DESIGN & IMPLEMENTATION AV ETT PROGRAMMERINGSSPRÅK

Institut Kungliga Tekniska Högskolan
Handledare Mads Dam

Allan Jerjas 871026-2232
jerjas@kth.se
Adam Sam 890323-7470
adamsam@kth.se

Abstract

A simple programming language

This paper describes the design and implementation of a simple and arbitrary intuitive programming language, with an interpreter to be used by programmers new to the programming domain. This report describes the design and implementation of a programming language that takes much inspiration from the programming languages C and BASIC. The language is created using standard methodology to define a language and then to implement it with a parser-generator using the meta-language C. Finally some reflections on the method of implementation and the possibilities they entail are presented.

Sammanfattning

Ett enkelt programmeringsspråk

Dokumentet beskriver designen och implementationen av ett enkelt och godtyckligt intuitivt programmeringsspråk, med en tolk för att användas av personer nya till programmering. Denna rapport beskriver utformningen och genomförandet av ett programmeringsspråk som tar mycket inspiration från programmeringsspråken C och BASIC. Språket skapas med konventionella metoder för att definiera ett språk och sedan genomföra det med en parser-generator med hjälp av meta-språket C. Slutligen dras några reflektioner över metoden av genomförandet och de möjligheter som det för.

Innehållsförteckning

1 Inledning

1.1 Inledning

Definitionen av ett programmeringsspråk är inte helt tydlig. Oftast beskrivs ett programmeringsspråk som ett artificiellt utformat språk, för att utföra beräkningar, till exempel av en maskin. 1936 presenterade matematikern Alan Turing ett tankeexperiment med en teoretisk maskin, numera kallad “Turing-machine” som med en lista av instruktioner och en kodremsa kan hoppa fram och tillbaka mellan olika *tillstånd* och därmed simulera logiken hos alla möjliga datoralgoritmer. Om och endast om ett språk är kapabel till detta, så kallas språket “Turingkomplett”[1] eller ”Turingfullständigt”, detta kan till exempel åstadkommas med en implementering av rekursiva funktioner.
Vi har valt att utveckla ett programmeringsspråk där man precist kan uttrycka och exekvera simpla algoritmer och instruktioner samt kunna manipulera den maskin koden körs på, till exempel att skriva ut data på en terminal.
Det finns grovt nästan 2500 dokumenterade programmeringsspråk för att utveckla program på en dator,[2] många med varierande paradigm och utbredning, vissa även skapade som esoteriska projekt, så som det minimalistiska språket Brainfuck[3]. Många av dessa programmeringsspråk är s.k Script-språk som körs av en tolk. Tolken översätter och exekverar koden utan eller med delvis kompilering. Rapporten redogör för implementationen av ett enkelt programmeringsspråk, där fokus är på själva implementationen och inte på språket eller dess antydda enkelhet. Sammanfattningsvis är rapporten en beskrivning och undersökning av en metod till att skapa ett nytt programmeringsspråk

---

[1] 'On Computable Numbers, with an Application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society* (Series 2, volume 42 (1936-37), pp. 230-265)
[2]Lévénez, Éric, *The History of Programming Languages*, 2011-04-01, http://oreilly.com/news/graphics/prog_lang_poster.pdf
[3]Raiter, Brian, *Brainfuck,* 2011-03-27, http://www.muppetlabs.com/~breadbox/bf/

2 Introduktion

2.1 Syfte

Detta dokument skrivs som en del i kursen DD143X samt som ett kandidat-examensarbete inom datalogi, grundnivå vid KTH i Stockholm.

2.2 Lista av definitioner

BNF - Backus Naur Form, ett metaspråk för att definiera strukturen hos ett programmeringsspråk
C - programmeringsspråket C
Syntax - reglerna som definierar kombinationen av symboler för ett korrekt strukturerat språk[4]
Semantik - Meningen eller innebörden i ett uttryck, ett teckensystems tolkning.
Regex/Reguljär uttryck - Ett metaspråk för att matcha text strängar
CWELL - Namnet vi givit det språk vi konstruerar i denna rapport
CFL - Context-free language, Kontext-fritt språk
LL parser - Left to right, Rightmost derivation
LR parser - Left to right, Leftmost derivation
struct - data struktur för att lagra mer än ett värde
PC - Personal computer
PIC – Peripheral Interface Controller

2.3 Design

I detta stycke presenteras de design-val som gjorts för språket och hur språket ska se ut.

2.3.1 Infallsvinkel

Grunden för detta arbete bygger på att definiera ett enkelt programmeringsspråk i syfte att undersöka implementeringen av ett sådant språk. En redogörelse av

[4]Friedman, Daniel P; Mitchell Wand, Christopher T. Haynes (1992), *Essentials of Programming Languages* (1st ed.), The MIT Press, ISBN 0-262-06145-7.

konstruktionen och hur en tolk för ett programmeringsspråk kan se ut presenteras samt en reflektion I slutet. Definitionen av detta språk sker med BNF vilket kommer att beskriva syntaxen av språket. Definitionen kommer att vara utformad så att språket påminner om ett proceduralt imperativt språk. För att gå från definition till ett färdigt språk ska man verkställa en tolk för att översätta från programmerarens kod till exekverbar kod för att sedan köras på en persondator. Tolken för detta språk kommer att utformas med C för att bland annat undvika beroende av stora bibliotek vid exekvering. Med andra ord är C metaspråket som används för att implementera tolken. Valet av tolk över kompilator, som omvandlar kod till hårdvarunära eller binär form, har gjorts med tanke på enkelhet för utvecklaren, Så att en utvecklare inte behöver tänka på det plattform språket ska exekveras på, det gör det också enklare för oss att utvidga språket till andra operativsystem än till exempel windows. Infallsvinkeln här har alltså varit att utforma en tolk för ett språk vi definerar, ingen kompilator eller kompilator relaterad implementation har undersökts.

2.3.2 Imperativ

Imperativ programmering är ett programmeringsparadigm där programmen skrivs i form av sekvenser samt där strukturen hos programmen byggs med hjälp av procedurer[5]. Det är denna paradigm språket kommer vara baserat på.

2.3.3 Språkets utseende

Tre ogrundade godtyckligt valda egenskaper hos språket har gjorts, dessa har valts för att språket ska vara annorlunda från etablerade språk så att tolken inte blir en tolk för ett existerande språk. Detta bidrar också till att begränsa arbetet till att bara implementera dom nämnda egenskaperna som utgör själva ”utseendet” hos språket samt dess funktioner.

[5]Tom Smedsaas; Tobias Wrigstad, 2011-03-15, *Imperative and objekt-oriented programming, page 1,*
http://www.it.uu.se/edu/course/homepage/imperoop/ht08/forelasningar/del13/del13.pdf

2.3.3.1 Utskrift till terminal

För att ge språket en möjlighet att skriva ut värden används en print-metod lik den som finns i C, nedanstående exempel illustrerar hur ett sådant anrop kan se ut. Värdet hos x skrivs ut i ett terminalfönster.

```
print(x)          // Lik ”printf” i C
```

2.3.3.2 Överlagring

Ett återkommande problem för många programmerare och programmeringsspråk är hantering av typer. För att begränsa denna del av språket och inte göra projektet allt för stort, sker tilldelning och manipulering av variabler i språket med hjälp av överlagring. Språket överlagrar, dvs omvandla typer då olika typer beblandas i källkoden. Till exempel ska hanteringen av ett heltal som sedan konkateneras med en textsträng göras automatiskt av tolken. Se nedanstående exempel.

```
x = 1
x=x+1          // x har nu värdet 2
x=x+”hej”      // x har nu värdet “2hej”
x=x+1          // x har nu värdet “2hej1”
```

2.3.3.2 Valfri End of a Statement

Många imperativa språk kan uttrycka flera instruktioner/påstående på en och samma rad, detta möjliggörs med en “End of Statement”-tecken, oftast ett semikolon. Man kan också skriva en instruktion över flera rader och sen avsluta med ett semikolon. I C är detta semikolon obligatorisk även vid rader som endast innehåller en instruktion. I detta språk är semikolonet ett retur-tecken, så att ny rad automatiskt innebär ny instruktion, medan det mer ovanliga “en instruktion över flera rader” får ett eget tecken. Detta finns utformat så i Programmeringsspråket BASIC(dialekt: Visual Basic). Se nedanstående exempel över vår design.

```
x = 1                  // En instruktion på en rad
x=x_                   // En instruktion över två rader
+1
x=x+”hej”;x=x+1        // Två instruktioner över en rad
```

3 Bakgrund

3.1 Existerande lösningar

Det finns många programmeringsparadigm för att lösa olika delar av programmeringsvärlden. Dessa har olika syften och används på olika sätt. Nedan är en kort lista över fyra paradigm och exempel på språk oftast klassade under respektive paradigm.

- **Procedural**
  - Basic
  - C
- **Object-Oriented**
  - C#
  - Java
- **Declarative**
  - Prolog
  - SQL
- **Functional**
  - Lisp
  - Haskell

3.2 Influencer

Det programmeringsspråk som är tänkt att implementeras är starkt influerat av två andra programmeringsspråk, båda procedurala, nämligen Visual Basic och C. Nedan ges en kort beskrivning av dessa två språk.

3.2.1 C

C är ett av världens mest populära språk genom tiderna[6]. C-kod kompileras till en arkitektur och har kompilatorer för de flesta arkitekturerna på marknaden. C är ett proceduralt språk som utvecklades för att vara minimalistisk, hårdvarunära och enkel att kompilera. Det har lett till att C nu finns till många olika plattformar och arkitekturer så som inbyggda system och PC'n.

3.2.2 Visual Basic

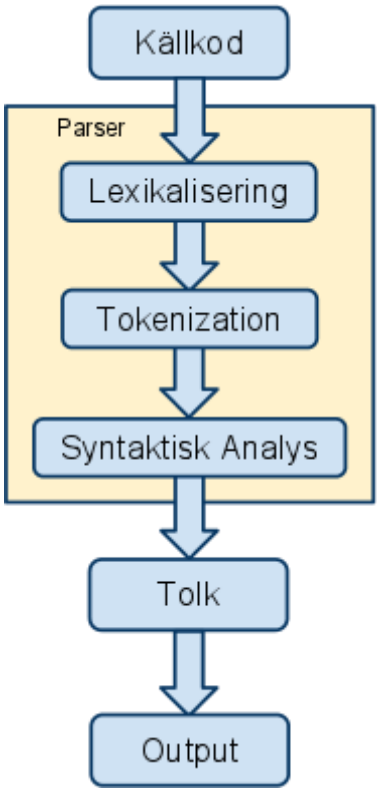
Visual Basic härstammar egentligen från BASIC och är menad att vara ett enkelt event-baserat språk, där programmets flöde styrs av event(händelser). Likt många andra språk har den influerats av C men här bryter Visual Basic från mängden, det är nämligen väldigt enkelt att förstå och programmera. Språket är mycket förlåtande i typ-behandling, till exempel omvandlas heltal till reella tal automatiskt vid division där resultatet ej är ett heltal, detta och mer gjorde språket mycket populärt bland nybörjare.

[6] *Programming Language Popularity, 2011-03-30,* http://www.langpop.com/

4 Metod

4.1 Parser

En parser sköter processen att omvandla med hjälp av givna regler en källkod till något en dator förstår. Detta sker ofta som en process av att behandla kod först med en lexer, sedan genom tokenization och sist genom att mata in resultatet som ett parse-träd till en tolk.



Figur 4.1.1 - Detta diagram beskriver dataflödet hos parser-processen och exekveringen av källkod, Tolken gör själva exekveringen medan parser-processen behandlar koden och omvandlar den till något en tolk förstår.

4.1.1 Lexikalisering

Lexikalisering är inom datalogi en process där en serie tecken omvandlas till lexikala element, dessa kallas även för “tokens”.[7] Lexikalisering innebär att den kod som matas in omvandlas till ord, konstanter, operatorer osv. Dessa behandlas vid ett senare skede. Syntaktiska fel kan dock inte upptäckas av lexaliserinsmetoden, då dess huvuduppgift är att tolka de individuella parametrarna.
För att ett språk ska lexaliseras till tokens används reguljära uttryck som har till uppgift att hitta delar i uttrycket . Här delas i princip ett uttryck upp i mindre delar till exempel kan uttrycket “summa=1+2” delas om till vektorn {summa,=,1,+,2}.

4.1.2 Tokenization

Uttrycket **summa=1+2** kan behandlas enligt nedan när de individuella tecknen skall bytas ut mot motsvarande “Token”.

Lexeme	**Token type**
summa	Identifier
=	Assignment operator
1	Number
+	Addition operator
2	Number

Tabell 4.1.2.1 – I denna tabell ser vi att summa har fått typen identifier, i vissa implementationer så motsvarar detta en variabel. Dessa tokens känns igen av parsern och har vissa distinkta attribut givet dom regler som matas in, på så sätt kan den längre fram “förstå” att en summa inte till exempel kan användas som en operator. Token referar till dessa ”egenskaper” som definierar grammatiken.

[7]Alfred V Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman, *Compilers Principles, Techniques, & Tools Second Edition*.Addison Wesley, ISBN 0-321-48681-1, sida 109

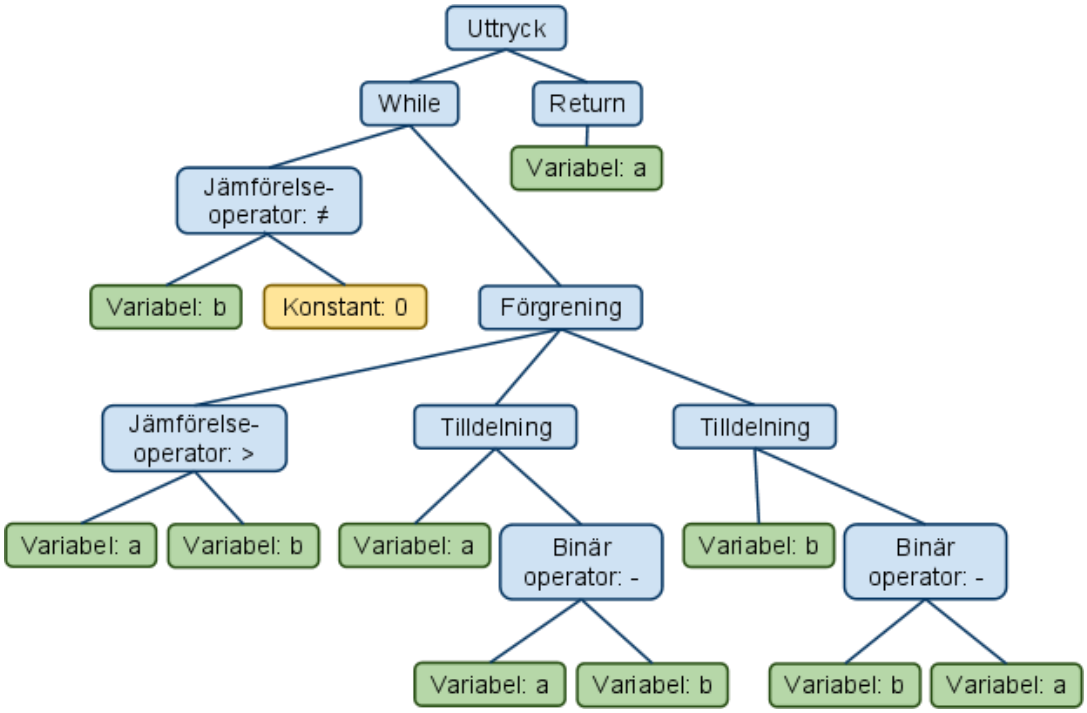
4.1.3 Syntaktiskt Analys

Syntaxen hos ett språk kan beskrivas av flödesdiagram eller BNF där de senare är den formella notationen för att beskriva ett givet språk[8]. I detta stadie kollas om alla token lyder under dem givna reglerna(givna i BNF), här kollas om det finns några syntaktiska fel, detta görs utan hänvisning till kontext, den kontext fria grammatiken kontrolleras, så regler så som “rätt ordning” och “rätt använda symboler” kollas, men inte huruvida en kod är korrekt, detta skulle kunna jämföras med rättningen av matematiska uttryck, här skulle 4/(1-1) godkännas men misslyckas längre fram i processen eftersom man dividerar fyra med noll. I källkod skulle detta vara som att godkänna tilldelningen **x = y** när y inte är deklarerad, detta skulle egentligen misslyckas vid körning. Själva processen av syntaktisk analys beskrivs ibland som en generering av ett syntax-träd, detta faktiska eller hypotetiska träd(beroende på implementering) kan te sig enligt exemplet på nästa sida.

[8] *BNF (Backus-Naur-Form), 2011-03-20,* http://otal.umd.edu/drweb/c++tutorial/lessons/BNF.HTM

```
while b ≠ 0
        if a > b
                a = a − b
        else
                b = b − a
return a
```

så ser det abstrakta syntax-trädet ut enligt *Figur 4.1.3* nedan.



Figur 4.1.3 - Det här abstrakta syntax-trädet beskriver koden och förenklar den syntaktiska analysen för att hitta syntaktiska fel, I många implementationer av parsers brukar tolken exekvera alla godkända syntaktiska uttryck parallellt med den syntaktiska analysen. Både token och värde syns i figuren.

4.2 Tolk

En tolks uppgift är att förstå semantiken i ett uttryck och utföra dem nödvändiga funktionerna som ett uttryck beskriver, därav heter det att en tolk “exekverar” koden. En tolk programmeras i ett metaspråk. I vissa implementationer tolkas koden parallellt med syntax analysen.

5 Implementation

5.1 EBNF

BNF har redan nämnts och är alltså ett metaspråk för definition av syntax. Extended BNF tillhör den familj av metasyntax som tillåter användandet av reguljära uttryck. Reguljära uttryck möjliggör en renare kod och är en anledning till varför den ibland används för att definiera programmeringsspråk som lätt kan bli rätt stora och komplexa. En annan skillnad mellan BNF och EBNF är att den senare använder kommatecken istället för whitespace/mellanslag. Nedan följer en lista över EBNF's syntax och deras användning.

Använding	**Notation**
definition	=
konkatenering	,
terminering	;
alternering	|
valbar	[...]
repetition	{ ... }
gruppering	(...)
terminal sträng	" ... "
terminal sträng	' ... '
kommentar	(* ... *)
speciellt tecken	? ... ?
undantag	-

Nedan är den EBNF Defintion som beskriver språket

```
(* CWELL in EBNF *)

program = function , {function};

function = function header, white space ,
        “{“ , white space ,
         code,
        “}” ;

code = { expression , ({";", expression} |{“_”,? new line ?,expression}), white
space } ;
function header = identifier , ”(“ , parameters , ”)”;
identifier = alphabetic character , { alphabetic character | digit } ;

expression = [comment, ? newline ?] , ( assignment | if clause | while clause |
callfunction | declaration) , [comment, ? newline ?] ;

callfunction = identifier , ”(“ , expression, { “,” , expression} , ”)”;
declaration = ”new”, identifier;
assignment = identifier , "=" , value;
parameters = declaration,{ “,” , declaration};

if clause = “if(” , statement , ”){” ,
                    code ,
                    ”}”,[ “else{” , code, “}”] ;

while clause= “while(” , statement , ”){” ,
                    code ,
                    ”}” ;

statement = value, ( “!=” | “==” | “>” | “<”  | “>=” | “<=”), value, {( “&” | “|” ),
statement };
value =  number | identifier | string | boolean | function header ;
number = [ "-" ] , digit , { digit } ;
string = "" , { all characters − "" } , "" ;
boolean = “true” | “false”;
```

```
comment = “//” , all characters;
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" | “a” | “b” | “c” | “d”
                     | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m” | “n”
                     | “o” | “p” | “r” | “”s | “t” | “u” | “v” | “w” | “x” | “y” | “z”;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
```

5.2 YACC

Yacc är en parser-generator skapad av Stephen C. Johnson vid AT&T för Unix operativsystem, den har använts i detta projekt för att generera en parser efter given definition, notationen för definitionen är en form av BNF[9]. Detta möjliggörs tillsammans med Flex(fast lexical analyzer generator), Flex är en lexikaliserings analysator som tillsammans med Yacc skapar en parser tillräckligt effektiv för detta ändamål, just Flex förenklar lexikaliserings-analysen då den till exempel tillåter användandet av reguljära uttryck vid definition.

5.2.1 Metaspråk

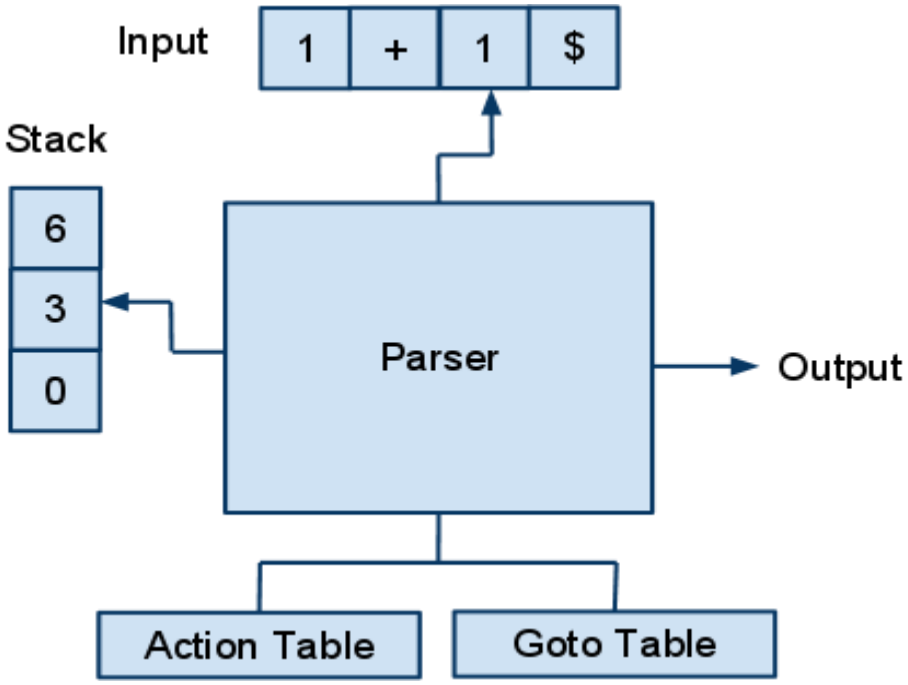
Yacc generar parsern i programmeringsspråket C, av enkelhet bibehålls detta som själva metaspråket, därav är också tolken skriven i C.

5.2.2 LALR-Parsing

Den algoritm Yacc använder sig av för att parsa, kallas LALR-parsing(Look-Ahead LR parser). LR-parsing(Left to Right parsing) är en parsing algoritm som läser indata från vänster till höger. Parsern genereras av den kontext-fria grammatiken(CFL rules) som ges i form av BNF liknande kod i Yacc. LR-parsers är rekursiva och snabba parsers som ofta finner fel mycket snabbare än till exempel LL-parser(Left most derivation) på grund av deras uppbyggnad, genom

[9] John R. Levine; Tony Mason; Doug Brow (1992), lex&yacc, O'Reilly & Associates Inc, ISBN: 1-56592-000-7, sida 51

att börja vid löven/hörnen kan den finna fel snabbare och avbryta eller informera[10].



Figur 5.2.2 - Det här är arkitekturen för en botten-upp tabell baserad parser, den stack som används i detta projekt kommer dock också inneha pekare för strängar, dessa är variabelnamn som används vid initiering och avläsning från minneshanteraren. Denna arkitektur är alltså *tillståndsbaserad*. Tecknet $ innebär "slutet av strömmen". *Action Table* är där koden förflyttas i en viss ordning för att sedan exekveras av tolken.

5.2.2.1 Exempel

I detta konkreta exempel illustreras parser-processen för input-strängen **1+1$** För att kunna parsa detta uttryck måste parsern ha en kontext-fri grammatik, nedan illustreras denna SLR(Simple LR) grammatik. Input-strömmen är alltså det uttryck som fås, nästa event-kolumnen beskriver den regel som känns igen

[10] Alfred V Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman, *Compilers Principles, Techniques, & Tools Second Edition*.Addison Wesley, ISBN 0-321-48681-1, sida 222

genom att referera till regelns nummer(Tillexempel används aldrig regel 4 eftersom siffran 0 saknas i input-strömen). Tillstånds-kolumnen visar vilket tillstånd eller steg parsern är på vid tillfället.

1. **E → E * B**
2. **E → E + B**
3. **E → B**
4. **B → 0**
5. **B → 1**

Tillstånd	**Input-ström**	**Output-ström**	**Stacken**	**Nästa event**
0	1+1$		[0]	Skifta 2
2	+1$		[0,2]	Reducera 5
4	+1$	5	[0,4]	Reducera 3
3	+1$	5,3	[0,3]	Skifta 6
6	1$	5,3	[0,3,6]	Skifta 2
2	$	5,3	[0,3,6,2]	Reducera 5
8	$	5,3,5	[0,3,6,8]	Reducera 2
3	$	5,3,5,2	[0,3]	Acceptera

Tabell 5.2.2.1 - Här har alltså **1+1** parsats och accepterats. “Nästa event”-kolumnen beskriver vad som kommer att ske med input-strömmen vid nästa event, den efterföljande siffran refererar till regeln som används från den kontext-fria gramatiken. Regel 4 används till exempel aldrig då värdet 0 inte finns i strängen. På stacken sparas dem tillstånd parsern har passerat.

Yacc använder som nämnt tidigare LALR-parsing, denna använder en finite-state machine (FSM) som är mycket komplicerat och svår för människor att konstruera, därför används parser-generatorer för att tillverka dessa, givet en

BNF kod[11]. Denna sorts algoritm använder sig av LALR-grammatik som är kapabel till att definiera mycket större klasser av språk än vad SLR är kapabel till. En fördel med LALR är att dess hastighet är linjär gentemot input-strängen endast och inte gentemot storleken inputen som ska behandlas. Redan i parsen kan felbehandling skötas samt om man så vill, kontext-baserad igenkänning, dvs. Semantik.

5.2.3 Action table

Som tidigare nämnt genereras parsern I Yacc av den kontext-fria grammatiken (CFL rules) som ges i form av BNF liknande kod. I CFL reglerna anger man även vilken C-kod som ska exekveras när en sådan regel påträffats av Yacc [12]. Eftersom CFL regler kan vara inbakade i andra regler alternativt anropa sig själva sparar man resultaten av dessa sub-regler i en *Action Table* för att senare ha möjlighet till att evaluera hela uttrycket, dvs Action Table motsvarar ett kortvarigt minne för ett uttryck. Implementationen av Action Table görs enklast med en länkad lista där varje plats ges av en struct/class som kan motsvara en Identifierare, konstant eller operation. Nedan följer ett exempel på hur ett uttryck representeras i Action Table.

Uttryck: summa = 1 + 1

ACTION TABLE	**TYPE**
1	CONSTANT
1	CONSTANT
+	OPERATION
=	OPERATION
summa	IDENTIFIER

[11] Alfred V Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman, *Compilers Principles, Techniques, & Tools Second Edition*.Addison Wesley, ISBN 0-321-48681-1, sida 266
[12] Johnson, Stephen;Yacc: Yet Another Compiler-Compiler, 2011-06-07, http://dinosaur.compilertools.net/yacc/index.html

5.3 Tolk

Tolkens uppgift är att gå igenom och exekvera uttrycken i *action table*. Tolken har en funktion *exec* där allt arbete sker. Funktionen tar emot ett element från action table, exekvera denna och anropa nästa element. För att exekvera i rätt ordning börjar tolken från det sista elementet som lades in i action table, för att därefter anropa sig själv rekursivt med nästa element tills det inte finns mer operationer att utföra. Detta kan ses som att först gå längst ner i syntax-trädet för att sedan gå uppåt och utföra en operation i taget. Efter att alla operationer är utförda i action table frigör tolken minnet som har allokerats av action table. Fortsättning på exemplet ovan följer här nedan:

1. getMemoryNode hämtar adressen till variabeln ”summa” som ligger i minnet (se 5.3.1 minneshantering) och tilldelar den värdet i nästa operation. Lägg märke till hur *exec* anropas igen med nästkommande operation som utförs innan *return* som returnerar adressen till ”summa” med resultatet av adressen.

Operation**:** Tilldelning (=)
Kod**:** `case '=': (getMemoryNode(p->opr.op[0]->id.value)->con = *exec(p->opr.op[1])); return &getMemoryNode(p->opr.op[0]->id.value)->con;`

2. Nu ska tolken utföra nästa operation som är av typen Plus (+). En ny konstant skapas för att kunna behålla det adderade värdet därav *newConstant*.

Operation**:** Plus (+)
Kod**:** `case '+':  return newConstant(TYPE_NUMERIC,exec(p->opr.op[0])->num_val + exec(p->opr.op[1])->num_val,NULL);`

3. Här hämtas konstanten 1 från minnet.

Operation**:** Hämta konstant
Kod**:** `return &p->con;`

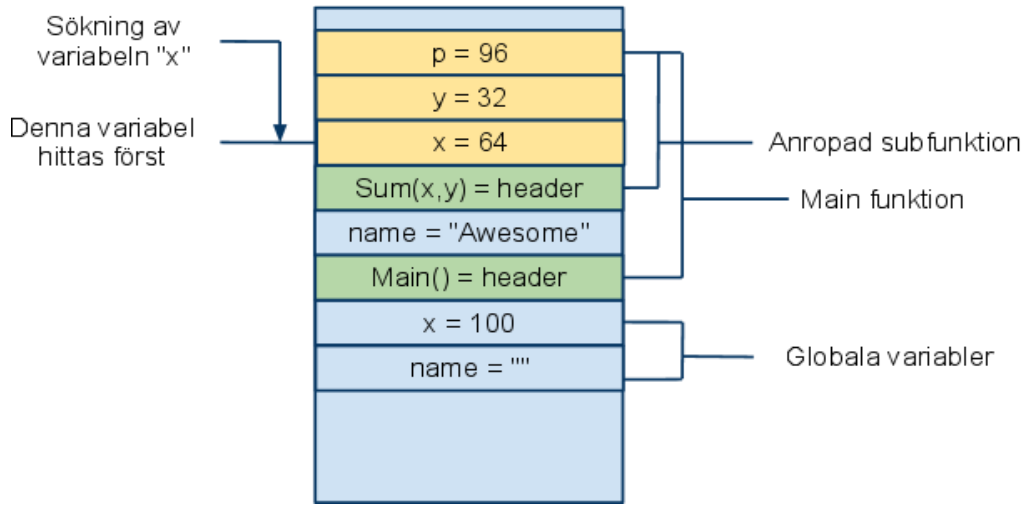
4. Nu hämtas nästa konstant och även i det här fallet är värdet 1.

Operation**:** Hämta konstant
Kod**:** `return &p->con;`
Efter dessa operationer innehåller nu variabeln “summa” värdet 2. Tolken frigör därefter minnet som används av action table

5.3.1 Minneshantering

Som minneshantering används en länkad lista, här lagras deklarerade variabler, parametrar och deras värden. En länkad lista av *structs* används, dessa lagrar variabelnamn, värde och datatyp. Samma lista används för globala som för lokala variabler. Insättning av deklarerade variabler görs uppifrån och sökning efter variabler görs i två stadier, först söks en variabel uppifrån och ned, på så sätt hämtas nyligen deklarerade variabler först, detta möjliggör en prioritering av lokala variabler över globala eller ovanför variabler från en högre tidigare scope(omfång) vid variabelnamn-kollision eller tvetydighet. När variabeln inte hittas i det loka omfånget, så sker nästa stadie. Då söker den nerifrån och upp efter globala variabler, den stannar slutligen vid första funktions header. i *Figur 5.3.1* nedan är den första headern “Main()”. Vid funktionsanrop lagras funktionens header, över denna lagring sparas värdet på dem insända parametrarna, på så sätt har den anropade funktionen tillgång till dessa värden.

```
1       name = “”
2       x = 100
3       Main(){
4               new name = “Awesome”
5               print(sum(64,32))
6       }
7       sum(new x, new y){
8               p = x+y
9               return p
10      }
```



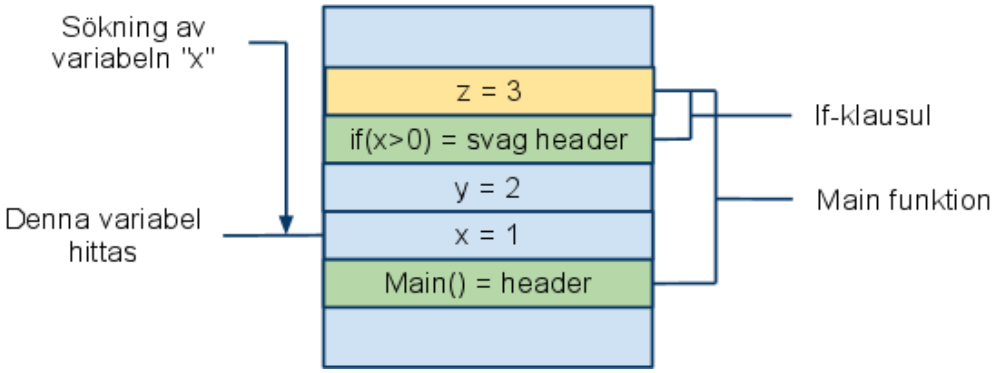
Figur 5.3.1 - De gröna cellerna visar den lagrade funktionsheadern, deras värde i detta exempel är deras typ, nämligen ”header”. Allt lagrat över en funktions header är variabler inom funktionens scope(omfång). Allt lagrat under den första funktionen, i detta fall “Main()”, är globala variabler.

5.3.1.1 IF & WHILE

När en variabel deklareras inne i en *If* eller *While* klausul så ska dessa variabler inte kunna användas vid ett senare skede, utanför klausulen, detta är omfång-conceptet eller den s.ka scope av en variabel. Det är av denna anledning som If och While klausuler behandlas som funktions-anrop vid minneshantering, därav försvinner alla deklarerade variabler när klausulen avslutas och stängs. Det enda som skiljer dessa klausuler från funktions-anrop är att minneshanteraren inte stannar up vid en If-header eller en While-header vid variabelsökning, på så

sätt kan man inne i klausulen använda sig av variabler deklarerade vid ett tidigare skede, i ett högre scope(omfång). Sen nedanstående exempel.

```
1       Main(){
3               x = 1
4               y = 2
5               if(x>0){
6                       z = 3
7                       print(y+x)
8               }
9       }
```

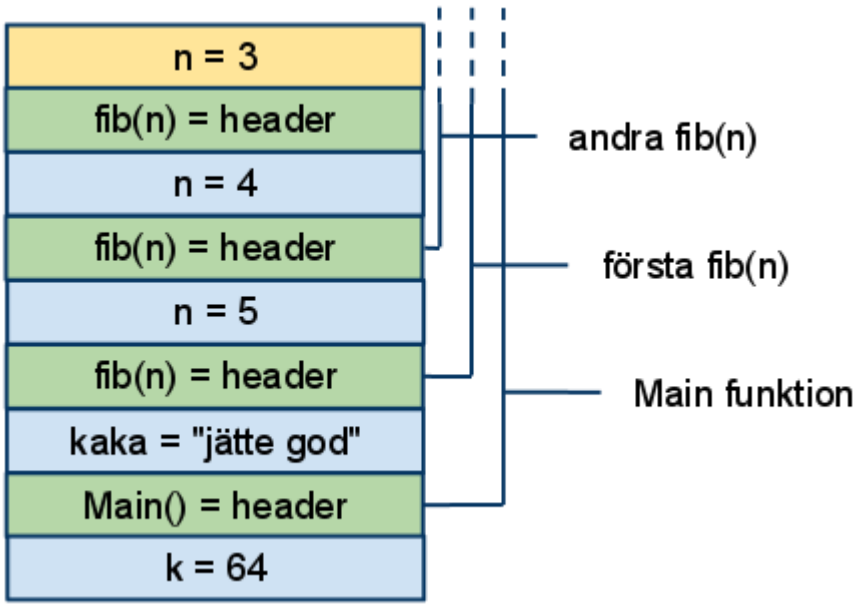


Figur 5.3.1.1 - När en variabel sökning görs stannar sökningen så fort den når en funktions header, detta gäller dock inte If och While klausuler som ignoreras. På så sätt kan man använda variabler deklarerade över en If-klausul, samtidigt kan man inte använda variabler deklarerade inne i en sådan klausul utanför själva klausulen, eftersom variablerna tas bort när klausulen avslutas.

5.3.1.2 REKURSION

Eftersom minneshanteraren är uppbyggd på detta dynamiska sätt så finns även möjlighet för rekursion, detta är på grund av att omfånget hos variabler alltid är i den lokala klausulen. Till exempel så kan en funktion som själv har ett par variabler ropa på sig själv och där återskapas nya lokala variabler i ett djupare omfång som inte berör den tidigare. Rekursion är därav möjlig. Se nedanstående exempel, samma exempel har exekverats på en android och linux tolk för språket och finns att se i Appendix A.2 och A.3.

```
1       k = 64
2       Main(){
3               kaka = “jätte god”
4               print(fib(5))
5       }
6
7       fib(new n){
8               if(n < 2){                                    Breakpoint
9                       return n
10              }else{
11                      return fib(n-1)+fib(n-2)
12              }
13      }
```



Figur 5.3.1.2 – En *Breakpoint* har placerats vid if-klausulen(rad 8) för att avbryta när den är sann, på så sätt kan vi se hur minnet ser ut innan alla funktioner returnerar sina värden. Givetvis kan minnet se annorlunda ut beroende på tolkens uppbyggnad, men funktionen är alltid den samma.

6 Resultat

Exempel på exekvering av tolken med motsvarande källkod.

6.1 Källkod

```
//välkommen till programmeringsspråket CWell
//allt efter "//" är kommentarer.
print("#### CWELL SAMPLE ####\n")
//nedan följer tilldelning av variabler.
sum = 1+1
more = 1 + 2 * 4 + 3

//om man vill ha fler statement på en rad avslutar man med ";"
cool = 30; bool = -44

//print skriver ut variabler/uttryck
print(sum)
print(more)
print((25 * 2) + 54 + (3 * 3) + (35 * 2 * ((-432) + 444)) + 2)
print("Hello World!")

//overcasting sker i bakgrunden.
sum = 55
print("The value of sum is " + sum*2)
print("Hello " + "KTH")
print(13 + " is a magic number")

//man behöver inte deklarera variabler utan de får standard värden
//om de är odeklarerade innan.
print("Undefined variable has the value " + undef)
```

```
print("#### OPERATORS ####\n")
//operatorer i CWell
a = 2; b = 1;
print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(a == b)
print(a != b)
print(a < b)
print(a > b)
print(a & b)
print(a | b)

print("\n#### IF-ELSE-WHILE ####\n")
//nedan följer en if-statement
if(a == 1){
    print(255)
}

//här är en if-else-statement.
if(a == b){
    print(kth)
}else{
    print(500)
}

//en while loop följer nedan.
print("While loop")
x = 5
while(x > 0){
    //skriver ut de nuvarande värdet på x.
    print(x)
    //minskar x så att loopen terminerar.
    x = x-1
}
```

```
print("\n#### SCOPE ####\n")
//alla variabler deklarede i en if/if-else eller while-loop är
//inte giltiga utanför (scope).
if(1){
    hidden = 4+4
    print("Hidden inside if " + hidden)
}
print("Hidden outside if " + hidden)

//avslutningsvis avslutar vi med fibonacci sekvens
print("#### TESTING FIBONACCI SEQUENCE ####\n")

n = 31; a = 0; b = 1; sum = 0; i = 0

while(i < n){
    print("FIB:" + i + " " + a)
    sum = a + b
    a = b
    b = sum
    i = i+1
}
```

6.2 Utskrift

```
#### CWELL SAMPLE ####
2
12
955
Hello World!
The value of sum is 110
Hello KTH
13 is a magic number
Undefined variable has the value 0

#### OPERATORS ####
3
1
2
2
0
1
0
1
0
3

#### IF-ELSE-WHILE ####
-1
While loop
5
4
3
2
1

#### SCOPE ####
Hidden inside if 8
Hidden outside if 0
```

TESTING FIBONACCI SEQUENCE
FIB:0 0
FIB:1 1
FIB:2 1
FIB:3 2
FIB:4 3
FIB:5 5
FIB:6 8
FIB:7 13
FIB:8 21
FIB:9 34
FIB:10 55
FIB:11 89
FIB:12 144
FIB:13 233
FIB:14 377
FIB:15 610
FIB:16 987
FIB:17 1597
FIB:18 2584
FIB:19 4181
FIB:20 6765
FIB:21 10946
FIB:22 17711
FIB:23 28657
FIB:24 46368
FIB:25 75025
FIB:26 121393
FIB:27 196418
FIB:28 317811
FIB:29 514229
FIB:30 832040

7 Diskussion

Vi som studenter har varit intresserade av programmering en längre period och allt eftersom vi använt oss av programmering har vi funderat på hur det skulle gå till att konstruera ett programmeringsspråk. Då vi inte har någon tidigare erfarenhet av kompilator-konstruktion eller tolk-implementering har vi studerat diverse litteratur för att konstruera programmeringsspråket i rapporten. Idéer till designen av språket har vi erhållit från C och BASIC som vi båda har goda erfarenheter av. Det konstruerade språket skulle vara influerat av dessa två språk. Rapportens fokus har varit på implementationen av ett programmeringsspråk och inte på utformningen av detta språk, dem design-val som gjordes vara bara godtyckligt valda för att inte språket allt för mycket skulle likna ett existerande språk men samtidigt vara tillräckligt intressant att implementera. Hela projektet har varit ”proof-of-concept” där vi utvecklat ett språk och sedan konstruerat ett par tolkar för den.
Yacc och Flex förenklade processen för oss men försvårade samtidigt implementeringen av till exempel minneshantering, eftersom vi använde C som meta-språk så försvårade det hanteringen av strängar och allokeringen av större data. Vi hade inte heller någon erfarenhet av Yacc eller Flex vilket krävde mycket läsning i manualer och exempel. Slutsatsen av detta är att verktyg kan förenkla processen om man känner till verktygen sedan innan.
Minneshanteringen blev dock en mycket intressant lösning och vi kände oss väldigt nöjda med omfångsbehandlingen. Tyvärr fanns inte heller några exempel på funktions-implementering för Yacc, vare sig i orginal-manualen eller i annan litteratur skriven av andra. Detta försvårade implementationen av funktioner i vårt språk, detta är beklagligt då vi var intresserade av att testa omfångkonceptet i minneshanteringen när man ropar på funktioner, därför implementerade vi språket i JAVA med funktionshantering. Programmet kunde sedan visuellt illustrera minneshanteringen, som visade sig fungera som tänkt.
Vi märkte att rekursion är långsam jämfört med en while-loop och ännu långsammare på Android-implementeringen.

Själva definitionen av språket var rätt enkel då vi båda var överens om hur språket skulle se ut från början, vi hade goda erfarenheter från C och BASIC och tog inspiration från dessa. Men att sedan skriva ned det i BNF/EBNF är inte lika enkelt, speciellt med den intressanta och smått krångliga semikolon lösning vi definierat för språket, både C och JAVA implementationen av tolken kunde tillslut hantera dem vilket var mycket givande.

Allan Jerjas implementerade språket i C med Yacc och Flex. Båda hjälptes åt för minneshanteringen för lagring av variabler i form av structar.
Adam Sam implementerade språket i Java för PC och Android mobil[se Appendix A] för att demonstrera funktionshantering i samband med minneshantering
Språkdefinition samt design gjordes tillsammans samt denna avhandling.

Slutligen kan vi nämna att utveckling av ett programmeringsspråk sker i flera steg där man måste planera varje steg mycket noga för att lyckas med sitt språk. Om man inte sedan tidigare har kunskap inom kompilator-konstruktion måste man först läsa till sig informationen innan man börjar skriva kod, detta underlättar för konstruktionen av språket. Man måste även vara bekväm med eventuella verktyg och val av meta-språk då problem kan uppstå även kring dessa.
Individuella Script-språk kan vara mycket givande för ett stort program och har många användningsområden, till exempel som tillägg till existerande program. Det gör ett program mer dynamiskt och öppnar upp möjligheter för tillägg och expansion av program, till exempel I form av uppdatering där inte hela programmet behöver uppdateras utan endast ett script. Därför uppmanar vi alla att testa på att konstruera ett eget språk.

Referenslista

Turing, Alan; 'On Computable Numbers, with an Application to the Entscheidungsproblem', *Proceedings of the London Mathematical Society* (Series 2, volume 42 (1936-37), pp. 230-265)

Alfred V Aho; Monica S. Lam; Ravi Sethi; Jeffrey D. Ullman, *Compilers Principles, Techniques, & Tools Second Edition,* Addison Wesley, ISBN 0-321-48681-1

BNF (Backus-Naur-Form), 2011-03-20, http://otal.umd.edu/drweb/c++tutorial/lessons/BNF.HTM

Friedman, Daniel P; Mitchell Wand, Christopher T. Haynes (1992), *Essentials of Programming Languages (1st ed.),* The MIT Press, ISBN 0-262-06145-7.

Levine , John R.; Tony Mason; Doug Brow (1992), *lex&yacc*, O'Reilly & Associates Inc, ISBN: 1-56592-000-7

Programming Language Popularity, 2011-03-30, http://www.langpop.com/

Lévénez, Éric, *The History of Programming Languages,* 2011-04-01. http://oreilly.com/news/languageposter_0504.html

Loui, Ronald. IEEE Computer, 2008, *In praise of scripting*

Raiter, Brian. *Brainfuck,* 2011-03-27. http://www.muppetlabs.com/~breadbox/bf/

Tom Smedsaas; Tobias Wrigstad. 2011-03-15. *Imperative and objekt-oriented programming, page 1*. http://www.it.uu.se/edu/course/homepage/imperoop/ht08/forelasningar/del13/del13.pdf

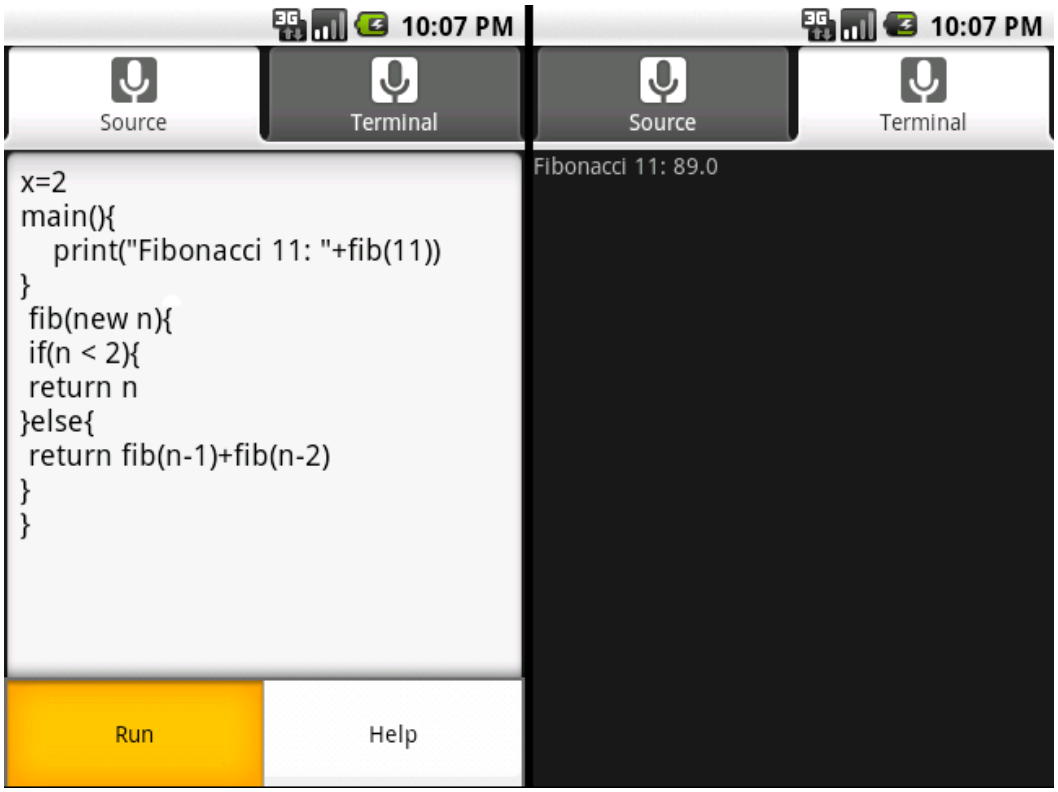
Johnson, Stephen;Yacc: Yet Another Compiler-Compiler, 2011-06-07, http://dinosaur.compilertools.net/yacc/index.html

Appendix A

Olika versioner av tolken implementerade i Java för Ubuntu Linux och Android OS för smartphones.

Figur Appendix A.1 – Implementation av tolken i java på en linux maskin

Figur Appendix A.2 – Tolken kör här en rekursiv fibonacci-metod med en breakpoint när den når n=1(så att minneshanteraren behåller alla värden för oss att se)



Figur Appendix A.3 – Tolken implementerad i java för Android OS, här ser vi den rekursiva Fibonacci koden igen utan breakpoint. Resultatet visas i Terminal fliken.