Implementering av en Lua parser

Oskar Schöldström

Examensarbete
Informationsteknik
2013

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informations- och medieteknik
Identifikationsnummer:	1234
Författare:	Oskar Schöldström
Arbetets namn:	Implementering av en Lua parser
Handledare (Arcada):	...
Uppdragsgivare:	

Sammandrag:
a a

Nyckelord:	parser, javascript, lua, lexer, tolk
Sidantal:	35
Språk:	Svenska
Datum för godkännande:	

DEGREE THESIS	
Arcada	
Degree Programme:	Information and Media Technology
Identification number:	1234
Author:	Oskar Schöldström
Title:	Implementing a Lua parser
Supervisor (Arcada):	...
Commissioned by:	

Abstract: a a

Keywords:	parser, javascript, lua, lexer, lexical analysis, tokenizer, scanner, lexical analyzer, syntactic analysis
Number of pages:	35
Language:	Swedish
Date of acceptance:	

OPINNÄYTE	
Arcada	
Koulutusohjelma:	Informaatio- ja mediatekniikka
Tunnistenumero:	1234
Tekijä:	Oskar Schöldström
Työn nimi:	Implementing a Lua parser
Työn ohjaaja (Arcada):	...
Toimeksiantaja:	

Tiivistelmä:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas molestie, nisi sed consectetur tincidunt, ante libero euismod nulla, eget commodo ligula purus eu nibh. Ut enim risus, congue sed viverra ut, tempor vel velit. Quisque consectetur tincidunt scelerisque. Donec nec mollis leo. Donec laoreet purus a massa ultrices egestas. Ut quis neque nisi. In hac habitasse platea dictumst. Suspendisse suscipit, ante eget venenatis molestie, elit nulla aliquam ligula, sit amet consectetur tortor magna eu ipsum. Pellentesque tempor tincidunt arcu, eget ultrices lorem commodo a.

Avainsanat:	parser, javascript, lua, lexer, lexical analysis, tokenizer, scanner, lexical analyzer, syntactic analysis
Sivumäärä:	35
Kieli:	Ruotsi
Hyväksymispäivämäärä:	

INNEHÅLL

TABELLER

FIGURER

1 INLEDNING

1.1 Syfte och målsättning

1.2 Utförande

2 SPRÅK TEORI

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

2.1 Backus-Naur Form

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

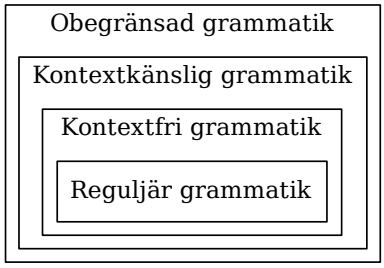
När man beskriver syntaxen av ett språk använder man sig av ett metaspråk för att definiera de syntaktiska regler man är tillåten att använda. Det finns ett flertal metaspråk men ett av de vanligaste inom programmeringsspråk är Backus-Naur Form (BNF) (Grune & Jacobs 2008 s. 27). Notationen är uppbyggd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

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

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

2.2 Chomskyhierarkin

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



Figur 1. De fyra nivåerna i Chomskyhierarkin

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

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

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

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

2.2.1 Reguljär grammatik

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

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

$\langle nummer \rangle$ ::= $\langle heltal \rangle$ | $\langle reellt\ tal \rangle$

$\langle heltal \rangle$ ::= $\langle siffra \rangle$ $\{\langle siffra \rangle\}$

$\langle reellt\ tal \rangle$ ::= $\langle heltal \rangle$ $\langle exponent \rangle$ | $\langle decimaltal \rangle$ $[\langle exponent \rangle]$

$\langle decimaltal \rangle$ ::= $\langle siffra \rangle$ (**.** $\langle siffra \rangle$ | $\langle siffra \rangle$ **.**) $\langle siffra \rangle$

$\langle exponent \rangle$ ::= (**e** | **E**) [+ | **-**] $\langle heltal \rangle$

$\langle siffra \rangle$ ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Figur 2. EBNF grammatik för ett nummer.

2.2.2 Kontextfri grammatik

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

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

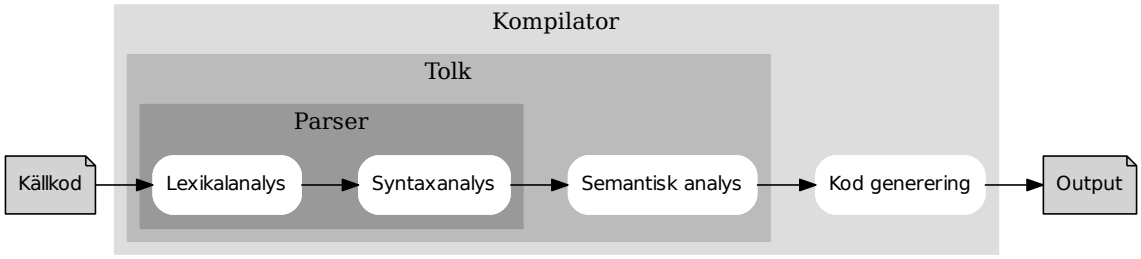
$\langle uttryck \rangle$ ::= $\langle nummer \rangle$ | (**-** | +) $\langle uttryck \rangle$ | **(** $\langle uttryck \rangle$ **)**
| $\langle uttryck \rangle$ $\langle operator \rangle$ $\langle uttryck \rangle$

$\langle operator \rangle$ ::= + | - | * | /

Figur 3. BNF grammatik för en kalkylator.

3 PARSNING

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



Figur 4. Översikt av komponenterna i en kompilator.

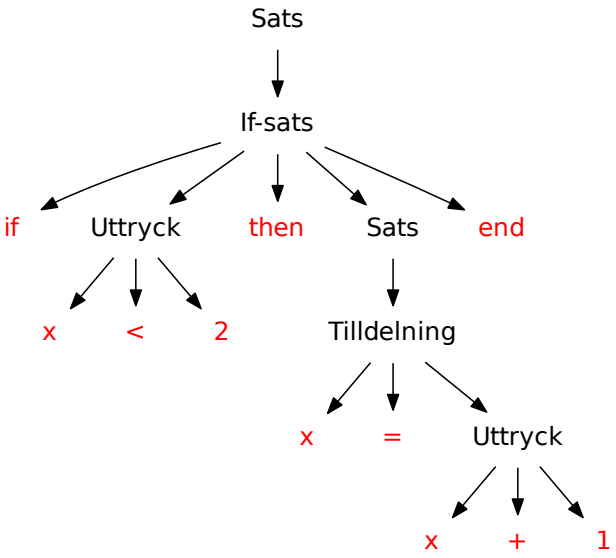
Parsnings processen kan delas upp i två skilda faser, först en s.k. lexikal analys som identifierar lexikala element, mer kända som tokens. Tokens är identifierbara teckensträngar med speciell betydelse. De kategoriseras enligt typer så som nyckelord, konstanter, parametrar osv. (Aho et al. 2006 s. 6).

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

3.1 Lexikal analys

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

3.2 Syntastisk analys



Figur 5. Den syntastiska uppdelningen av en kod-sats

3.3 Parser typer

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

När man pratar om parsers nämner man ofta hur långt fram den kan se innan den gör ett beslut, dvs. hur många tokens framåt den kan se. Detta antal skriver man inom en parentes efter parsertypen, dvs. för att uttrycka den vanligaste formen av parser som är en LR parser som ser en token framåt skriver man LR(1) (Scott 2009 s. 69).

3.3.1 LL parser

LL parsers går att skriva förhand eftersom de i allmänhet följer en logisk tankegång. De börjar med att gissa sig till en rot-regel, likt en ändlig automat, och förväntar sig sedan att löv-reglerna skall passa in en efter en. Om det misslyckas kan den gå till nästa alternativa regel och fortsätter fram tills den bevisat en gissning. Denna typ av strategi kallas även för en uppifrån-och-ner strategi (Scott 2009 s. 67).

3.3.2 LR parser

LR parsers genereras huvudsakligen av maskiner eftersom de är svårare att visualisera. Dessa parsers börjar med att läsa löv-reglerna, alltså de minsta identifierbara reglerna och ansluter dem sedan till varandra fram till det att den nått en slutgiltig rot-regel. Detta kallas även för en nerifrån-och-upp strategi (Scott 2009 s. 67).

Både LL parsers och LR parsers används i kompilatorer men LR är mer vanligt (Scott 2009 s. 67). En orsak är att LR parsers täcker en större andel av grammatiker och därmed använder ofta programvara som genererar parsers sig ofta av av nerifrån-och-upp strategin (Aho et al. 2006 s. 61).

3.3.3 Rekursivt nedstigande parser

En form av LL parsers som är vanlig för handskrivna parsers är den rekurisvt nedstigande parsern. Den fungerar genom att knyta varje icke-terminal i grammatik till en egen funktion som ansvarar för att identifiera dess grammatiska löv. För varje icke-terminal i sin egen regel kallar den rekursivt vidare på dess anknytande funktioner fram till det att rot funktionen får hela syntax trädet returnerat (Parr 2010 s. 24).

@TODO epsilon

@TODO

3.4 Tekniker

3.4.1 Vänster faktorering

Hur gör man left factoring @TODO detta är snabbt flyttat från en annan sektion hit och kommer beskriva processen för left factoring.

$\langle exp \rangle \quad ::= \ldots | \ \langle exp \rangle \ \langle binop \rangle \ \langle exp \rangle \mid \ldots$

Om denna regel skulle implementeras skulle den se ut enligt figur **??** och vi kan se hur funktionen alla skulle komma vidare.

```
function exp() {
  exp(); binop(); exp();

}
```

Figur 6. Direkt implementation av binära operationer enligt Lua grammatiken.

3.4.2 Backtracking

I vissa fall är det användbart att inte ha en begränsad framåtblick och då kan man använda sig av en teknik som heter backtracking som tillåter parsern att se en obegränsad mängd tokens framåt. Implementationen av en backtracker varierar men i simpel form fungerar det genom att när mångtydlighet uppstår välja ett alternativ och sedan ha möjligheten att återgå till ett tidigare tillstånd när alternativet bevisats rätt eller fel (Parr 2010 s. 55).

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

3.4.3 Memoisation

Kort om vad memoisation är.

3.5 Parser genererare

Vad är parser genererare

3.5.1 Jison

Exempel på parser genererare, samt projekt som använder det.

https://github.com/josevalim/luascript/blob/master/lib/lua/compiler/parser.jison

3.5.2 Fördelar och nackdelar

Diskutera vad som allmänt ses som fördelar och nackdelar. Citera skaparna av lua om varför de böt från yac i 3.0.

4 IMPLEMENTATION

Implementation är uppdelad i fyra komponenter. När parsern startas börjar en syntaktisk analysator parsa input från rot-noden, dvs. från program-noden. Det första som görs är en kallelse till den andra komponenten, lexern. Lexern läser input tecken för tecken och tokeniserar värden som sedan returneras till den syntaktiska analysatorn. Utanför lexern hanteras inte enskilda tecken mer utan enbart tokens. Utöver dessa två komponenter finns en separat uttrycksparser samt ett delegerare som producerar ett syntax träd vilket slutligen returneras till programmet som kallat på parsern i första hand.

4.1 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 går sedan genom en serie villkor för att bestämma vilken var token slutar och av vilken typ den är. När en token tar slut har automaten nått sitt slutliga tillstånd och informationen returneras varefter automaten återgår till startläget för att vänta på nästa förfrågan.

4.1.1 Identifierare och nyckelord

Beskriv identifier start, identifier part, prestanda. Notering om varför variabler inte börjar med siffror.

4.1.2 Kommentarer och blanksteg

Hur kommentarer och blanksteg ignoreras. Kommenterar lagras avskiljt.

4.1.3 Litteraler och symboler

Beskriv string vs long string. Hex och exponenter. Dec och exponenter med JS native funktioner, escape sequences.

4.2 Syntaktisk analysator

Den syntaktiska analysatorn är en rekursivt nedstigande parser. Den börjar från rot-noden, i Lua länd som *chunk*. Denna nod består av ett block som i sin tur består av en obestämnd mängd satser. Varje nodtyp, eller enligt grammatiken regel är implementerad som en egen funktion som rekursivt kallar på andra regel funktioner. När input strängen tagit slut har parsern också returnerat ett träd av information till rot-noden som sedan returneras till programmet som begärt informationen.

Parsern är implementerad som en LL(1) parser med minne för en aktuell token, samt en lookahead token. Med hjälp av dessa två tokens kan hela Lua grammatiken syntaktiskt analyseras.

Mycket av den syntaktiska analysatorns grammatik är samma som den ursprungliga Lua grammatiken, dock har vissa finjusteringar gjorts för att bland annat kunna identifiera variabel deklarationer från funktionskallelser. Dessa två sats-typer skulle normalt kräva fler lookahead tokens för att kunna identifieras korrekt. Implementationen har istället implementerat funktionalitet för memoisation. Detta innebär att det första alternativet testas och om det misslyckas används istället det andra alternativet.

4.2.1 EBNF notation till JavaScript

Hur översätts notationen, exempel på if / while.

4.2.2 Satser

<stat> implementationen.

4.3 UttryckSparser

Lua grammatikens uttrycksregel, *expr*, utnyttjar vänster rekursion, som inte är möjligt i rekursivt nedstigande parsers eftersom de hamnar i en oändlig loop. Detta ser vi i alternativet var ett uttryck kan vara ett uttryck kombinerat med en binär operator samt ett annat uttryck.

$\langle exp \rangle \quad ::= \ldots | \ \langle exp \rangle \ \langle binop \rangle \ \langle exp \rangle \mid \ldots$

Eftersom uttrycks regeln även består av en serie andra icke-terminaler vars funktioner är svåra att få ett grepp om kommer vi att expandera och analysera alla delregler för att slutligen få en fungerande och hoppeligen mer organiserad regel för uttryck.

Processen för att införa detta kommer bestå av tre steg, var det det första visar den ursprungliga regeln, den andra visar resultatet från vänster faktoreringen och den tredje visar regeln skriven utan epsilon.

4.3.1 Gruppering av uttryck

För att lättare arbeta med definitionen grupperar vi in uttryck i binära och unära operatioer, prefix uttryck samt primära uttryck. Primära uttryck består av de kvarstående alternaten, huvudsakligen datatyper.

Ursprungsregel

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

$\langle primaryexp \rangle ::= \ldots$

Eliminering av vänster rekursion

$\langle exp \rangle$ ::= $\langle primaryexp \rangle$ $\langle exp' \rangle$ | $\langle prefixexp \rangle$ $\langle exp' \rangle$ | $\langle binop \rangle$ $\langle exp \rangle$ | $\langle unop \rangle$ $\langle exp \rangle$

$\langle exp' \rangle$ ::= $\langle binop \rangle$ $\langle exp \rangle$ | ε

Resultat

$\langle exp \rangle$ ::= ($\langle unop \rangle$ $\langle exp \rangle$ | $\langle primaryexp \rangle$ | $\langle prefixexp \rangle$) { $\langle binop \rangle$ $\langle exp \rangle$ }

4.3.2 Analys av prefix uttryck

Ursprungsregel

$\langle prefixexp \rangle$ ::= $\langle var \rangle$ | $\langle functioncall \rangle$ | ($\langle exp \rangle$)

$\langle var \rangle$ ::= Name | $\langle prefixexp \rangle$ [$\langle exp \rangle$] | $\langle prefixexp \rangle$. Name

$\langle functoincall \rangle$ ::= $\langle prefixexp \rangle$ $\langle args \rangle$ | $\langle prefixexp \rangle$: Name $\langle args \rangle$

$\langle args \rangle$ ::= ([$\langle explist \rangle$]) | $\langle tableconstructor \rangle$ | String

Den fullständiga regeln för argument kan hittas i bilaga 1 och vi kan konstatera att regeln inte behöver bearbetas mer. Vi expanderar ytterligare underreglerna för att få en bättre överblick.

$\langle prefixexp \rangle$ ::= Name | $\langle prefixexp \rangle$ [$\langle exp \rangle$] | $\langle prefixexp \rangle$. Name
| $\langle prefixexp \rangle$ $\langle args \rangle$ | $\langle prefixexp \rangle$: Name $\langle args \rangle$
| ($\langle exp \rangle$)

Eliminering av vänster rekursion

$\langle prefixexp \rangle$ ::= Name $\langle prefixexp' \rangle$ | ($\langle exp \rangle$) $\langle prefixexp' \rangle$

$\langle prefixexp' \rangle$ ::= [$\langle exp \rangle$] | . Name $\langle args \rangle$ | : Name $\langle args \rangle$ | ε

Resultat

$\langle prefixexp \rangle$ ::= (Name | ($\langle exp \rangle$)) { [exp] | . Name | : Name $\langle args \rangle$ | $\langle args \rangle$ }

4.3.3 Sammanknytning

Med hjälp av dessa modifieringar ha vi nu en uttrycks regel som inte använder sig av vänster rekursion 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*⟩ }

4.3.4 Binärt företrädande

I Lua manuelen (Ierusalimschy et al.) nämns det att grammatiken inte beskriver binärt företrädande, dvs. logiken som bestämmer att multiplikation skall utföras innan addition. Denna prioritering kan vi dock hitta i källkoden av lparse.c.

En implementationsmöjlighet som Lua själv använder är att spjälka ut funktionen som parsar uttryck till en ny funktionen som tar emot vänstra operandens prioritet som argument. Funktionen itererar sedan över de binära operationerna som följer, och lägger rekursivt till dom som binära operationer i den vänstra operanden fram till det att en operator har lägre prioritet eller inte är en operator. När detta sker har hela binära uttrycket hittats och strukturerats i ett träd enligt prioritet. Denna algoritm kallas även för operator-precedence parsing och är en form av nerifrån-och-upp parsning. Pseudokoden för detta visas i figur **??**.

4.3.5 Höger associativa regler

Associativa regler innebär hur uttryck grupperas när parenteser inte existerar. Ett exempel på detta är upphöjt till var grupperingen spelar stor betydelse. Om uttrycket *5^4^3^2* läses från vänster likt addition skulle gruppering bli *((5^4)^3)^2*, men eftersom upphöjt till har en höger associativ regel skall grupperingen egentligen vara *5^(4^(3^(2)))*, vilket ger ett annat resultat. Alla operatorer förutom ^ och .. är vänster associativa i Lua. Implementationen för detta hittas i figur **??** och utgörs av att subtrahera 1 från opretor prioriteten

```
function parseExpression() {
  return parseSubExpression(0);
}

function parseSubExpression(minPrecedence) {
  var expression;
  if (isUnary(token)) expression = unaryExpression(token, parseUnaryExpression());
  else if (isPimary(token)) expression = parsePrimaryExpression();
  else expression = parsePrefixExpression();

  while (true) {
    var operator = token
      , precedence = getPrecedence(operator);
    if (precedence <= minPrecedence) break;
    var right = parseSubExpression(precedence);
    expression = binaryExpression(expression, right);
  }
  return expression;
}
```

Figur 7. operator-precedence parser

om en höger associativ regel hitta innan den högra operanden parsas. Detta orsakar en omvändning i grupperingen.

4.3.6 Resultat

Resultatet av denna omvandling visar sig vara en nära exakt kopia av Luas egna uttrycks parser med vissa skillnader för operator associationer. Det kan konstateras att skaparna användt sig av en enkel vänster faktorering för att omvandla sin tidigare yacc skapade parser till en handskriven rekurisvt nedstigande parser.

4.4 Syntax träd

Vad är AST?

```
while (true) {
  var operator = token
    , precedence = getPrecedence(operator);
  if (precedence <= minPrecedence) break;
  if (parseRightAssociative(operator)) precedence--;
  var right = parseSubExpression(precedence);
  expression = binaryExpression(expression, right);
}
```

Figur 8. Implementation för höger associativa regler

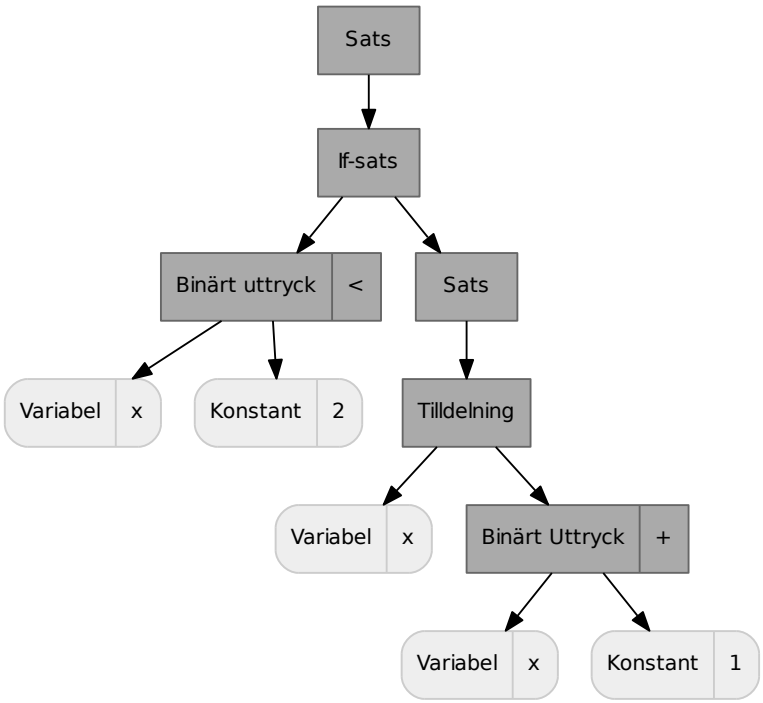
4.4.1 Representation

Beskriv Mozilla Parser API.

Vilken representation som används

4.4.2 Delegerare

Varför delegerare.



Figur 9. Abstrakt syntax träd som representation av en parsnings analys.

5 PRESTANDAOPTIMERING

Beskriv varför JavaScript behöver micro optimering.

5.1 Insamling av data

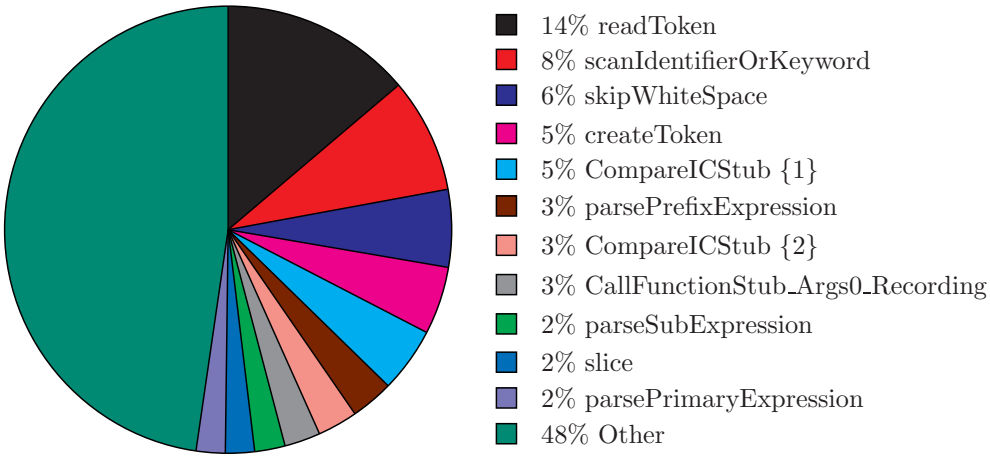
Beskriv processen för test samt sample standard deviation.

5.2 Analysering av data för V8

Beskriv varför v8.

5.2.1 Sampling

Beskriv samplings verktyget och hur datat läses.



Figur 10. Översikt av de mest prestanda krävande funktionera

5.3 Optimerings tekniker för V8

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

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

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

5.3.1 Undvikandet av bailouts

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

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

Funktionerna som orsakat dessa bailouts är *isUnary()* samt *parsePrimaryExpression()* som bägge består av en switch-sats som utför olika operationer beroende på om en given token är lika med ett visst variabelvärde. Detta är ett enkelt problem att lösa eftersom bailouten inte sker för if-sats motsvarigheten och därför kan koden optimeras enligt figur 11.

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

a) Funktion före

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

b) Funktion efter

Figur 11. Förändring av funktionen isUnary från icke-optimerbar till optimerbar.

5.3.2 Inlining

@TODO expandera kanske? Hur översätta literaler?

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

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

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

5.3.3 Teckenkoder istället för reguljära uttryck

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

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

5.3.4 Optimering av sträng funktioner

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

StringAddStub orsakas av att lexern sammanfoga token värden för kommentarer, nummer och identifierare genom att iterera över varje tecken så länge som en avgränsare inte hittas. Denna procedur går att försimpla genom att istället räkna var värdet börjar och sedan iterera fram till avgränsaren och vid det skedet skära ut värdet från dess start position till positionen var avgränsaren enligt exempel figur 12.

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

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

a) Procedur före

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

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

b) Procedur efter

Figur 12. Procedur förändring från sträng sammanfogning till sträng utskärning.

der i formen av ett träd enligt figur 13. Efter att optimeringen gjorts validerade jag ännu funktionen var tillräkligt liten för att bli inlined, vilket den visade sig vara.

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

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

Figur 13. Optimerad träd implementation av en vanlig switch-sats för strängar.

5.3.5 Garbage Collection

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

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

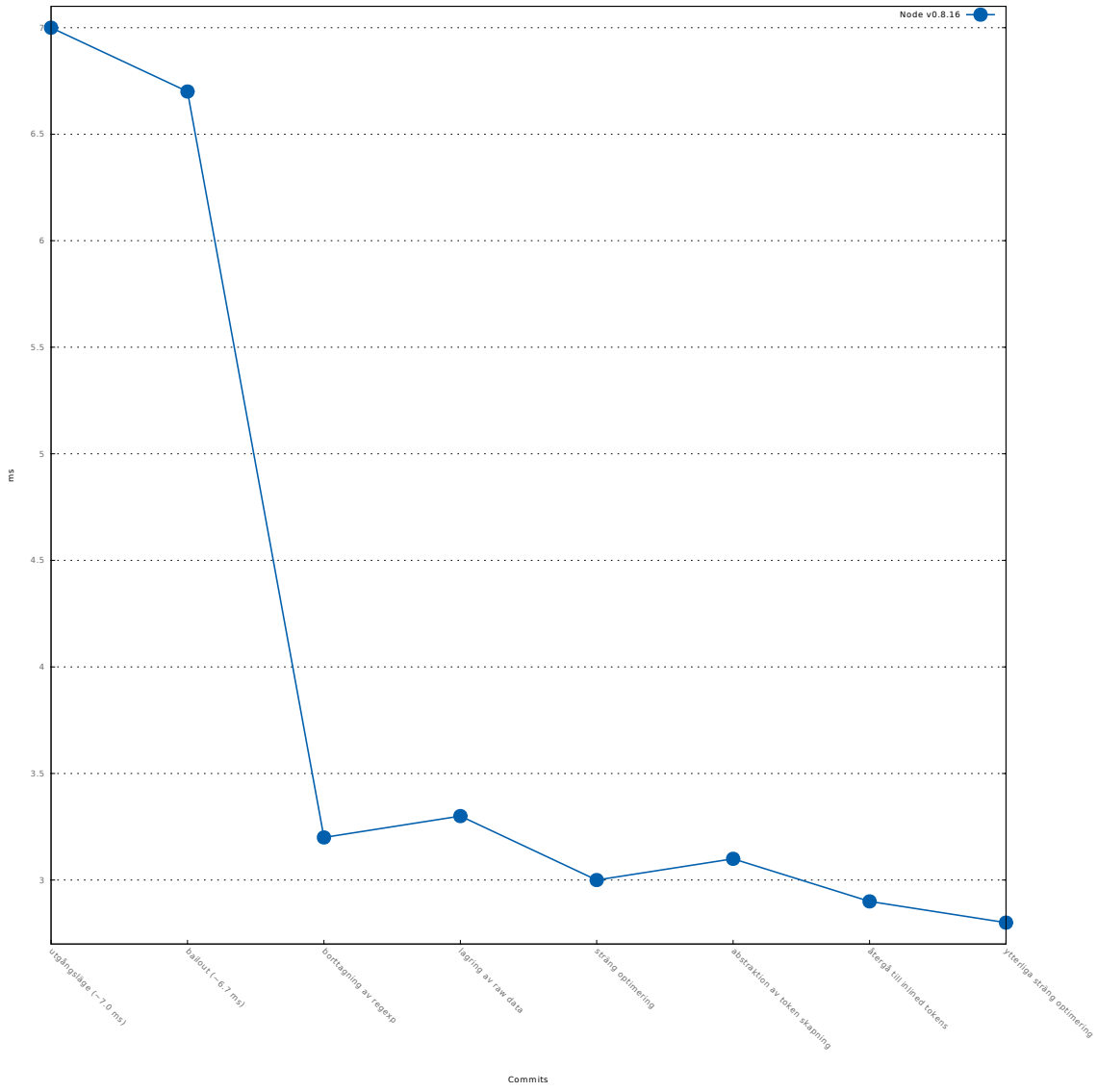
Eftersom parsern implementerats med hjälp av scopes och funktioner istället för med ett objektorienterat paradigm är all allokering indirekt och därmed finns det ingen oro om att minne skall läcka så länge som mer avancerade closure användningar inte används. När en funktion körts färdigt blir alla variabler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

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

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

5.4 Resultat

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



Figur 14. Översikt av de mest prestanda påverkande kod commitsen

6 DISKUSSION

...

KÄLLOR

Ager, Mads. 2010, *The V8 JavaScript Engine: Design, Implementation, Testing and Benchmarking*, doktorsavhandling, Odense: University of Southern Denmark. Tillgänglig: http://websrv0a.sdu.dk/ups/SCM/slides/lecture_03_mads_ager.pdf, Hämtad: 8.1.2013.

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

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

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

lparse.c. 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.

Parr, Terence. 2010, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, Pragmatic Bookshelf, 374 s.

Scott, Michael L. 2009, *Programing Language Pragmatics*, 3 uppl., Burlington: Morgan Kauffmann Publishers, 910 s.

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

BILAGA 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** | **#**