalanguage, JavaScript, Lua
Number of pages:	73
Language:	Swedish
Date of acceptance:	25.04.2013

INNEHÅLL

TABELLER

FIGURER

FÖRKORTNINGAR

API	Application Programming Interface
AST	Abstract Syntax Tree
ASCII	The American Standard Code for Information Interchange
BNF	Backus-Naur Form
EBNF	Extended Backus-Naur Form
EOF	End-of-file
GCC	The GNU Compiler Collection
GHz	Gigahertz
GNU	GNU's Not Unix
JIT	Just-In-Time
LL	Left to right, Leftmost derivation
LR	Left to right, Rightmost derivation
LALR	Look-Ahead Left to right, Rightmost derivation

1 INLEDNING

Det existerar ett flertal olika parser-arkitekturer i dagens kompilatorer. Vissa använder sig av maskingenererad programkod medan andra implementerar komponenten manuellt. Beroende på komplexiteten av det kompilerade språkets syntax påverkas även komplexiteten av parser-implementationen av att vissa implementationer väljer att inte vara fullständiga.

Denna examensarbetsrapport utgår från att läsaren har kunskap inom programmering i allmänhet samt en förståelse av JavaScript-syntax.

1.1 Målsättning

Målet med detta examensarbete är att redogöra för hur ett programmeringsspråk är uppbyggt och hur parsningen av dess syntax implementeras. Varför är vissa språk svårare att parsa än andra? Vad är orsaken till att vissa kompilatorer implementerar parserkomponenten manuellt medan andra använder en maskingenererad komponent? Hur utförs implementationen av en parser i praktiken och hur kompliceratär det? Dessa frågor ämnar examensarbetet besvara.

1.2 Utförande

Teoridelen i detta arbete redogör för hur programmeringsspråk är uppbyggt samt hur de huvudsakliga parsning-arkitekturerna fungerar och används.

Den praktiska delen använder denna teori för att implementera en handskriven Lua-parser i JavaScript. I kapitel 6 beskrivs en prestandaanalys och parserns väsentliga flaskhalsar avlägsnas. Orsaken till att JavaScript valts som implementationsspråk är att i framtiden kunna använda parsern som ett analyseringsverktyg i en nätbaserad programkodsredigerare.

För att upprätthålla en hög kvalitet på implementationen har ett antal kvalitetssäkringar uppgjorts. Över 500 funktionstest har skapats med hjälp av kodgenerering och manuell verifiering. Testen baserar sig på Yueliang-projektets testsvit. Med dessa test har ett flertal fel korrigerats och implementationen har i nuläget en programkodstäckning på 100%.

Funktionstest, mätning av programkodstäckning, test för funktionskomplexitet samt en statisk programkodsanalys körs före varje uppdatering för att verifiera dess kvalitet. Ytterligare används Travis kontinuerlig integrering för att verifiera att utomstående uppdateringar håller standarden.

1.3 Avgränsning

Teorin om formella språk innehåller enbart det som krävs för att förstå senare kapitel. Ytterligare presenteras enbart tekniker som är anses vara aktuella för parser-implementationen eller Lua-språket. Till detta examensarbete hör inte senare kompilatorskeden såsom semantisk analys eller programkodsgenerering.

2 SPRÅKTEORI

För att en dator skall ha möjlighet att förstå innebörden 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 naturliga språk som svenska utan existerar för en specifik typ av språk, de formella språken. Formella språk är uppbyggda enligt matematiska regler som definierar språkets alfabet och hur alfabetsymboler kan kombineras för att skapa uttryck. Teorin härstammar från språkvetenskap men har idag en stor betydelse inom datavetenskap eftersom den utnyttjas för att konstruera programmeringsspråk (Scott 2009 s. 41).

2.1 Backus-Naur-notation

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

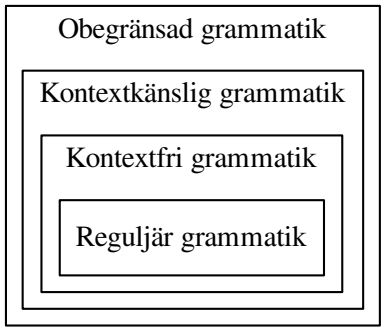
När man beskriver syntaxen av ett språk använder man sig av ett metaspråk för att definiera de syntaktiska regler man är tillåten att använda. Det finns ett flertal metaspråk men ett av de vanligaste inom programmeringsspråk är Backus-Naur-notation (Backus-Naur Form, BNF) (Grune & Jacobs 2008 s. 27). Notationen är uppbyggd enligt produktionsregler som var för sig definierar en tillåten sammansättning av teckensträngar som kallas terminaler eller icke-terminaler. Terminaler är teckensträngar som inte refererar vidare till andra teckensträngar medan icke-terminaler är sekvenser av terminaler som bildar giltiga produktionsregler eller språkliga meningar. Ytterligare existerar vissa symboler för att uttrycka vilken typ av sammansättningsfunktion är tillåten.

BNF i sig existerar dessutom i flera varianter där vissa varianter är strikta och ämnade att läsas av maskiner, medan andra varianter 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 produktionregels 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 vågparentes ($\{\ldots\}$) omkring uttrycket och slutligen valfrihet som skrivs med en hakparentes ($[\ldots]$) 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 enkel till komplicerad (Grune & Jacobs 2008 s. 19) enligt figur 1.



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. Typen av automat påverkar komplexiteten av implementationen samt tidsmängden som krävs för att läsa språket.

Delmängderna börjar från den enklaste 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 innersta, reguljära 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. 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 deras komplexitet och deras obetydliga användning (Thomas 2005 s. 2).

2.2.1 Reguljär grammatik

Den innersta delmängden i Chomskyhierarkin är reguljär grammatik och kan uttryckas enbart m.h.a. reglerna sammanfogning, alternering och repetition. I programmeringsspråk används ofta en reguljär grammatik för att identifiera s.k. lexikala element och går att läsa med en ändlig automat (Scott 2009 s. 100).

För att beskriva alla variationer av ett naturligt tal i en kalkylator kan man använda sig av EBNF grammatiken i figur 2. Ett tal 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 i en riktig kalkylator 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.

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

Figur 2. EBNF grammatik för ett tal.

2.2.2 Kontextfri grammatik

Tillåter man ytterligare rekursion i en giltig regel är grammatiken inte längre reguljär, utan klassas som en kontextfri grammatik och måste läsas av en s.k. *"push-down"* automat ofta kallad parser. Skillnaden från en ändlig automat är att *"push-down"* automaten har en stack av tillstånd (Scott 2009 s. 100). Rekursion innebär att en produktionsregel kan innehålla sig själv som en icke-terminal i regeldefinitionen. Denna funktionalitet är användbar när ett uttryck skall vara flexibelt, exempelvis i en kalkylator var ett uttryck kan bestå av ett tal, en matematisk operation samt en oändlig uppsättning av dessa.

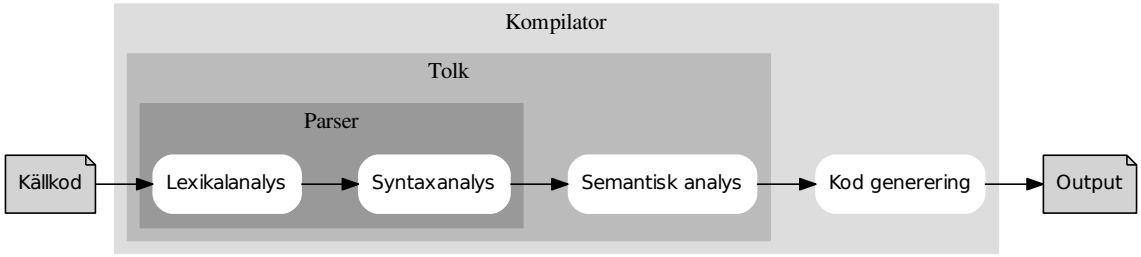
Figur 3 visar ett exempel på en kalkylator som kan uttrycka alla dessa funktionaliteter genom att rekursivt hänvisa till sig själv och därmed tillåta uttryck så som *1 + (2 / 7) * -3*.

$$
\begin{array}{lll}
\langle uttryck \rangle & ::= & \langle tal \rangle \\
 & \mid & (\ \langle uttryck \rangle\ ) \\
 & \mid & \langle uttryck \rangle\ \langle operator \rangle\ \langle uttryck \rangle \\
\langle operator \rangle & ::= & + \mid \text{-} \mid * \mid /
\end{array}
$$

Figur 3. EBNF grammatik för en kalkylator.

3 PARSNING

Implementeringen av ett programmeringsspråk finns i flera varianter och en vanlig sådan är kompilatorn. Denna implementation läser input 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 input och konstruerar en maskinläslig struktur utgående från den grammatik 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 modifierade version av strukturen tillbaka till huvudkomponenten, kompilatorn. Kompilatorn fungerar som en översättare som slutligen genererar den maskinkod som datorns processor förstår. En översikt av dessa komponenter och dess funktioner visas i figur 4 (Parr 2010 s. 16).



Figur 4. Översikt av komponenterna i en kompilator.

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

Den andra fasen som sker efter identifieringen av tokens är den syntaktiska analysen där elementen sammansätts till helhetsuttryck granskat enligt grammatikens produktionsregler.

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 startläge var den väntar på ett tecken att läsa. När ett tecken läses går den genom en serie alterneringar 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 skapar ett felmeddelande. När lösningen är hittad skickar lexern det identifierade 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, avlägsna onödig information såsom mellanslag och kommentarer för att sedan ge uttryckets egentliga element vidare till parsern som nu enkelt vet om en teckensträng är en nyckelordsterminal eller ett tal (Scott 2009 s. 51). Ett lexikaliseringsexempel visas i figur 5.

```
if 2 + 1 > 3 == false then
  -- en kommentar
  print "foo";
end
```

Figur 5. Ett lexikaliseringsexempel på en if-sats skriven i Lua. Blanksteg samt kommentarer ignoreras och tokentyperna är nyckelord, tal, specialsymbol och teckensträng.

3.2 Syntaktisk analys

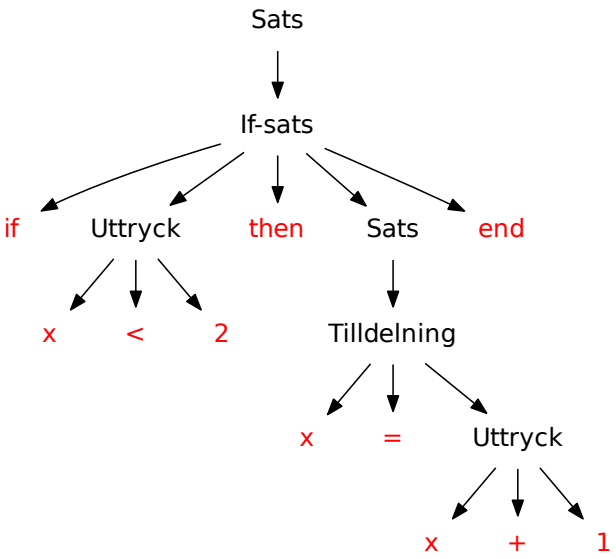
Processen att parsa input och validera dess syntax enligt en kontextfri grammatik kallas syntaktisk analys eller enbart parsning. Detta görs vanligtvis i kombination med en lexikal analys för att förenkla implementation men kan också genomföras direkt på input (Aho et al. 2006 s. 8).

Analysen kombinerar de tokens som identifierats en efter en och försöker hitta en giltig produktionsregel för kombinationen. Om en produktionsregel identifierats förväntas alla tokens överensstämma med regeldefinitionens terminaler och icke-terminaler. Visar det sig att en token inte överensstämmer skapas ett felmeddelande.

Vid implementationen av en kompilator avbryts parsningen när ett fel påträffats eftersom syntaxen inte är giltig. Vissa andra implementationer såsom *"syntax highlighters"* i textredigerare försöker hoppa över produktionsregeln och fortsätta med nästa eftersom detta ger en bättre användarupplevelse.

3.3 Syntaxrepresentation

Allteftersom produktionsregler parsats i.o.m. en syntaktisk analys skapas en maskinförståelig representation av dess innehåll. Lättast är det att tänka sig denna representation som en trädstruktur trots att den inte nödvändigtvis behöver vara det.



Figur 6. En trädrepresentation av en if-sats skriven i Lua.

Varje nod i trädet representeras av en produktionsregel. Löv-noderna är terminaler (figur 6). Terminaler såsom *if*, och *else* utelämnas ofta i en praktisk implementation eftersom de kan knytas som attribut till en nod (Scott 2009 s. 49).

Representationen skapas i den syntaktiska analysatorn när en produktionsregel identifierats och kan sedan användas i senare skeden såsom i en semantisk analysator eller i en komponent för programkodsoptimering.

Trädstrukturerna varierar beroende på syftet av parsern. Kompilatorer vill att de skall vara så nära maskinkod som möjligt medan andra verktyg såsom statiska programkodsanaly-

satorer vill att de ska vara på en högre nivå (Parr 2010 s. 6).

En vanlig representationsform på en högre nivå är ett abstrakt syntaxträd (AST). Detta är verkligen uppbyggt såsom ett träd enligt vad beskrivits hittills. Karaktärsdrag av ett AST är att det är kompakt, enkelt att förflytta sig genom och betydelsefullt. Exempelvis kan varje nod existera som ett objekt i en klass namngiven efter produktionsregeln (Parr 2010 s 77).

3.4 Parsertyper

Beroende på grammatiken av ett språk krävs det olika algoritmer för att kunna parsa den. Den s.k. Early-algoritmen kan parsa alla typer av kontextfria grammatiker i $O(n^3)$ tid, där n är inputlängd (Scott 2009 s. 67). De flesta parsers behöver dock inte en så generell grammatik utan kan parsas i $O(n)$ tid med hjälp av *"uppifrån-och-ner"*-algoritmer eller *"nerifrån-och-upp"*-algoritmer. Det existerar ytterligare algoritmer för olika delmängder av kontextfria grammatiker men de vanligaste bygger på någon av dessa (Aho et al. 2006 s. 61).

När man beskriver en parser nämner man ofta hur långt fram den kan se innan den gör ett beslut av vilken produktionsregel den skall följa, hur många s.k. *"lookahead"*-tokens den har. Detta antal skriver man inom en parentes efter parsertypens namn. Exempelvis skulle den s.k. LL-parsern med 2 *"lookahead"*-tokens skrivas LL(2) (Scott 2009 s. 69).

3.4.1 LL-parser

LL-parsers härleder regler från vänster och använder sig av *"uppifrån-och-ner"*-algoritmen. Parsern klarar av en mindre delmängd av kontextfria grammatiker och man använder benämningen LL-grammatik för att uttrycka den delmängd som en LL-parser kan parsa.

En LL-parser går att skriva för hand eftersom den i allmänhet följer en logisk tankegång. Den börjar från en rot-regel och arbetar sig sedan neråt med hjälp av att jämföra terminaler

från vänster, likt en ändlig automat. När alla alternativa lösningar uteslutits förväntar sig parsern att löv-reglerna skall passa in en efter en.

3.4.2 LR-parser

LR-parsers härleder regler från höger och använder sig av *"nerifrån-och-upp"*-algoritmen. Denna typ av parser klarar av att analysera en större delmängd grammatiker än t.ex. LL-parsers och är därmed mer vanlig i programvara som genererar parsers (Aho et al. 2006 s. 61).

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 (Scott 2009 s. 67).

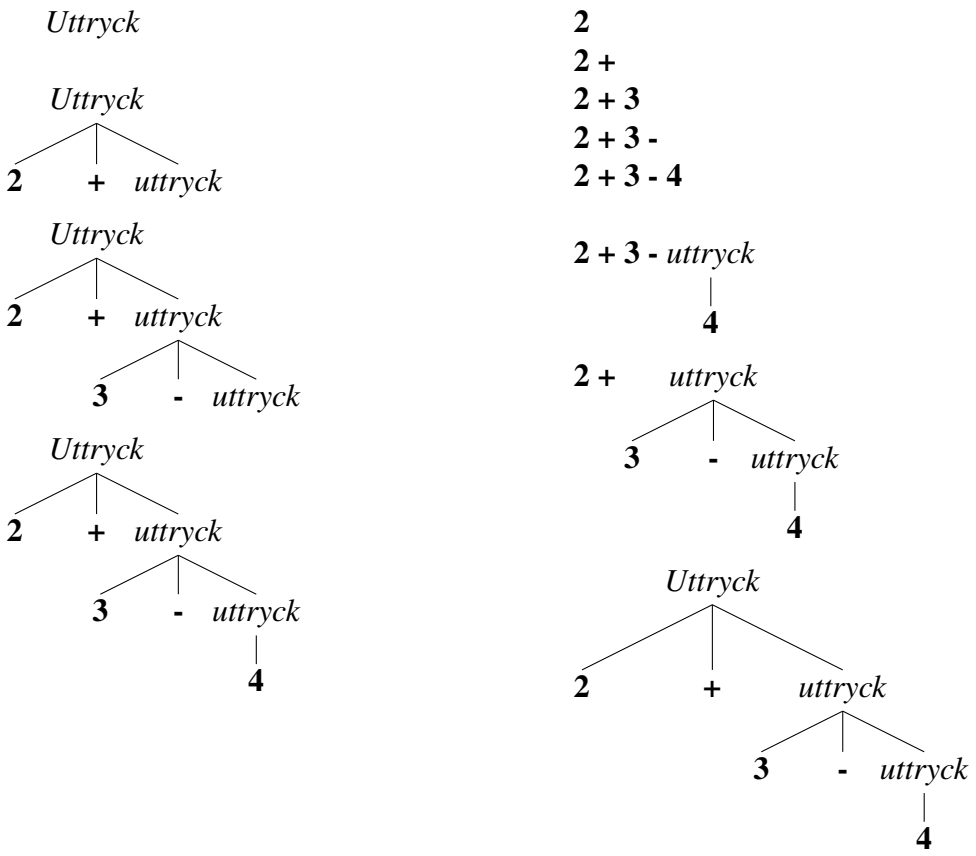
En steg för steg jämförelse mellan LR- och LL-parserns algoritmer visas i figur 7.

3.4.3 LALR-parser

LALR-parsern baserar sig på en LR(0)-parsers struktur men med stöd för *"lookahead"*-tokens. Skillnaden mellan LR-parsers med stöd för *"lookahead"*-tokens och LALR-parsern är antalet tillstånd *"push-down"*-automaten har. Eftersom LALR-parsern kräver färre tillstånd och dessutom kräver mindre minne kan den ses som en förenklad och mer effektiv version (Aho et al. 2006 s. 266).

3.4.4 Rekursivt nedstigande parser

En annan typ av parser som använder sig av *"uppifrån-och-ner"*-algoritmen är den rekursivt nedstigande parsern. Denna parsertyp klarar av att parsa LL-grammatik och är därför vanlig för handskrivna parsers. Istället för att utnyttja en automat fungerar den genom att knyta varje icke-terminal i grammatiken till en egen funktion som ansvarar för att identifiera dess löv i grammatiken. För varje icke-terminal i sin egen regel anropar den rekursivt



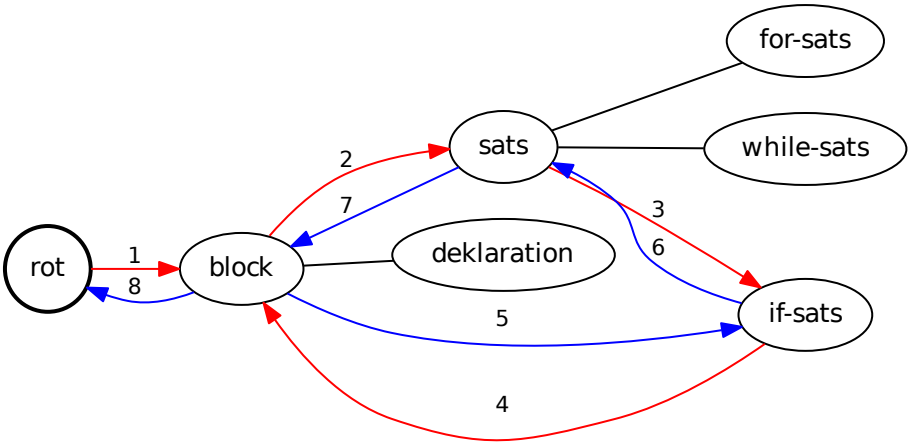
Figur 7. Steg för steg parsning av ett matematiskt uttryck med en "uppifrån-och-ner"-algoritm (vänster) och en "nerifrån-och-upp"-algoritm (höger).

vidare dess funktioner fram till det att rot funktionen får hela syntaxträdet returnerat (Parr 2010 s. 24).

Flödet av en rekursivt nedstigande parser visas i figur 8 där varje nod symboliserar en funktion. En röd pil symboliserar dess funktionsanrop medan en blå pil symboliserar funktionens returnering.

3.5 Eliminering av mångtydighet

Eftersom handskrivna parsers i huvudsak använder sig av *"uppifrån-och-ner"* algoritmer kan de inte parsa alla kontextfria grammatiker. Vissa grammatiker innehåller mångtydigheter som måste elimineras för att denna typ av algoritm skall kunna användas.



Figur 8. Flöde av en rekursivt nedstigande parser steg för steg.

3.5.1 Vänsterrekursion

Ett vanligt problem med grammatiken i en parser med *"uppifrån-och-ner"* algoritmen är vänsterrekursion. Detta innebär att produktionsregel använder sig själv som icke-terminal längst till vänster i sammansättningen. När detta inträffar kommer implementationen att fastna i en oändlig upprepning. För att implementera en LL-parser måste man därför eliminera vänsterrekursion.

Exempelvis är produktionsregeln för kalkylatorn i figur 3 på sida 15 vänsterrekursiv.

$$\begin{aligned}\langle uttryck\rangle ::= &\ \langle tal\rangle \\ |\ &\ (\ \langle uttryck\rangle\ ) \\ |\ &\ \langle uttryck\rangle\ \langle operator\rangle\ \langle uttryck\rangle\end{aligned}$$

Genom att expandera rekursionen i alla existerande villkor som nya villkor får vi:

$$\begin{aligned}\langle uttryck\rangle ::= &\ \langle tal\rangle \\ |\ &\ (\ \langle uttryck\rangle\ ) \\ |\ &\ \langle tal\rangle\ \langle operator\rangle\ \langle uttryck\rangle \\ |\ &\ (\ \langle uttryck\rangle\ )\ \langle operator\rangle\ \langle uttryck\rangle\end{aligned}$$

Detta beskriver fortfarande en identisk grammatik och följande steg är att förenkla uttrycket med en ny produktionsregel som utnyttjar ε, en symbol för att representera ett tomt värde.

$$\langle uttryck\rangle ::= \langle tal\rangle\ \langle uttryck'\rangle$$

$$
\begin{array}{lll}
 & | & \textbf{(}\ \langle uttryck \rangle\ \textbf{)}\ \langle uttryck' \rangle \\
\langle uttryck' \rangle & ::= & \langle tal \rangle \\
 & | & \textbf{(}\ \langle uttryck \rangle\ \textbf{)} \\
 & | & \langle operator \rangle\ \langle uttryck \rangle \\
 & | & \varepsilon
\end{array}
$$

Grammatikens funktion är fortfarande identisk men vänsterrekursionen som är omöjlig att utföra i en *"uppifrån-och-ner"* algoritm är eliminerad.

En förenklad matematisk regel som kan användas för att eliminera vänsterrekursionen $A \rightarrow A\alpha \mid \beta$ är (Aho et al. 2006 s. 212):

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

3.5.2 Vänsterfaktorering

Ytterligare ett problem *"uppifrån-och-ner"*-algoritmen är produktionsregler där två villkor påminner om varandra i början av definitionen.

Exempelvis behövs en obestämd mängd *"lookahead"*-tokens för att identifiera det korrekta villkoret i följande produktionsregel:

$$
\begin{array}{lll}
\langle if \rangle & ::= & \textbf{if}\ \langle uttryck \rangle\ \textbf{then}\ \langle block \rangle\ \textbf{else}\ \langle block \rangle\ \textbf{end} \\
 & | & \textbf{if}\ \langle uttryck \rangle\ \textbf{then}\ \langle block \rangle\ \textbf{end}
\end{array}
$$

Med hjälp av vänsterfaktorering kan vi omvandla regeln till:

$$
\begin{array}{lll}
\langle if \rangle & ::= & \textbf{if}\ \langle uttryck \rangle\ \textbf{then}\ \langle block \rangle\ \langle else \rangle \\
\langle else \rangle & ::= & \textbf{else}\ \langle block \rangle\ \textbf{end} \\
 & | & \textbf{end}
\end{array}
$$

Den matemtiska regeln för liknande uttryck i formen $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ är (Aho et al. 2006

s. 214):

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

3.6 Vidareutveckling av parsertyper

Genom att eliminera mångtydigheter i en grammatik klarar *"uppifrån-och-ner"*-baserade parsers av en större mängd programmeringsspråk. I vissa fall kan mångtydigheter inte elimineras, vilket leder till att parsern behöver ett större antal *"lookahead"*-tokens för att kunna göra ett beslut.

3.6.1 Backtracking

I vissa fall är det användbart att inte ha en begränsad mängd *"lookahead"*-tokens och då kan man använda sig av en metod som heter *"backtracking"* förutsatt det handlar om en rekursivt nedstigande parser.

Implementationen av en *"backtracker"* görs genom att markera positionen var en mångtydighet uppstår och sedan välja ett regelalternativ. När alternativet bevisats vara korrekt eller inkorrekt återgår parsern till den markerade positionen. Om ett felmeddelande skapats försöker parsern med följande alternativ och repeterar detta fram till att en regel visar sig vara korrekt varefter den påbörjar den egentliga parsningen utan att markera positionen (Parr 2010 s. 57).

För att uttrycka att en parser kan se en obegränsad mängd tokens framåt anger man antalet som k i LL(k).

3.6.2 Packrat parsning

En nackdel med den tidigare nämnda *"backtracking"*-metoden är att parsern alltid måste återgå till det tidigare tillståndet och därefter påbörja en andra parsning av regeln. Detta ökar den tidsmängd som krävs för att parsa ett inputvärde.

Packrat parsning är en annan typ av parser vars metod går att implementera för en rekursivt nedstigande parser. Den fungerar likt *"backtracking"*-metoden men skiljer sig genom att memorera resultatet av parsningsträdet samt positionen var parsningen avslutades. När parsern återgår till det tidigare tillståndet granskar den om parsningen var framgångsrik. Om den visar sig vara, använder parsern sig av det redan memorerade parsningsträdet och hoppar vidare till positionen där parsningen avslutades.

3.7 Parser-generatorer

Att implementera en parser för hand är en tidskrävande och felbenägen process. Produktionsregler påminner mycket om varandra och repetitionen att implementera varje regel blir en långdragen uppgift med möjlighet för introduktion av fel vid varje steg. På grund av detta har det blivit populärt att använda en s.k. parser-generator som förser programmeraren med en färdig parser-implementation utgående från en given grammatik (Parr 2010 s. 26).

Med hjälp av en parser-generator kan implementatören upprätthålla grammatiken av ett språk genom att skriva och modifiera en BNF-liknande syntax istället för den egentliga programkoden. Detta har stora fördelar om grammatiken inte är helt stabil eftersom enbart några rader behöver korrigeras vid förändringar. För ett språk med en stabil grammatik existerar fördelarna endast vid implementationsskedet och kan vara en nackdel i situationer som felsökning och skapandet av läsbara felmeddelanden för syntaxfel (Biancuzzi & Warden 2009 s. 175).

Uppdelningen av genererade parsers och handskrivna parsers är delad, de flesta språk använder sig av en genererad parser men några såsom C-språkets GCC (Myers 2004) samt

Luas egna parser (Ierusalimschy et al. 2013) har börjat från en genererad parser och sedan övergått till en handskriven rekursivt nedstigande parser för utökad funktionalitet.

Parser-generatorer existerar för flera språk samt för diverse grammatik- och algoritmtyper. I detta kapitel kommer jag att presentera två parser-generatorer som är vanliga i unix-system.

3.7.1 Lex

Lex är en generator för reguljära grammatiker skapad av Mike Lesk och Eric Schmidt. Verktyget använder en egen grammatiksyntax för att definiera produktionsregler i ett språk och generera sedan en lexer i C eller Ratfor. För varje produktionsregel kan användaren definiera en åtgärd som kommer att exekveras när en regel identifierats. Vanligen är detta C-kod för att returnera värdet till en syntaktisk analysator. Eftersom Lex genererar programkod existerar funktionalitet för att inkludera valfri C-kod som kopieras till resultatfilen. Detta kan användas till att inkludera bibliotek eller definiera funktioner som produktionsreglerna använder (Lesk & Schmidt 2013).

I figur 9 visas en enkel grammatikspecifikation som läser input och identifierar variabeldeklarationer av strängar eller heltal samt ignorerar blanksteg.

```
%option noyywrap
                #include <stdio.h>
%%
[ \t\n]+        ;
[a-zA-Z]+       { printf("Variable: %s\n", yytext); }
[0-9]+          { printf("Integer: %s\n", yytext); }
=               { printf("Equal sign\n"); }
%%

main() {
  yylex();
}
```

Figur 9. En Lex-specifikation för att identifiera variabeldeklarationer av strängar eller heltal.

En fritt tillgänglig version av Lex är Flex som distribueras av bl.a. GNU-projektet. I allmänhet är Flex-kompatibel med Lex och är därför ett populärt alternativ (Brown et al. 1992 s. 279).

3.7.2 Yacc

Eftersom de flesta programmeringsspråk använder sig av en kontextfri grammatik är inte Lex verktyget tillräkligt utan man behöver ytterligare en syntaktisk analysator. Yacc är en parser-generator för kontextfri grammatik skapad av Stephen Johnson. Syntaxen för grammatikspecifikationen liknar mycket Lex. Användaren definierar produktionsregler samt deras åtgärder men har nu möjligheten att använda rekursion. Ytterligare bör man definiera en lexikaliseringsfunktion, exempelvis *yylex()*, samt vilka tokentyper som existerar. Integreringen måste ske både i den valfria lexern och i Yacc-specifikationen, t.ex. med en delad definitionsfil av tokens som lexern returnerar och den Yacc-genererade parsern kan identifiera. Ett pseudokod-exempel på ett språk som tillåter variabeldeklarationer av heltal och strängar visas i figur 10. För att parsern skall fungera måste logiken för *input* samt lexikaliseringen av *VARIABLE*, *STRING*, *INTEGER* och *EQUAL_SIGN* definieras i lexer koden.

GNU-projektet har även skapat en *"copyleft"*-licenseriad version av Yacc som heter GNU bison. Bison är i allmänhet kompatibel med Yacc-syntax och är därför ett populärt alternativ (Brown et al. 1992 s. 277).

3.7.3 Jison

Jison är en JavaScript-programmerad parser-generator för både reguljära grammatiker och kontextfria grammatiker. Verktyget använder samma koncept som Lex/Flex samt Yacc/-Bison och därmed är grammatik-specifikationen samma. Skillnaden är att specifikationen kan innehålla grammatik för både lexern och den syntaktiska analysatorn. Trots att det finns funktionalitet för att generera både en lexer och en syntaktisk analysator tillåter Jison att användaren utnyttjar en separat lexer som följer ett specificerat gränssnitt (Carter

```
%token VARIABLE EQUAL_SIGN INTEGER STRING
%%
statement:    declaration;
declaration:  VARIABLE EQUAL_SIGN primary { printf("Valid declaration"); }
primary:      INTEGER | STRING;
%%

main() {
  do {
    yyparse();
  } while(!feof(input));
}
```

Figur 10. En Yacc-specifikation för variabeldeklarationer av heltal och strängar.

2010).

4 PROGRAMMERINGSSPRÅKET LUA

Lua är ett skriptspråk som kombinerar procedurell programmering med funktionalitet från det funktionella paradigmet och det objektorienterade paradigmet (Ierusalimschy et al. 2013).

Målet av Lua är att använda en enkel syntax med en liten grupp av datastrukturer, att vara snabbt såväl i kompilering som exekvering, att kunna köras på så många plattformer som möjligt och slutligen att vara effektivt för inbäddningen i programvara. Dessa mål anses vara uppnådda enligt Ierusalimschy et al. (2013).

Utgående från dessa mål är Lua ett bra alternativ för konfigurationsspråk och används idag som sådant i bland annat Adobes Photoshop Lightroom och World of Warcraft (About Lua 2013).

4.1 Syntax och uppbyggnad

Trots att ett av Luas mål är dess enkelhet har språket en stor mängd funktionalitet som fås p.g.a. dess flexibilitet och val av datastruktur.

4.1.1 Datatyper

Lua består av 8 datatyper: *nil, boolean, number, string, userdata, table, thread, function*. Eftersom språket använder sig av dynamisk typning kan variabler referera till vilken datatyp som helst. Därmed använder språket inte heller datatypsdefinitioner vid variabeldeklarationer (Ierusalimschy 2006 s. 9).

Tabell är Luas enda datastruktur och denna är egentligen en associativ tabell. Detta betyder att den är dynamiskt växande, kan indexeras av vilken datatyp som helst (förutom *nil*) samt kan även hålla vilket värde som helst. Med hjälp av denna flexibilitet kan tabeller användas både som vanliga tabeller och som hashtabeller (Ierusalimschy 2006 s. 15).

Teckensträngar i Lua har en unik flerradsfunktion. Istället för att definiera teckensträngar med citationstecken kan man använda identifieraren `[[`. Efter dessa hakparenteser fortsätter teckensträngen fram till att de omvända hakparentserna, `]]`, hittas. Innanför dessa hakparenteser kan man ange ett godtyckligt antal likhetstecken för att definiera ett teckensträngsdjup (`[==[ text ]==]`). Med att definiera ett större djup kan en teckensträng omfatta teckensträngsdefinitioner med ett mindre djup. Denna funktionalitet existerar även för kommentarer och är därmed användbar för att kommentera ut kod.

4.1.2 Satser och block

Lua har stöd för de vanliga satserna i exempelvis C såsom if, while, for, break och return. Ytterligare existerar satser såsom repeat, lokala variabeldeklarationer samt flervariabeldeklarationen. En annan mer unik funktionalitet för Lua är att funktioner kan returnera flera värden.

En syntaktisk skillnad till språk i likhet med C är hur sats-block definieras. Istället för att omringa block med vågparenteser ({ och }) använder man sig av terminaler. Initialiseringsterminalen beror på sats-typen men alla satser använder *end* som avslutande terminal.

Ett exempel på hur de olika sats-strukturerna ser ut i Lua visas i figur 11.

4.2 Funktionell programmering

Ett karaktärsdrag från det funktionella paradigmet är att alla luas datatyper är värden av första ordningen. Eftersom en funktion är en datatyp kan den såsom tal och teckensträngar accepteras som funktionsargument samt som returvärde.

När man kombinerar denna funktionalitet med Luas lexikala omfång (scope) möjliggörs användningen av s.k. *"closures"*. *"Closures"* innebär att en funktion vid exekvering har referenser till variabler vid olika händelseförlopp. En *"closure"* har tillgång till egna inre variabler vid exekveringen men också till yttre variabler från den miljö där funktionen

```
i, value = 11, "part" .. "ial"

if i > 10 then
  value = true
elseif i < 10
  value = false
else
  value = nil
end

for k=0,10,1
  i = i + k
end

function test(o)
  return o, 2
end

i, j = test(1)
```

```
while true
  i = i + 1
  if i > 20
    break
  end
end

repeat
  i = i - 1
until i < 10

local days = {"Mon", "Tue", "Wed", "Thu", "Fri
    ", "Sat", "Sun"}
for i,day in ipairs(days) do
  print(day)
end
```

Figur 11. Överblick av Luas sats-typer.

definierats (Ierusalimschy 2006 s. 45).

Användningen av *"closures"* är populärt i funktionella programmeringsspråk för att implementera diverse algoritmer såsom *"curry"*-funktionen.

4.3 Objektorienterad programmering

Trots att Lua inte innehåller klasser eller funktionalitet för arv finns det koncept som möjliggör deras implementation.

4.3.1 Objekt

Luas tabeller är objekt och har därmed också funktionalitet för metoder och för en identitet i betydelsen att ett objekt som ändrar värde alltid är samma objekt och olikt ett annat

objekt med samma värde. Ytterligare kan en metod definiera referensen till sig själv med hjälp av kolon-operatorn. Detta gör tabeller till en nära identisk representation av objekt i ett objektorienterat språk (Ierusalimschy 2006 s. 149). Användningen av Lua objekt visas i figur 12.

```
Person = {name = "Charles"}

function Person:greet(name)
  print("Hello " .. name)
  print("My name is " .. self.name)
end
```

Figur 12. Ett objekt i Lua

4.3.2 Klasser

Lua har inget koncept av klasser, alla objekt definierar sin egen struktur och har inget direkt arv. Vi kan dock simulera klasser och arv genom att använda en struktur lik prototyp-baserat arv. Detta görs med hjälp av s.k. metatabeller (Ierusalimschy 2006 s. 151).

Efter följande deklaration kommer värden att sökas i Person-objektet om det inte hittas hos carl-objektet:

```
setmetatable( carl, {__index = Person});
```

Vad som återstår är att kombinera funktionaliteten för prototyp-arv samt objekt-metoder till något som liknar klasser. Figur 13 visar en klass-lik implementation i Lua.

4.4 Grammatik

Luas grammatik existerar i en EBNF-notation som inkluderats i ursprunglig form i bilaga 1.

```
Person = {}
function Person:new(name)
  o = { name = name }
  setmetatable(o, self)
  self.__index = self
  return o
end

function Person:greet(name)
  print("Hello " .. name)
  print("My name is " .. self.name)
end

carl = Person:new("Carl")
carl: greet("Roger");
```

Exempel output:

```
> Hello Roger
> My name is Carl
```

Figur 13. En arv-implementation i Lua med hjälp av metatabeller.

I version 3.0 övergick Lua från en genererad parser till en handskriven rekursivt nedstigande parser (Ierusalimschy et al. 2013). På grund av detta gjordes det förändringar i grammatiken som inte dokumenterats. Uttrycksparsern har exempelvis omstrukturerats totalt för att kunna beskrivas med en LL(1)-grammatik (Biancuzzi & Warden 2009 s. 175). Trots förändringen är grammatikens validering identisk med den ursprungliga versionen. Förändringen gjordes för att kunna byta ut parsern.

Vid analys av grammatiken kan det konstateras att mycket av grammatiken går att implementera med en LL(1)-grammatik utan större förändringar. Det problem som uppstår är en vänsterrekursion i uttrycksregeln. Detta hanteras i följande kapitel.

5 LUA-PARSERNS IMPLEMENTATION

Implementationen är indelad i fyra komponenter. När parsern startas börjar en syntaktisk analysator parsa input från rot-noden. Det första som görs är ett anrop till den andra komponenten, lexern. Lexern läser input tecken för tecken och tokeniserar värden som sedan returneras till den syntaktiska analysatorn var den används för att välja produktionsregel. Utanför lexern hanteras inte enskilda tecken utan enbart tokens. Utöver dessa två komponenter finns en separat uttrycksparser samt ett delegerare som producerar ett syntaxträd vilket slutligen returneras till programmet som anropat parsern i första hand.

5.1 Lexer

Lexern läser input enligt en reguljär grammatik och gör detta som en deterministisk ändlig automat. När parsningen av en token börjar hämtas det aktuella tecknet i input strängen och lexern går sedan genom en serie villkor för att bestämma var en token slutar och av vilken typ den är. När en token slutat har automaten nått sitt slutliga tillstånd och tokenvärdet returneras. Slutligen återgår automaten till utgångsläget för att vänta på nästa förfrågan.

Tokentyperna som lexern användar är: *"End-of-file"* (EOF), teckensträngs-litteral, nyckelord, identifierare, tal-litteral, symbol, boolean samt nil. Dessa definieras som uppräkningsflaggor (enum flag) för att kunna jämföras med bit operationer. När en token identifieras av lexern returneras den som ett objekt med attribut för tokentypen, dess indexposition i input-teckensträngen, dess rad och kolumn i input samt dess tokenvärde.

5.1.1 Blanksteg och kommentarer

Eftersom blanksteg utgör 40% av implementationens programkod kan det konstateras att lexern spenderar en väsentlig del av sin tid på blanksteg. Det första lexern gör när den anropas är att anropa en funktion som ignorerar alla blankstegstecken som hittas.

Eftersom token-objekt innehåller information om rad och kolumn i input memoreras varje radförändringen i funktionen som ignorerar blanksteg. Ytterligare lagras input-positionen när en rad påbörjas. När ett tokenobjekt skapas subtraheras radens startposition från den nuvarande input-positionen och bildar därmed kolumnen.

Kompilatorer kan ignorera kommentarer eftersom de inte innehåller semantiska värden men eftersom målet med denna implementation är användning för programkodsanalys sparas även kommentarer. Detta görs genom att söka efter avgränsarsymbolen (`--`) och lagra dess information i en separat tabell som slutligen anknyts till det returnerade syntaxträdet.

5.1.2 Identifierare och nyckelord

Från och med Lua 5.2 kan variabelnamn, eller s.k. identifierare, endast bestå av bokstäver, siffror och understreck i ASCII-uppsättningen. Ytterligare kan ett variabelnamn inte börja med en siffra (Ierusalimschy et al. 2011).

Nyckelord är terminaler med samma regler som identifierare och därmed kan de använda samma logik för att avskilja giltiga tecken. En skillnad mellan parser-implementationen och Lua-grammatiken är åtskiljandet mellan nyckelord och datatyperna nil, boolean.

Med att granska grammatiken för tokentyperna kan det utläsas att enbart nil, boolean, nyckelord samt identifierare kan börja med en bokstav. Eftersom lexern arbetar tecken för tecken kan detta användas som en villkorsförgrening. Om en ny token börjar med en bokstav måste det tillhöra någon av dessa fyra tokentyper, ytterligare vet vi att tokentypen slutar där ett tecken inte är en bokstav, en siffra eller ett understreck.

Implementationen av denna fögrerning görs genom att läsa ASCII-koden för tecknet och granska om det är en bokstav eller ett understreck. Om testet lyckas fortsätter lexern att acceptera tecken så länge som teckenkoden tillhör serien för bokstäver, siffror och understreck. När testet misslyckas betyder det att denna token tagit slut och nu kan lexern granska om värdet är lika med ett boolean-värde, ett nil-värde, ett av nyckelorden eller

inte tillhör någon av dessa och därmed är en identifierare.

5.1.3 Symboler

Identifieringen av symboler implementeras med en serie switch- och if-satser vilka granskar vilka tecken framåt i input tillhör en giltig symbol. När den längsta möjliga symbolen identifieras returneras den.

Luas giltiga symboler är:

```
. .. ... : :: ; # = == > >= < <= ~= { } ( ) [ ] ^ % * / - +
```

5.1.4 Litteraler

Lua stöder tal likt vanliga programmeringsspråk men har även stöd för hexadecimaltal innehållande en valfri binär expontent. Implementationen läser input och verifierar att talet är giltigt. När token-värdet identifierats konverteras det till ett flyttal och returneras.

Teckensträngar i Lua börjar antingen med enkla citationstecken (') eller dubbla citationstecken (") och samma avgränsarsymbol förväntas innan raden tar slut. Lua tillåter dock att man inaktiverar en symbol såsom citations- eller radbrytningstecken genom att använda ett omvänt snedstreck (\). En annan användning för det omvända snedstrecket är att definiera en styrkodsekvens. När lexern hittar ett snedstreck inom en teckensträngslitteral granskas först giltiga styrkodssekvenser och om ingen identifieras inaktiveras följande tecken i input.

5.2 Syntaktisk analysator

Den syntaktiska analysatorn är implementerad som en rekursivt nedstigande parser för en LL(1)-grammatik. Parsern börjar från rot-noden. Denna nod består av en block-nod med en obestämd mängd satser-noder. Varje nodtyp, eller regel, är implementerad som en funktion som rekursivt anropar andra regelfunktioner och slutligen returnerar ett syntaxträd

över sin struktur. När input strängen tagit slut har det slutgiltiga syntaxträdet returnerats till rot-noden och kan därefter förmedlas till programmet som begärt informationen.

Mycket av den syntaktiska analysatorns grammatik är samma som den ursprungliga Lua-grammatiken. Dock har vissa finjusteringar gjorts för att bland annat kunna åtskilja variabeldeklarationer från funktionsanrop.

5.2.1 Hjälp-funktioner

För att underlätta hanteringen av produktionsregler samt kommunikationen med lexer-komponenten används vissa hjälp-funktioner.

Kommunikationen med lexern sker med en *next()*-funktion. Funktionen ansvarar även för att parsern skall ha tillgång till en *"lookahead"*-token som krävs för att syntaxen skall kunna analyseras. När funktionen anropas ersätts den aktuella token med *"lookahead"*-token. Efter detta anropas lexern och dess resultat sparas som *"lookahead"*-token. Dessa tokens lagras som variabler tillgängliga för hela implementationen.

Ytterligare existerar två viktiga hjälp-funktioner för att hålla implementationen mer läslig. *expect()*-funktionen anropas med ett tokenvärde, som jämförs med den aktuella token. Stämmer bägge överens hämtas nästa token mha. *next()* metoden. Om de inte överenstämmer skapas ett felmeddelande eftersom syntaxen inte är korrekt. Denna funktion används för att granska terminaler när en produktionsregel hittats.

Den andra väsentliga hjälp-funktionen är *consume()* som likt *expect()* granskar om ett tokenvärde överenstämmer med den aktuella token. Skillnaden är att *consume()* inte skapar ett felmeddelande utan returnerar istället sant eller falskt. Detta används för att granska valfria delar av en produktionsregel, t.ex. if-satser som har en valfri *else* komponent.

5.2.2 EBNF-notation till JavaScript-kod

För att minimera komplexiteten av parsern har implementationen delat upp produktionsregeln ⟨*sats*⟩ så att den enbart identifierar vilken typ av sats det handlar om utgående från den första terminalen. Efter att satsen identifieras anropas funktionen knyten till satstypen.

Ytterligare optimerar jag sats-grammatiken med att ordna reglerna enligt populariteten baserad på en provtagning av ett exempelprogram (bilaga 2).

Den nya ⟨*sats*⟩-grammatiken visas i figur 14.

⟨*sats*⟩ ::= **local** ⟨*local-sats*⟩
| **if** ⟨*if-sats*⟩
| **return** ⟨*retur-sats*⟩
| **function** ⟨*funktion-sats*⟩
| **while** ⟨*while-sats*⟩
| **for** ⟨*for-sats*⟩
| **repeat** ⟨*repeat-sats*⟩
| **break** ⟨*break-sats*⟩
| **do** ⟨*do-sats*⟩
| **goto** ⟨*goto-sats*⟩
| **::** ⟨*label-sats*⟩
| **;**
| ⟨*deklaration-eller-anrop-sats*⟩

Figur 14. Parser-implementationens produktionsregel för en sats

Processen att omvandla EBNF-notation är en enkel process när eliminieringen av ε gjorts. Alternering implementeras som switch-satser, villkor som if-satser och repetition som while-, for- eller do-satser.

Genom att omvandla varje produktionsregel till en JavaScript-funktion enligt detta mönster fullständiggörs den syntaktiska analysatorn. Som exempel omvandlas EBNF-notationen för if- samt while-satser till dess JavaScript-motsvarighet i figur 15.

⟨*if-sats*⟩ ::= **if** ⟨*uttryck*⟩ **then** ⟨*block*⟩ {**elseif** ⟨*uttryck*⟩ **then** ⟨*block*⟩} [**else** ⟨*block*⟩] **end**

⟨*while-sats*⟩ ::= **while** ⟨*uttryck*⟩ **do** ⟨*block*⟩ **end**

```
function parseIfStatement() {
  clauses = [];

  do {
    condition = parseExpression();
    expect('then');
    block = parseBlock();
    clauses.push({
      condition: condition,
      block: block
    });
  } while (consume('elseif'));

  if (consume('else')) {
    clauses.push({
      block: parseBlock()
    });
  }
  expect('end');

  return {
    type: 'IfStatement',
    clauses: clauses
  };
}
```

```
function parseWhileStatement() {
  condition = parseExpression();
  expect('do');
  body = parseBlock();
  expect('end');

  return {
    type: 'WhileStatement',
    condition: condition,
    block: block
  };
}
```

Figur 15. Produktionsregelerna för if- och while-satser i JavaScript-kod

5.3 Uttryckparser

Lua-grammatikens uttrycksregel, $\langle expr \rangle$, utnyttjar vänsterrekursion, vilket inte är möjligt i rekursivt nedstigande parsers eftersom de hamnar i en oändlig upprepning.

Uttrycksregeln består av en serie andra icke-terminaler vilkas funktioner är svåra att få ett grepp om och därför kommer vi att expandera och analysera alla delregler för att slutligen få en fungerande och förhoppningsvis mer organiserad produktionsregel för uttryck.

Processen för att införa detta kommer bestå av tre steg, där det det första steget visar den ursprungliga regeln, det andra steget visar resultatet från att eliminera vänsterrekursion och det tredje steget visar regeln skriven utan epsilon. Med att omforma regeln till att använda repetition istället för epsilon är regeln färdig för att översättas till programkod.

5.3.1 Gruppering av uttryck

För att lättare arbeta med definitioner indelas uttryck i binära och unära operationer, prefix uttryck samt primära uttryck. Primära uttryck består av de kvarstående alternativen, huvudsakligen datatyper.

Ursprungsregel

$$\langle exp \rangle \quad ::= \langle primaryexp \rangle \mid \langle prefixexp \rangle \mid \langle exp \rangle \ \langle binop \rangle \ \langle exp \rangle \mid \langle unop \rangle \ \langle exp \rangle$$
$$\langle primaryexp \rangle ::= \ \ldots$$
$$\langle prefixexp \rangle ::= \ \ldots$$

Eliminering av vänsterrekursion

$$\langle exp \rangle \quad ::= \langle primaryexp \rangle \ \langle exp' \rangle \mid \langle prefixexp \rangle \ \langle exp' \rangle \mid \langle unop \rangle \ \langle exp \rangle$$
$$\langle exp' \rangle \quad ::= \langle binop \rangle \ \langle exp \rangle \mid \varepsilon$$

Resultat

⟨*exp*⟩ ::= (⟨*unop*⟩ ⟨*exp*⟩ | ⟨*primaryexp*⟩ | ⟨*prefixexp*⟩) { ⟨*binop*⟩ ⟨*exp*⟩ }

5.3.2 Analys av prefixuttryck

Ursprungsregel

⟨*prefixexp*⟩ ::= ⟨*var*⟩ | ⟨*functioncall*⟩ | **(** ⟨*exp*⟩ **)**

⟨*var*⟩ ::= Name | ⟨*prefixexp*⟩ **[** ⟨*exp*⟩ **]** | ⟨*prefixexp*⟩ **.** Name

⟨*functoincall*⟩ ::= ⟨*prefixexp*⟩ ⟨*args*⟩ | ⟨*prefixexp*⟩ **:** Name ⟨*args*⟩

⟨*args*⟩ ::= **(** [⟨*explist*⟩] **)** | ⟨*tableconstructor*⟩ | String

Den fullständiga regeln för ⟨*args*⟩ finns i bilaga 1, men för att hålla grammatiken enkel utelämnas dess helhet här och vi konstaterar endast att regeln inte behöver bearbetas mer. Vi expanderar de övriga delreglerna för att få en bättre överblick.

⟨*prefixexp*⟩ ::= Name | ⟨*prefixexp*⟩ **[** ⟨*exp*⟩ **]** | ⟨*prefixexp*⟩ **.** Name
| ⟨*prefixexp*⟩ ⟨*args*⟩ | ⟨*prefixexp*⟩ **:** Name ⟨*args*⟩
| **(** ⟨*exp*⟩ **)**

Eliminering av vänsterrekursion

⟨*prefixexp*⟩ ::= Name ⟨*prefixexp'*⟩ | **(** ⟨*exp*⟩ **)** ⟨*prefixexp'*⟩

⟨*prefixexp'*⟩ ::= **[** ⟨*exp*⟩ **]** | **.** Name ⟨*args*⟩ | **:** Name ⟨*args*⟩ | ε

Resultat

⟨*prefixexp*⟩ ::= (Name | **(** ⟨*exp*⟩ **)**) { **[** exp **]** | **.** Name | **:** Name ⟨*args*⟩ | ⟨*args*⟩ }

5.3.3 Sammanknytning

Med hjälp av dessa modifieringar har vi nu en produktionsregel för uttryck som inte använder sig av vänsterrekursion och kan därmed användas i en rekursivt nedstigande parser.

⟨*exp*⟩ ::= (⟨*unop*⟩ ⟨*exp*⟩ | ⟨*primary*⟩ | ⟨*prefixexp*⟩) { ⟨*binop*⟩ ⟨*exp*⟩ }

⟨*primary*⟩ ::= **nil** | **false** | **true** | Number | String | **...** | ⟨*functiondef*⟩ | ⟨*tableconstructor*⟩

⟨*prefixexp*⟩ ::= (Name | **(** ⟨*exp*⟩ **)**) { [exp] | **.** Name | **:** Name ⟨*args*⟩ | ⟨*args*⟩ }

5.3.4 Operationsprioritet

Operationsprioritet innebär den logik som bestämmer att multiplikation prioriteras högre än addition och därmed grupperar uttryck såsom *5 + 5 * 2* till *5 + (5 * 2)* istället för *(5 + 5) * 2*. Lua-manualen nämner att grammatiken inte beskriver operationsprioritet (Ierusalimschy et al. 2011) men en lista över prioriteterna kan hittas i källkoden av Luas egna parser (lparse.c 2011) samt i tabell 1.

Tabell 1. Tabell över Luas operatorprioritet

Operator	Prioritet
`^`	10
`* / %`	7
`+ -`	6
`..`	5
`== < <= > >= ~=`	3
`and`	2
`or`	1

En implementationsmöjlighet som Lua själv använder är att spjälka ut funktionen som parsar uttryck till en ny funktion som tar emot operatorprioriteten som ett argument med den lägsta prioriteten som ursprungsläge. Funktionen spjälker sedan uttrycket i ett vänsterled, ett högerled och operatorn mellan dessa. Om operatorns prioritet är högre än prioriteten som givits som argument parsar funktionen rekursivt operationens högerled och skapar en ny operationsgruppering så länge som prioriteten är högre. Om operatorns prioritet är lägre än eller samma som argumentets prioritet kommer iterationen att avbrytas och det nuvarande uttrycket att returneras till det högra ledet. Eftersom ursprungslägets prioritet är den lägsta kommer alltid ett uttryck att fortsätta så länge som en operator kan hittas.

Denna algoritm kallas för *"operator precedence"* parsning och använder sig av *"nerifrån-och-upp"*-strategin. Pseudokoden för en implementation visas i figur 16 och ett grupperingsexempel visas i figur 17.

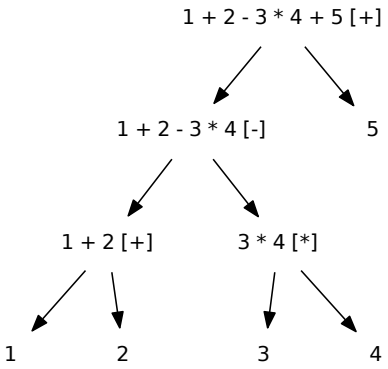
```
function parseSubExpression(minPrecedence = 0) {
  expression = parseOperand();

  while (true) {
    operator = parseToken();
    precedence = getPrecedence(operator);

    if (precedence > minPrecedene) {
      var right = parseSubExpression(precedence);
      expression = binaryExpression(operator, expression, right);
    } else {
      break;
    }
  }
  return expression;
}
```

Figur 16. En "operator precedence"-uttryckstolkare.

Ett steg för steg exempel av parsningen för uttrycket 1 + 2 - 3 * 4 + 5.

Punkterna 1 till 9 beskriver stegen för det ursprungliga funktionsanropet medan en underlista representerar stegen i ett rekursivt funktionsanrop.

1. Vänstra ledet har värdet 1 och operatorn är + med en prioritet på 1.
2. Eftersom prioriteten är högre än ursprungsläget anropar funktionen sig själv med den nya prioriteten.
    (a) Vänstra ledet är 2 och operatorn är - med en prioritet på 1.
    (b) Eftersom prioriteten är samma som föregående returneras värdet 2.
3. Högra ledet är 2.
4. En gruppering sker och vänstra ledet blir nu (1 + 2).
5. Följande operator är - med en prioritet på 1 som är mer än 0 och därmed anropar funktionen sig själv.
    (a) Vänstra ledet är 3 och operatorn är * med en prioritet på 2.
    (b) Eftersom 2 är högre än 1 anropar funktionen sig själv.
        i. Vänstra ledet är 4 och operatorn är + med en prioritet på 1.



*Figur 17. Grupperingsträd av det matematiska uttrycket 1 + 2 - 3 * 4 + 5.*

ii. Eftersom 1 är lägre 2 returneras värdet 4.

(c) Högra ledet är nu 4.

(d) En gruppering sker och vänstra ledet blir nu (3 * 4).

(e) Operatorn är + med en prioritet på 1 vilket är samma som argumentets prioritet och därmed returnerar funktionen värdet (3 * 4).

6. Högra ledet är nu (3 * 4) och en gruppering sker så att vänstra ledet blir ((1 + 2) - (3 * 4)).
7. Följande operator är + med en prioritet på 1 som är mer än 0 och därmed anropar funktionen sig själv.
   (a) Vänstra ledet är 5 och eftersom ingen operator hittas returneras värdet.
8. Högra ledet är nu 5 och grupperingen blir (((1 + 2) - (3 * 4)) + (5)).
9. Eftersom ingen operator kan hittas returneras det slutgiltiga uttrycket.

5.3.5 Operatorassociativitet

Operatorassociativitet används för att gruppera operatorer med samma prioritet och utan användningen av parenteser. De vanliga matematiska operatorerna har ingen associativitet eftersom $(1 * 2) * 3$ har samma resultat som $1 * (2 * 3)$. I uttrycksparsern från figur 16 hanteras dessa operatorer som vänsterassociativa och grupperas därmed som $(1 * 2) * 3)$. Detta kan implementeras godtyckligt.

Lua har dock två operatorer med högerassociativitet, nämligen operatorn för sammanfogningen av teckensträngar (..) samt potensoperatorn (^). Om uttrycket *5^4^3^2* grupperas

från vänster likt addition skulle det se ut som *((5^4)^3)^2*, vilket inte har samma resultat som det korrekta högerassociativa *5^(4^(3^(2)))*.

Implementationen för detta visas i figur 18 och är en modifiering av algoritmen i figur 16. Lösningen utgörs av att subtrahera 1 från operatorprioriteten om en högerassociativ regel hittas innan det högra ledet parsas. Detta orsakar en omvändning i grupperingen.

```
while (true) {
  operator = parseToken();
  precedence = getPrecedence(operator);

  if (precedence > minPrecedene) {
    if (parseRightAssociative(operator)) precedence -= 1;
    var right = parseSubExpression(precedence);
    expression = binaryExpression(operator, expression, right);
  } else {
    break;
  }
}
```

Figur 18. Implementation för högerassociativa operatorer.

5.3.6 Resultat

Resultatet av denna omvandling visar sig vara en nära exakt kopia av Luas egna uttrycksparser med vissa skillnader för operatorassociativitet. Det kan konstateras att skaparna använt sig av metoden för eliminering av vänsterrekursion för att omvandla sin tidigare Yacc-skapade parser till en handskriven rekursivt nedstigande parser.

5.4 Abstrakt syntaxträd

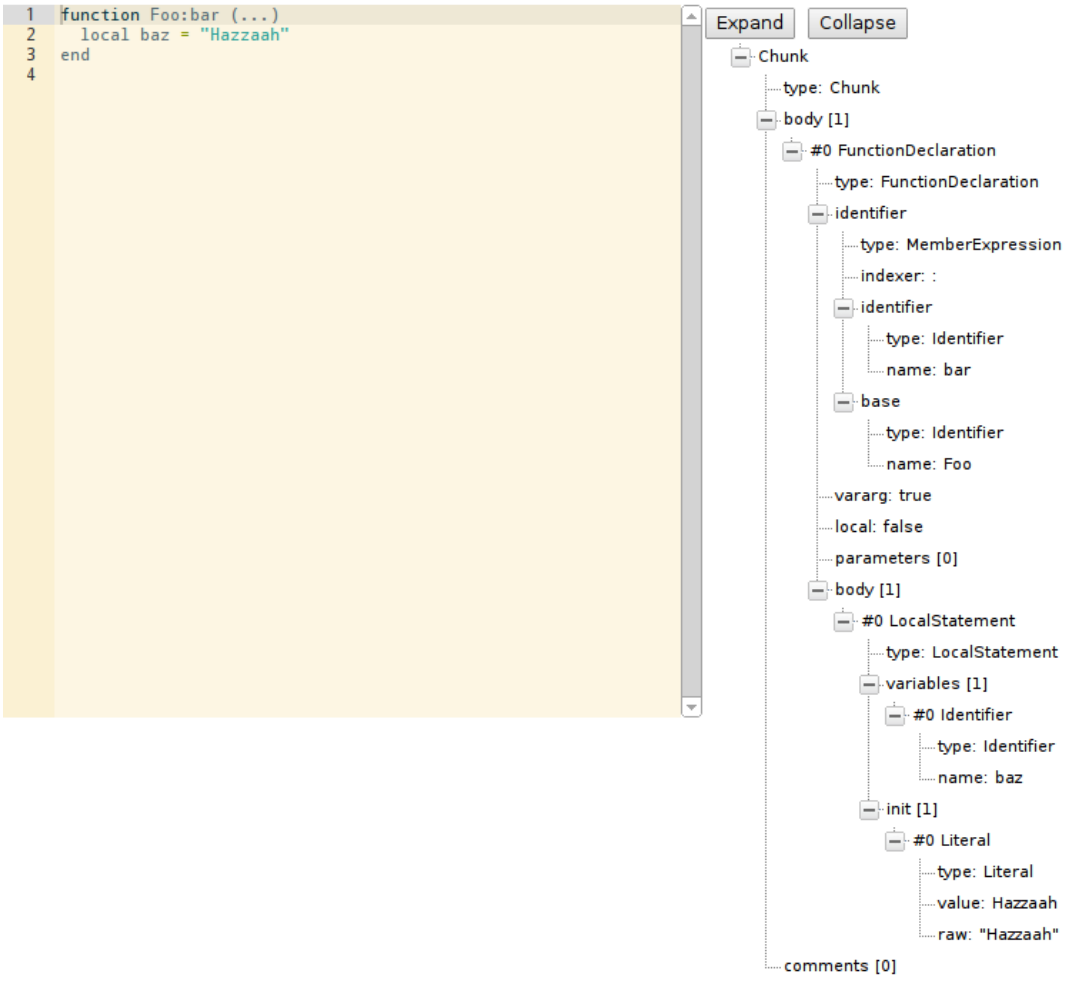
Eftersom implementationens mål är att användas som programkodanalysator representeras parsningens syntaxträd som ett abstrakt syntaxträd. Varje nod i trädet innehåller ett obligatoriskt typ-attribut samt en serie andra attribut som är nod-specifika. Dessa noder är implementerade som objekt-litteraler och returneras till det anropande programmet bör-

jade från programmets rot, Luas s.k. *"chunk"*.

Implementationen av det abstrakta syntaxträdet är uppbyggt med att varje nod existerar som en funktion vars ansvar är att skapa en representation av informationen. Under den syntaktiska analysen delegeras skapandet av noder till dessa funktioner. För att det anropande programmet skall ha möjlighet att så enkelt som möjligt omstrukturera syntaxträdet tillgängliggörs dessa funktioner via parserns API.

Eftersom syntaxträdet representerar en mycket låg nivå i Luas egna parser har jag valt att ta inspiration av Mozillas Parser API (SpiderMonkey - Parser API 2012).

Ett exempel på syntaxträd genererat från en parsning visas i figur 19.



Figur 19. Abstrakt syntaxträd (höger) för parsningen av en Lua-funktion (vänster).

6 PRESTANDAOPTIMERING

Tolkade språk såsom JavaScript bör fokusera mer tid på prestandaoptimering än kompilerade språk, eftersom de evaluerar programkod vid exekvering. Dessutom är JavaScript ett språk som huvudsakligen körs i webbläsare med varierande prestanda. Mindre applikationer med ett fåtal funktionsanrop behöver vanligen inte prioritera prestandaoptimering eftersom webbläsarens JavaScript-motor är tillräkligt effektiv för att inte påverka användarupplevelsen. Målet med detta examensarbete handlar dock om att implementera en parser som kan användas i nätbaserade programkodsredigerare, exempelvis som en *"syntax highlighter"*. Eftersom en *"syntax highlighter"* behöver uppdateras vid varje programkodsförändring kan antalet funktionsanrop snabbt stiga och påverka användarupplevelsen. När användaren gör en uppdatering förväntas en omedelbar respons och därmed måste en parser ämnad för webbläsare uppfylla detta krav.

Insamligen av data utgörs av att parsa ett exempelprogram vid specifika revisioner av implementation. Utgående från resultatet uppdateras implementationen och det nya resultatet granskas.

6.1 JavaScript-motorn V8

För att möjliggöra automation av provtaging utförs dessa test med Node.js som använder sig av JavaScript-motorn V8. V8 är en JavaScript-motor med öppen källkod som skapats av Google ämnad för deras webbläsare Google Chrome.

V8 består av en optimeringskompilator kallad Crankshft. Syftet med denna kompontent är att identifiera och optimera enbart de funktioner som är väsentliga för exekveringen av JavaScript-kod. Processen att optimera programkod i en tolk är tidskrävande och därmed bör optimeringen enbart fokusera på funktioner som har en inverkan på exekveringstiden.

Processen för Crankshaft utgörs av fyra steg där det första innebär att baskompilatorn kompilerar programkoden som normalt. Efter detta börjar en s.k. *"runtime-profiler"* över-

vaka körningen av programmet för att identifiera de väsentliga funktionerna. När en funktion anses vara väsentlig för exekveringen börjar en s.k. girig omkompilering.

En girig kompilering innebär att kompilatorn inte bryr sig om optimeringen kommer att lyckas eller inte. Istället för att analysera programkoden väljer kompilatorn att testa sig fram och sedan återgå till den ursprungliga maskinkoden om optimeringen misslyckades (Millikin & Schneider 2010). Denna åtgärd kallas för avoptimering och inträffar aktivt i körningen av JavaScript eftersom språket är dynamiskt och en avoptimering sker varje gång en datatyp konverteras. Ytterligare existerar det specifika strukturer som kompilatorn inte kan optimera, exempelvis try-satser samt vissa typer av switch-satser. Även här måste en avoptimering ske men för dessa struktur-relaterade problem kallas det för en s.k. *"bailout"*.

För att identifiera dessa avoptimeringar och *"bailouts"* har V8 diverse kommandoradsflaggor som kan användas. Dessa kan även användas för att inspektera funktionsanrop och händelseförlopp.

6.1.1 Kommandoradsflaggor

Nedan följer en lista på de kommandoradsflaggor som används i detta kapitel (V8Profiler - How to profile V8 using the built-in profiler 2012).

--prof

Skapar en v8.log fil bestående av provtagning från JavaScript-stacken samt C++-stacken. Med hjälp av verktyget tick-processor som följer med V8 kan användaren skapa en översikt av denna information. Översikten återger hur stor andel av den totala exekveringen som utgörs av varje funktion.

--trace_bailout

Anger orsaken till varför en funktion avoptimerats med en *"bailout"*.

--trace_inlining

Registrerar varje gång Crankshaft försöker omkompilera en funktion med hjälp av

en teknik som kallas *"function inlining"*. Denna optimeringsteknik innebär att kompilatorn ersätter ett funktionsanrop med själva funktionsinnehållet och därmed sparar kostnaden av själva andropet samt dess returnering. När en *"function inlining"* utförs möjliggör det ytterligare optimeringsmöjligheter eftersom programkoden nu är mer kompakt och introducerar satser som möjligen kan kombineras med omliggande programkod.

6.2 Provtagning

Följande moment har genomförts för att optimera prestandan av parsern fram till att resultatet har en relativ förbättring större än 10%. Alla test har utförts på en dual-core 2.6 GHz dator med ett Linux-baserat operativsystem.

6.2.1 Undvikandet av "bailouts"

Trots att *"bailouts"* möjliggör en allmän prestandaförbättring hos V8 är det klokt att undvika avoptimeringen eftersom den kostar en del prestanda och kommer att återupprepas vid varje exekvering.

Genom att analysera resultatet från en parsning med kommandoradsflaggan *--trace_bailout* i bilaga 6 kan två *"bailout"*-avoptimeringar utläsas. Funktionerna som orsakat dessa är *isUnary()* samt *parsePrimaryExpression()* som bägge innehåller en ooptimerbar switch-sats. Node.js v0.8.16 har inte möjlighet att optimera funktioner som innehåller en switch-sats med icke-litterala labels, d.v.s. variabler.

Detta är ett enkelt problem att lösa eftersom avoptimeringen inte sker för "if-sats"-motsvarigheten och därmed kan programkoden optimeras enligt figur 20.

```
function isUnary(token) {
  switch (token.type) {
    case Tokens.Punctuator:
      return ~'#-'.indexOf(token.value);
    case Tokens.Keyword:
      return token.value === 'not';
  }
  return false;
}
```

```
function isUnary(token) {
  if (token.type === Tokens.Punctuator)
    return ~'#-'.indexOf(token.value);
  if (token.type === Tokens.Keyword)
    return token.value === 'not';
}
```

Figur 20. Förändring av funktionen isUnary från icke-optimerbar (vänster) till optimerbar (höger).

6.2.2 Teckenkoder istället för reguljära uttryck

Genom att analysera resultat i bilaga 7 som fås från provtagningsverktyget i V8 framgår det att 53.5% av en hel exekvering går åt till reguljära uttryck som används för att identifiera blanksteg och giltiga identifierare.

Tidigare prestandamätningar (charCodeAt vs charAt 2011) visar ytterligare att jämförelser på teckenkoder istället för på teckensträngar kan vara mer än dubbelt snabbare. Orsaken för denna prestandaskillnad är att JavaScript-motorn utnyttjar binära operationer när den jämför teckenkoder. På grund av denna prestandaskillnad ersätts reguljära uttryck med switch- och if-satser som jämför teckenkoder.

6.2.3 Optimering av sträng-funktioner

Efter att de tidigare optimeringarna genomförts betonas nu 2 huvudsakliga funktioner i provtagningsanalysen i bilaga 8. Dessa funktioner, dvs. *StringAddStub* som är en maskinkodsmetod för att sammanfoga strängar och *CompareICStub* som är en maskinkodsmetod för att jämföra strängar, upptar 20% av parsningsprocessens exekveringstid.

StringAddStub orsakas av att lexern sammanfogar token-värden för kommentarer, tal och identifierare genom att iterera över varje tecken så länge som en avgränsare inte påträffas. Denna procedur går att förenkla genom att istället memorera var värdet börjar och se-

dan iterera fram till avgränsaren. När avgränsaren hittats kan värdet identifieras från dess startposition till dess avgränsare enligt exemplet i figur 21.

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

```
var start = index;
while (isIdentPart(input.charCodeAt(index)))
  index++;

var value = input.slice(start, index);
```

Figur 21. Optimering av vänstra funktionens strängsammanfogning till högra funktionens identifiering.

Orsaken till den långsamma *CompareICStub*-operationen visar sig vara en *"function inlining"* i uttrycksparsern. Funktionen ansvarar för att ange prioriteten av en given operator utgående från en switch-sats med strängjämförelser. Eftersom funktionen körs för varje uttryck som hittas valde jag att uppoffra läsligheten med att istället jämföra teckenkoder en efter en enligt figur 22. Efter att optimeringen gjorts valideras ännu att funktionen är tillräkligt liten för *"function inlining"*, vilket den visar sig vara.

6.3 Resultat

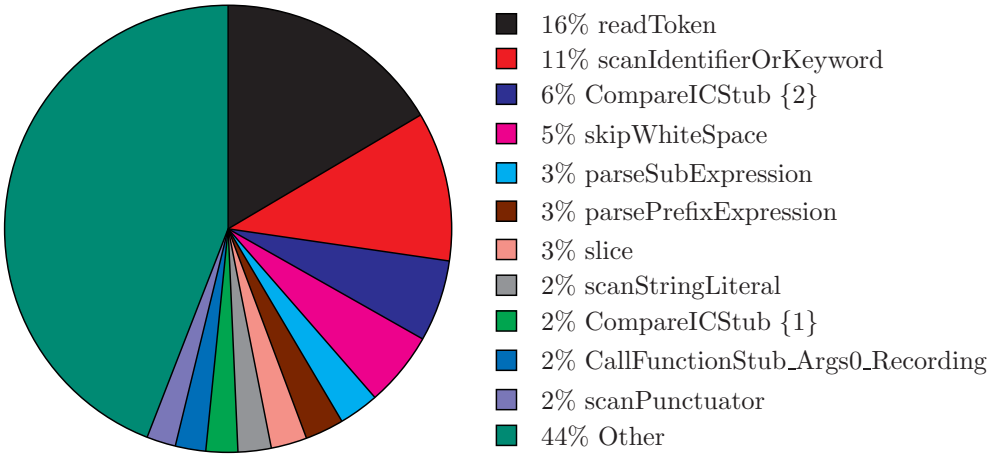
Efter optimeringarna i detta kapitel har de huvudsakliga flaskhalsarna avlägsnats från parsern. En överblick av exekveringstiden för de nuvarande funktionerna visas i figur 23. Funktionerna *readToken()* och *scanIdentifierOrKeyword()* som kräver mest exekveringstid är redan optimerade och kan inte förbättras.

En översikt av förbättringsskillnaderna från kapitlets prestandaoptimeringar visas i figur 24.

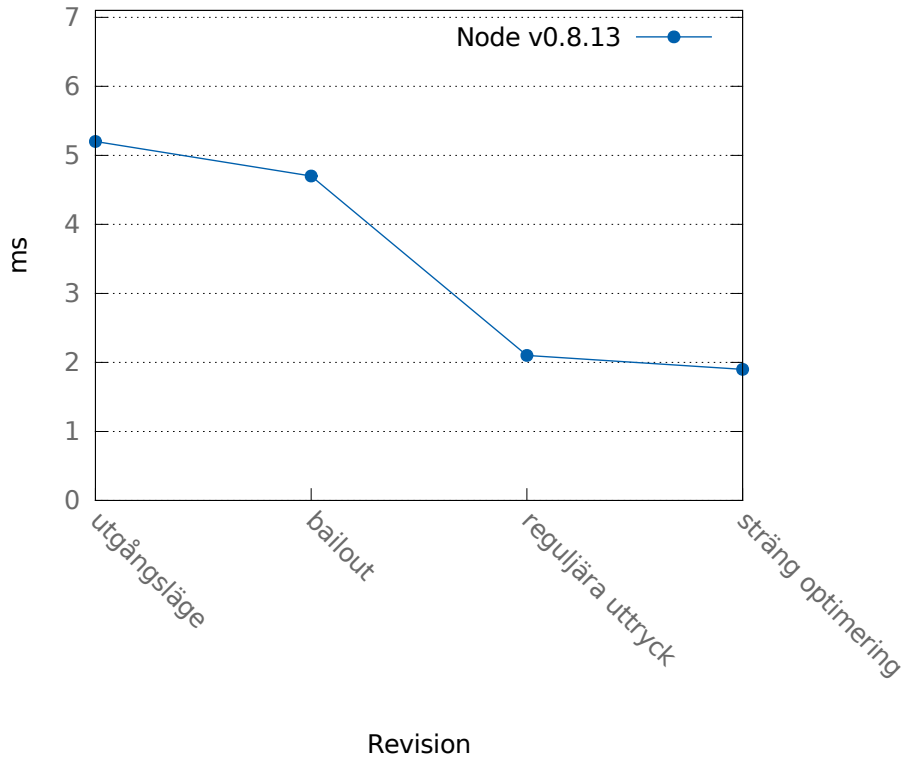
```
function binaryPrecedence(operator) {
  switch (operator) {
    case '^':
      return 10;
    case '*': case '/': case '%':
      return 7;
    case '+': case '-':
      return 6;
    case '..':
      return 5;
    case '<': case '<=': case '>':
    case '>=': case '==': case '~=':
      return 3;
    case 'and':
      return 2;
    case 'or':
      return 1;
  }
  return 0;
}
```

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

    if (1 === length) {
      switch (character) {
        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 (character) {
        case 46: return 5; // ..
        case 60: case 62: case 61: case 126:
          return 3; // <= >= == ~=
        case 111: return 1; // or
      }
    } else if (97 === character
      && 'and' === operator)
      return 2;
    return 0;
}
```

Figur 22. Optimering från jämförelse av tecken (vänster) till jämförelse av teckenkoder (höger).



Figur 23. Slutgiltig översikt av exekveringstiden för parser-implementations funktioner.



Figur 24. Översikt av moment i.o.m. parser-implementationens prestandaoptimering.

7 RESULTAT

För att mäta resultat av examensarbetets parser-implementation kommer jag att göra en prestandaanalys av alternativa parsers.

7.1 Jison-genererad parser

Parser-generatorn Jison kommer att användas för att generera en Lua-parser som utnyttjar en *"nerifrån-och-upp"*-algoritm. Grammatiken som används är baserad på Lua.js-projektets Lua-grammatik. Eftersom Lua.js kompilerar Lua-kod till JavaScript-kod har grammatiken förenklats till att enbart parsa och inte lagra någon information från processen. Den slutliga grammatiken visas i bilaga 3.

I det tidigare kapitlet framgick det att lexern kan utgöra 40% av en parsningsprocess. Eftersom den Jison-genererade lexern använder sig av reguljära uttryck kan den förväntas vara ineffektiv. I ett försök att förbättra resultatet kommer prestandaanalysen av den genererade parsern existera i två versioner. En version kommer att använda den fullständiga parsern medan en annan kommer att utnyttja lexern som utvecklats i detta examensarbete.

7.2 LuaMinify

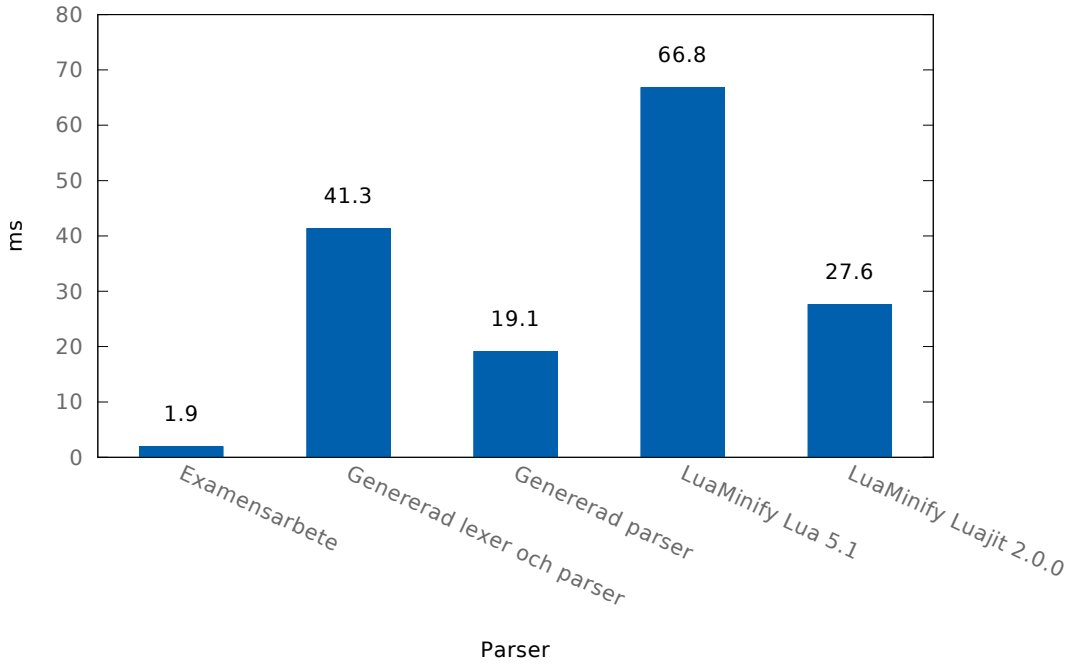
LuaMinify är en källkods-komprimerare för Lua implementerad i Lua. Projektet är gjort som ett hobbyprojekt och använder sig en av handskriven rekursivt nedstigande parser. Implementationen av parsern har inte prioriterat optimering, vilket är förståeligt eftersom en källkods-komprimerare inte har ett behov av att vara snabb. För att få en mer komplett bild av prestandan för handskrivna parsers inkluderas denna parser i analysen.

Prestandaanalysen av parsern från LuaMinify kommer att göras i två versioner. En version kommer att köras med Luas standardkompilator och en annan med Luas JIT-kompilator.

7.3 Mätning

JavaScript-mätningar görs i Node.js v0.8.13 med Benchmark.js-verktyget och skriptet för mätningen visas i bilaga 4. Lua-mätningen är implementerad i Lua och använder sig av socket-biblioteket för att få tillgång till en högre noggrannhet. Skriptet för mätningen visas i bilaga 5. Alla parsningar görs på ParseLua.lua från LuaMinify-projektet.

7.4 Slutsats



Figur 25. Prestandaresultat från parsning av ParseLua.lua.

Resultatet från tabell 2 visualiseras i figur 25 och visar att examensarbetets parser-implementation är den snabbaste både när det kommer till lexikal analys och syntaktisk analys.

Tabell 2. Resultat från prestandaanalysen mellan parsers.

Parser	Tid (ms)	Standardavvikelse
Examensarbetets implementation	1.9	0.1
Fullständig Jison-genererad implementation	41.3	1.7
Jison-genererad implementation utan lexer	19.1	0.4
LuaMinify med Lua 5.1	66.8	4.0
LuaMinify med Luajit 2.0.0	27.6	2.8

Parsern som genererats av Jison visar sig vara 20 gånger långsammare än examensarbe-

tets implementation. Genom att byta ut lexern med examensarbetets lexer halveras exekveringstiden för testet. Lexer är den mest prestandaoptimerade komponenten i implementationen medan den syntaktiska analysatorn är en typisk rekursivt nedstigande parser. Eftersom den genererade parsern fortfarande är långsam jämfört med examensarbetets implementation kan det konstateras att lösningen inte är passande för ett prestandakritiskt projekt.

Parsern tillhörande LuaMinify visar sig vara mer än 30 gånger långsammare än examensarbetets parser när den exekveras med Lua 5.1. Detta resultat är svårt att tolka eftersom implementationerna använder sig av skilda språk. Vid en exekvering med LuaJIT som är känd för att vara en mycket snabb tolk är LuaMinify-parsern 15 gånger långsammare är examensarbetets parser. I en diskussion från 2010 hävdar LuaJIT skaparen Mike Pall (Pall 2010) att skräpinsamling är ett problem för LuaJIT jämfört med V8. Eftersom parsningsprocessen konstruerar 5000 noder i ett syntaxträd vid varje test och testen körs 100 gånger är det möjligt att detta är en orsak till det sämre resultatet. Skräpinsamlingsproblem går vanligtvis att optimera genom att kontrollera användningen av de objekt som skapas under en exekvering.

8 DISKUSSION OCH SLUTSATSER

Slutresultatet av detta examensarbete är en prestandaoptimerad rekursivt nedstigande parser av Lua-språket. Implementationen av parsern är en pågående process som påbörjats 6 månader innan detta examensarbete fullgjorts.

Tidsmängden krävd för implementationen är omöjlig att mäta, men resultatet utgörs av mer än 1600 källkodsrader samt mer än 600 logiska programkodsrader. Ytterligare existerar det mer än 500 test som används för att upprätthålla parserns kvalitet. I efterhand kan jag konstatera att implementationen varit en okomplicerad men tidskrävande process. Eftersom tidsmängden för att generera en Lua-parser med hjälp av Jison enbart tog enstaka timmar är skillnaden mellan implementationsalternativen enorm. Slutprodukten visar sig dock vara en stabil och prestandaeffektiv implementation som kan användas i praktiken. Likt skaparna av GCC och Lua anser jag inte tidsmängden vara för mycket när det handlar om ett språk med en stabil och enkel syntax. Fördelen som fås visas tydligt i dess prestanda, dess felmeddelanden och enkelheten av källkoden.

För språk med en mer komplicerad syntax såsom C++ eller PHP anser jag dock inte det vara möjligt att upprätthålla en handskriven parser. Dessutom sker det kontinuerligt uppdateringar i syntaxen av ett språk, vilket vi kan se i senaste versionen av PHP samt i den uppkommande versionen av JavaScript. Processen att uppdatera syntaxgrammatiken för en genererad parser innebär enbart justeringar i en grammatikspecifikation. I en handskriven parser kan det innebära stora strukturella förändringar.

Slutsatsen jag har är att projekt som körs i en prestandakritisk miljö med ett tolkat språk bör överväga att implementera en handskriven parser. Om tidsmängden är begränsad eller projektet inte är prestandakritiskt kan dock en maskingenererad parser användas.

Det finns fortfarande en serie förbättringar som kan göras till denna parser för att vara mer användbar som ett analyseringsverktyg. I nästa version har jag planer att introducera inkrementell parsning så att syntaxmarkerare inte behöver parsa en hel fil för att göra uppdateringar. Ytterligare planerar jag anknyta källdkodspositionen till varje nod i syn-

taxträdet för att ett verktyg lättare skall kunna manipulera programkod.

Slutprodukten av arbetet finns i http://oxyc.github.io/luaparse/ och är licenseriad under MIT-licensen. Implementationen kan användas med Node.js eller en webbläsare med en JavaScript-motor som implementerats enligt ECMA-262 version 3.

KÄLLOR

About Lua. 2013, Lua.org PUC-Rio. Tillgänglig: http://www.lua.org/about.html, Hämtad: 10.2.2013.

Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi & Ullman, Jeffrey D. 2006, *Compilers: Principles, Techniques & Tools*, 2 uppl., Prentice Hall, 1000 s.

Biancuzzi, Federico & Warden, Shane. 2009, *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*, O'Reilly Media, 496 s.

Brown, Doug; Levine, John & Mason, Tony. 1992, *lex & yacc*, O'Reilly Media, 388 s.

Carter, Zach. 2010, *Jison Documentation*. Tillgänglig: http://zaach.github.com/jison/docs/, Hämtad: 24.2.2013.

charCodeAt vs charAt. 2011, jsPerf. Tillgänglig: http://jsperf.com/charcodeat-vs-string-comparison/2, Hämtad: 24.2.2013.

Grune, Dick & Jacobs, Ceriel J.H. 2008, *Parsing Techniques: A Practical Guide*, 2 uppl., New York: Springer, 662 s.

Ierusalimschy, Roberto. 2006, *Programming in Lua*, 2 uppl., Rio de Janeiro, Brazil: Lua.org, 328 s.

Ierusalimschy, Roberto; de Figueiredo, Luiz Henrique & Celes, Waldemar. 2011, *Lua 5.2 Reference Manual*. Tillgänglig: http://www.lua.org/manual/5.2/manual.html, Hämtad: 10.2.2013.

Ierusalimschy, Roberto; de Figueiredo, Luiz Henrique & Celes, Waldemar. 2013, *The implementation of Lua 5.0*. Tillgänglig: http://www.lua.org/doc/jucs05.pdf, Hämtad: 10.2.2013.

Lesk, Mike & Schmidt, Erich. 2013, *Lex - A Lexical Analyzer Generator*. Tillgänglig: http://dinosaur.compilertools.net/lex/, Hämtad: 21.2.2013.

lparse.c. 2011, Lua.org PUC-Rio. Tillgänglig: http://www.lua.org/source/5.2/lparser.c.html, Hämtad: 10.2.2013.

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.

Myers, Joseph S. 2004, *New C parser [patch]*. Tillgänglig: http://gcc.gnu.org/ml/gcc-patches/2004-10/msg01969.html, Hämtad: 21.2.2013.

Pall, Mike. 2010, *Re: Hyperblock Scheduling*. Tillgänglig: http://lambda-the-ultimate.org/node/3851#comment-57761, Hämtad: 25.2.2013.

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.

SpiderMonkey - Parser API. 2012, Mozilla Developer Network. Tillgänglig: https://developer.mozilla.org/en-US/docs/SpiderMonkey/Parser_API, Hämtad: 31.2.2013.

Thomas, Rudolf. 2005, *A Parser of the C++ Programming Language*, Examensarbete, Prague: Charles University, Faculty of Mathematics and Physics, s 44.

V8Profiler - How to profile V8 using the built-in profiler. 2012, V8 JavaScript Engine. Tillgänglig: http://code.google.com/p/v8/wiki/V8Profiler, Hämtad: 24.2.2013.

BILAGA 1. LUA-GRAMMATIK

⟨*chunk*⟩ ::= ⟨*block*⟩

⟨*block*⟩ ::= {⟨*stat*⟩} [⟨*retstat*⟩]

⟨*stat*⟩ ::= **;**
| ⟨*varlist*⟩ **=** ⟨*explist*⟩
| ⟨*functioncall*⟩
| ⟨*label*⟩
| **break**
| **goto** Name
| **do** ⟨*block*⟩ **end**
| **while** ⟨*exp*⟩ **do** ⟨*block*⟩ **end**
| **repeat** ⟨*block*⟩ **until** ⟨*exp*⟩
| **if** ⟨*exp*⟩ **then** ⟨*block*⟩ {**elseif** ⟨*exp*⟩ **then** ⟨*block*⟩} [**else** ⟨*block*⟩] **end**
| **for** Name **=** ⟨*exp*⟩ **,** ⟨*exp*⟩ [**,** ⟨*exp*⟩] **do** ⟨*block*⟩ **end**
| **for** ⟨*namelist*⟩ **in** ⟨*explist*⟩ **do** ⟨*block*⟩ **end**
| **function** funcname ⟨*funcbody*⟩
| **local** ⟨*function*⟩ Name ⟨*funcbody*⟩
| **local** ⟨*namelist*⟩ [**=** ⟨*explist*⟩]

⟨*retstat*⟩ ::= **return** [⟨*explist*⟩] [**;**]

⟨*label*⟩ ::= **::** Name **::**

⟨*funcname*⟩ ::= Name {**.** Name} [**:** Name]

⟨*varlist*⟩ ::= ⟨*var*⟩ {**,** ⟨*var*⟩}

⟨*var*⟩ ::= Name | ⟨*prefixexp*⟩ **[** ⟨*exp*⟩ **]** | ⟨*prefixexp*⟩ **.** Name

⟨*namelist*⟩ ::= Name {**,** Name}

⟨*explist*⟩ ::= ⟨*exp*⟩ {**,** ⟨*exp*⟩}

⟨*exp*⟩ ::= **nil** | **false** | **true** | Number | String | **...** | ⟨*functiondef*⟩ | ⟨*prefixexp*⟩ | ⟨*tableconstructor*⟩ | ⟨*exp*⟩ ⟨*binop*⟩ ⟨*exp*⟩ | ⟨*unop*⟩ ⟨*exp*⟩

⟨*prefixexp*⟩ ::= ⟨*var*⟩ | ⟨*functioncall*⟩ | **(** ⟨*exp*⟩ **)**

⟨*functioncall*⟩ ::= ⟨*prefixexp*⟩ ⟨*args*⟩ | ⟨*prefixexp*⟩ **:** Name ⟨*args*⟩

⟨*args*⟩ ::= **(** [⟨*explist*⟩] **)** | ⟨*tableconstructor*⟩ | String

⟨*functiondef*⟩ ::= **function** ⟨*funcbody*⟩

⟨*funcbody*⟩ ::= **(** [⟨*parlist*⟩] **)** ⟨*block*⟩ **end**

⟨*parlist*⟩ ::= ⟨*namelist*⟩ [**,** **...**] | **...**

⟨*tableconstructor*⟩ ::= **{** [⟨*fieldlist*⟩] **}**

⟨*fieldlist*⟩ ::= ⟨*field*⟩ {⟨*fieldsep*⟩ ⟨*field*⟩} [⟨*fieldsep*⟩]

⟨*field*⟩ ::= [⟨*exp*⟩] **=** ⟨*exp*⟩ | Name **=** ⟨*exp*⟩ | ⟨*exp*⟩

⟨*fieldsep*⟩ ::= **,** | **;**

⟨*binop*⟩ ::= **+** | **-** | ***** | **/** | **^** | **%** | **..** | **<** | **<=** | **>** | **>=** | **==** | **~=** | **and** | **or**

⟨*unop*⟩ ::= **-** | **not** | **#**

BILAGA 2. NODANTAL ANALYS AV LUAMINIFY

Provtagningen är baserad på LuaMinify-projektet och implementerad med Luaparse.

Nod-typ	Antal
Identifier	1771
Literal	705
MemberExpression	339
CallExpression	334
AssignmentStatement	215
ElseifClause	181
LocalStatement	174
TableValue	172
BinaryExpression	162
IfStatementh	138
ReturnStatement	130
UnaryExpression	122
Comment	122
TableConstructorExpression	101
IndexExpression	56
TableKeyString	49
LogicalExpression	44
ElseClause	39
FunctionDeclaration	38
CallStatement	26
WhileStatement24TableKey	20
BreakStatement	12
TableCallExpression	9
ForGenericStatement	6
ForNumericStatement	5
RepeatStatement	4
Chunk	1
Totalt	4999

BILAGA 3. LUA-GRAMMATIK FÖR JISON

```
/*
    Copyright 2011 Maximilian Herkender

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
*/
%lex

%%

\s+                     /* skip whitespace */
"--[["(.|\n|\r)*?"]]"   /* skip multiline comment */
"--".*                  /* skip comment */
"0x"[0-9a-fA-F]+        return 'NUMBER';
\d+(\.\d*)?([eE]"-"?\d+)? return 'NUMBER';
\.\d+([eE]"-"?\d+)?     return 'NUMBER';
 "\""("\\".|[^"])*"\""  return 'STRING';
 "'"("\\".|[^'])*"'"    return 'STRING';
 "[["(.|\n|\r)*?"]]"    yytext = longStringToString(yytext); return 'STRING';
":"                     return ':';
";"                     return ';';
"("                     return '(';
")"                     return ')';
"["                     return '[';
"]"                     return ']';
"{"                     return '{';
"}"                     return '}';
"+"                     return '+';
"-"                     return '-';
"*"                     return '*';
"/"                     return '/';
"%"                     return '%';
"^"                     return '^';
"=="                    return '==';
```

```
"="                     return '=';
"~="                    return '~=';
"<="                    return '<=';
">="                    return '>=';
"<"                     return '<';
">"                     return '>';
"#"                     return '#';
","                     return ',';
"..."                   return '...';
".."                    return '..';
"."                     return '.';
"not"                   return 'NOT';
"and"                   return 'AND';
"or"                    return 'OR';
"true"                  return 'TRUE';
"false"                 return 'FALSE';
"nil"                   return 'NIL';
"function"              return 'FUNCTION';
"until"                 return 'UNTIL';
"do"                    return 'DO';
"end"                   return 'END';
"while"                 return 'WHILE';
"if"                    return 'IF';
"then"                  return 'THEN';
"elseif"                return 'ELSEIF';
"else"                  return 'ELSE';
"for"                   return 'FOR';
"local"                 return 'LOCAL';
"repeat"                return 'REPEAT';
"in"                    return 'IN';
"return"                return 'RETURN';
"break"                 return 'BREAK';
[a-zA-Z_][a-zA-Z0-9_]*  return 'NAME';
<<EOF>>                 return 'EOF';

/lex

%token NUMBER STRING ":" ";" "(" ")" "[" "]" "{" "}" "." "+" "-" "*" "/" "%" "^" "=" "=="
    "~=" "<" "<=" ">" ">=" ".." "#" "," "..." NOT AND OR TRUE FALSE NIL FUNCTION
    UNTIL DO END WHILE IF THEN ELSEIF ELSE FOR LOCAL REPEAT IN RETURN
    BREAK NAME EOF

%left "OR"
%left "AND"
%left "<" "<=" ">" ">=" "==" "~="
```

```
%right ".."
%left "+" "-"
%left "*" "/" "%"
%right "NOT" "#"
%right "^"

%start chunk

%%

chunk: block EOF {};

semi: ";" {} | {};

block: statlist {} | statlist laststat {};

prefixexp: var {} | functioncall {} | "(" exp ")" {};

statlist : stat semi statlist {} | {};

stat
  : varlist "=" explist {}
  | LOCAL namelist "=" explist {}
  | LOCAL namelist {}
  | functioncall {}
  | DO block END {}
  | WHILE exp DO block END {}
  | REPEAT block UNTIL exp {}
  | IF conds END {}
  | FOR namelist_setlocals "=" exp "," exp DO block END {}
  | FOR namelist_setlocals "=" exp "," exp "," exp DO block END {}
  | FOR namelist_setlocals IN explist DO block END {}
  | FUNCTION funcname funcbody {}
  | FUNCTION funcname ":" NAME funcbody {}
  | LOCAL FUNCTION name_setlocals funcbody {}
  ;

laststat : RETURN explist {} | RETURN {} | BREAK {};

conds: condlist {} | condlist ELSE block {};

condlist: cond {} | condlist ELSEIF cond {};

cond: exp THEN block {};
```

varlist : *varlist* "," *var* {} | *var* {};

explist : *explist* "," *exp* {} | *exp* {};

namelist_setlocals: *namelist_setlocals* "," *NAME* {} | *NAME* {};

namelist: *namelist* "," *NAME* {} | *NAME* { };

name_setlocals: *NAME* {};

arglist : *arglist* "," *NAME* {} | *NAME* {};

funcname: *funcname* "." *NAME* {} | *NAME* {};

exp
: *NUMBER* { *$$ = $1*; }
| *STRING* { *$$ = $1*; }
| *TRUE* { *$$ = true*; }
| *FALSE* { *$$ = false*; }
| *tableconstructor* {}
| *NIL* { *$$ = null*; }
| *prefixexp* {}
| *FUNCTION funcbody* {}
| *exp* "+" *exp* {}
| *exp* "−" *exp* {}
| *exp* "∗" *exp* {}
| *exp* "/" *exp* {}
| *exp* "^" *exp* {}
| *exp* "%" *exp* {}
| *exp* ".." *exp* {}
| *exp* "<" *exp* {}
| *exp* ">" *exp* {}
| *exp* "<=" *exp* {}
| *exp* ">=" *exp* {}
| *exp* "==" *exp* {}
| *exp* "~=" *exp* {}
| *exp AND exp* {}
| *exp OR exp* {}
| "−" *exp* {}
| *NOT exp* {}
| "#" *exp* {}
| "..." {}
;

tableconstructor: "{" "}" { } | "{" *fieldlist* *fieldsepend* "}" {};

```
funcbody: "(" ")" block END { } | "(" arglist ")" block END {}
  | "(" "..." ")" block END {} | "(" arglist "," "..." ")" block END {};

var: NAME {} | prefixexp "[" exp "]" {} | prefixexp "." NAME {};

functioncall: prefixexp args {} | prefixexp ":" NAME args {};

args: "(" explist ")" {} | "(" ")" {} | tableconstructor {} | STRING {};

fieldlist : field {} | fieldlist fieldsep field {};

field: exp {} | NAME "=" exp {} | "[" exp "]" "=" exp {};

fieldsep: ";" {} | "," {};

fieldsepend: ";" {} | "," {} | {};

%%

module.exports = parser;
```

BILAGA 4. SKRIPT FÖR PRESTANDAANALYS I JAVASCRIPT

```
var fs = require('fs'),
  jison = require('./lua_parser'),
  luaparse = require('luaparse'),
  Benchmark = require('benchmark'),
  suite = new Benchmark.Suite,
  source = fs.readFileSync(process.argv[2], 'utf8'),
  isEOF = false,
  jisonLexer = jison.lexer,
  parser;

var luaparseLexer = {
  setInput: function(_input) {
    luaparse.parse(_input, { wait: true });
    luaparse.rewind();
    isEOF = false;
  },
  lex: function() {
    var token = luaparse.lex();

    switch (token.type) {
      case 4: // Keyword
        return token.value.toUpperCase();
      case 2: // StringLiteral
        return 'STRING';
      case 16: // NumericLiteral
        return 'NUMBER';
      case 128: // NilLiteral
        return 'NIL';
      case 1: // EOF
        // jison expects the input to stop after EOF.
        if (!isEOF) {
          isEOF = true;
          return 'EOF';
        }
        return '';
      case 8: // Identifier
        return 'NAME';
      case 64: // BooleanLiteral
        return token.value ? 'TRUE' : 'FALSE';
      default:
        return token.value;
    }
  }
```

```
};


suite
  .add('jison-lexer', function() {
    jison.parse(source);
  }, { onStart: function() { jison.lexer = jisonLexer; } })
  .add('luaparse-lexer', function() {
    jison.parse(source);
  }, { onStart: function() { jison.lexer = luaparseLexer; } })
  .add('luaparse', function() {
    luaparse.parse(source);
  })
  .on('cycle', function(e) {
    var target = e.target,
      variance = target.stats.variance * 1000 * 1000,
      stats = [
        target.stats.mean * 1000,
        Math.sqrt(variance),
        variance
      ];
    console.log(target.name + '\t' + stats.join('\t'));
  });

suite.run();
```

BILAGA 5. SKRIPT FÖR PRESTANDAANALYS I LUA

```
require './ParseLua'
require 'socket'
samples=100

function getMean(res)
  local total, count = 0, 0
  for key,value in pairs(res) do
    total = total + value
    count = count + 1
  end
  return total / count
end
function getVariance(res, mean)
  local total, count = 0, 0
  for key,value in pairs(res) do
    total = total + (value - mean)^2
    count = count + 1
  end
  return total / (count - 1)
end
function getDeviation(variance)
  return (math.sqrt(variance))
end

do
  local inf = io.open('ParseLua.lua', 'r');
  local text = inf:read('*all');
  inf:close();
  local results = {}
  for i=0,samples do
    local start = socket.gettime()
    ParseLua(text)
    local time = (( socket.gettime() - start )) * 1000
    results[i] = time
  end
  local mean = getMean(results)
  local variance = getVariance(results, mean)
  local sd = getDeviation(variance)
  print (mean..'\t'..sd..'\t'..variance)
end
```

BILAGA 6. “BAILOUT”-DATA

```
Bailout in HGraphBuilder: @"IN": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"ToString": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"toString": inlined runtime function: ClassOf
Bailout in HGraphBuilder: @"ToObject": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"NonNumberToNumber": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"SUB": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"ToNumber": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"DefaultNumber": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"isUnary": SwitchStatement: non-literal switch label
Bailout in HGraphBuilder: @"parsePrimaryExpression": SwitchStatement: non-literal switch label
Bailout in HGraphBuilder: @"forProps": ForInStatement is not fast case
Bailout in HGraphBuilder: @"FILTER_KEY": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"DELETE": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"extend": bad value context for arguments value
Bailout in HGraphBuilder: @"NonStringToString": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"extend": bad value context for arguments value
Bailout in HGraphBuilder: @"TO_NUMBER": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"MUL": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"sqrt": inlined runtime function: MathSqrt
Bailout in HGraphBuilder: @"DefaultString": call to a JavaScript runtime function
Bailout in HGraphBuilder: @"EQUALS": call to a JavaScript runtime function
```

BILAGA 7. PROVTAGNING 1

```
Statistical profiling result from v8.log, (5313 ticks, 0 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
      2    0.0%    0.0%  /usr/lib/libc-2.17.so

 [JavaScript]:
   ticks  total  nonlib   name
    749   14.1%   14.1%  Stub: RegExpExecStub
    703   13.2%   13.2%  LazyCompile: *test native regexp.js:210
    379    7.1%    7.1%  KeyedLoadIC: A keyed load IC from the snapshot
    340    6.4%    6.4%  LazyCompile: *readToken lib/luaparse.js:464
    254    4.8%    4.8%  Stub: CompareICStub {1}
    253    4.8%    4.8%  RegExp: ^[ \\t\\u000B\\u000C\\u0020]$
    238    4.5%    4.5%  RegExp: ^[\\n\\r]$
    233    4.4%    4.4%  Stub: StringAddStub
    216    4.1%    4.1%  LazyCompile: *scanIdentifierOrKeyword lib/luaparse.js:557
    177    3.3%    3.3%  RegExp: ^[a-zA-Z0-9_]$
    158    3.0%    3.0%  Stub: CompareICStub
    128    2.4%    2.4%  RegExp: ^[a-zA-Z_]$
     85    1.6%    1.6%  LazyCompile: *parsePrefixExpression lib/luaparse.js:1502
     70    1.3%    1.3%  LazyCompile: *parseSubExpression lib/luaparse.js:1460
```

BILAGA 8. PROVTAGNING 2

```
Statistical profiling result from v8.log, (2688 ticks, 0 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
      3    0.1%    0.0%  /usr/lib/libc-2.17.so

 [JavaScript]:
   ticks  total  nonlib   name
    338   12.6%   12.6%  LazyCompile: readToken lib/luaparse.js:452
    259    9.6%    9.6%  Stub: CompareICStub {1}
    242    9.0%    9.0%  LazyCompile: *scanIdentifierOrKeyword lib/luaparse.js:547
    212    7.9%    7.9%  Stub: StringAddStub
    121    4.5%    4.5%  LazyCompile: *skipWhiteSpace lib/luaparse.js:530
    119    4.4%    4.4%  KeyedLoadIC: A keyed load IC from the snapshot
     80    3.0%    3.0%  LazyCompile: *parseSubExpression lib/luaparse.js:1447
     65    2.4%    2.4%  LazyCompile: *parsePrefixExpression lib/luaparse.js:1489
     46    1.7%    1.7%  Stub: CompareICStub
     46    1.7%    1.7%  LazyCompile: *scanStringLiteral lib/luaparse.js:597
     42    1.6%    1.6%  LazyCompile: *scanPunctuator lib/luaparse.js:579
     41    1.5%    1.5%  Stub: CallFunctionStub
     41    1.5%    1.5%  LazyCompile: *parsePrimaryExpression lib/luaparse.js:1579
     38    1.4%    1.4%  LazyCompile: *parseStatement lib/luaparse.js:1049

 ...

 [Bottom up (heavy) profile]:
  Note: percentage shows a share of a particular caller in the total
  amount of its parent calls.
  Callers occupying less than 2.0% are not shown.

  ...

    259    9.6%  Stub: CompareICStub {1}
    105   40.5%    LazyCompile: *parseSubExpression lib/luaparse.js:1447
```