Programspråket Rnd och andra liknande språk

Mats Persson

June 11, 1996

Sammanfattning

Rnd är ett programspråk som jag själv konstruerat. Det är ett mycket annorlunda språk. Det är helt interpreterande och ganska litet, men är ändå nästan fullt programmerbart. Det fungerar ungefär som ett makrospråk som exekverar under själva makroexpansionen. Det här programspråket är implicit evaluerande och självmodifierande, vilket kan vara praktiskt när man vill blanda kod och text i till exempel databaser. Tyvärr medför detta att det är svårt att implementera effektivt. Rnd jämförs sedan med andra liknande språk såsom m4, tcl, perl, Lisp och LPC, och för- och nackdelar kommer att utvärderas. Rapporten kommer även att undersöka kompilering av Rnd, och hur det begreppet är definierat. Den beskriver också nya begrepp som explicit och implicit evaluering och undersöker eval-funktionen i andra språk.

Innehåll

1 Inledning **4**

1.1 Inledande frågeställning 4

1.2 Varför ett nytt språk? 6

1.3 Litteratur 6

1.4 Grundläggande begrepp 7

1.5 Historisk bakgrund 8

1.6 Designmål 9

1.7 Udda egenskaper 9

2 Programspråket Rnd **11**

2.1 Språkbeskrivning 11

2.1.1 Syntax och Semantik 11

2.1.2 Text 12

2.1.3 Variabler 13

2.1.4 Quotning 13

2.1.5 Aritmetiska uttryck 14

2.1.6 Datatyper 15

2.1.7 Scanning 15

2.1.8 Tabeller 16

2.2 Övriga konstruktioner 20

2.2.1 Funktioner 20

2.2.2 Parametrar 21

2.2.3 Villkors-satser 22

2.2.4 Loopar 22

2.2.5 Saknas 23
2.3 Ett större programexempel 23

3 Jämförelser med andra liknande språk 27
3.1 m4 28
3.2 Tcl 28
3.3 Perl 29
3.4 LISP 30
3.5 LPC 30
3.6 Jämförelser 30
3.6.1 Implicita villkor 31
3.6.2 Stora arrayer 31
3.6.3 Implicit/explicit evaluering 32
3.6.4 Filter 33
3.6.5 Praktiskt 33
3.6.6 Lat evaluering 34
3.6.7 Funktionskonstruktorer 35
3.6.8 Funktionell programmering 35

4 Teori 36
4.1 Grundläggande teori 36
4.2 Implicit/Explicit evaluering 38
4.3 Teori om kompilering 39
4.4 Kompilering 40

5 Utvidgningar 42
5.1 Felhantering 42
5.2 Stränghantering 42
5.3 Fil- och pipehantering 43
5.4 Start-stop evaluering 43
5.5 Dumpar 43
5.6 Exakt get 44
5.7 Flyttal 44
5.8 Inbyggda funktioner 44

5.9 Komplexa argument vid programstart 44
5.10 Ta bort entryn 44
5.11 Lokala variabler och namn på parametrar 45
5.12 Strängvariabler 45
5.13 Strängar som index 45
5.14 Flerdimensionella arrayer 45
5.15 Externa funktioner 45
5.16 Reguljära funktionsnamn 46
5.17 Temporära tabeller 46

6 Implementation 47

6.1 Begränsningar 47
6.2 Nackdelar 48
6.3 Implementation av interpretatorn 48

7 Övrigt 50

7.1 Användningsområden 50
7.2 Utvärdering 50

Kapitel 1

Inledning

Det finns många olika trender inom eller grupper av programspråk. En traditionell grupp av språk är de imperativa såsom Pascal och C, och modernare som C++, Java och LPC[1] . I denna grupp finns även flera objektorienterade språk och större programsystem är ofta utvecklade i dessa. En annan grupp är de mer abstrakta deklarativa och funktionella språken, som Lisp och ML. De trender som har dykt upp är de objektorienterade och en typ som ofta används för små "on-the-fly" program, till exempel tcl, perl, Python och många shell-språk. Dessa är nästan alltid interpreterande och är inne på ett område som jag kommer att kalla kodevaluering. De kan evaluera kod inuti själva programmet men då genom att explicit ange det. I den här rapporten kommer jag att undersöka hur pass kodevaluerande dessa är och även att visa och jämföra med programspråk som är implicit evaluerande och självmodifierande.

Den här rapporten handlar alltså om en speciell typ av programspråk. De är interpreterande och används ofta till små program. Först kommer en beskrivning av vad det hela handlar om, sedan en undersökning och en motivering till att konstruera ett nytt språk. Som en hjälp för fortsatt analys definieras även en del begrepp. Vidare innehåller rapporten en beskrivning av det nykonstruerade programspråket Rnd, och jämförelser med andra liknande existerande språk. Slutligen kommer en del teori om kompilering av den här typen av språk.

1.1 Inledande frågeställning

Tänk dig ett programspråk som utformas, växer och förändras medan du skriver det. Du är inte begränsad till de gamla vanliga if-satserna och for-

[1]LPC är ett objektorienterat språk som liknar C. Det beskrivs utförligare i kapitel 3.5 och i [PH90]

looparna, utan du kan konstruera nya satstyper efterhand som de behövs. Det finns inte längre någon begränsning i hur du kan manipulera datastrukturer och bygga operatorer för dem. Du är inte heller besvärad av nya versioner av programspråket, utan grundstommen är så liten och enkel att den inte förändras. Det är du själv som kontrollerar förändringarna. Det är också ett språk som har inget och allt. Det har alltså enkelt i grunden, men kan byggas ut obegränsat. Andra programspråk har väggar, tak, dörrar och fönster till att bygga vilka hus man vill med, men man kan inte bygga en bil med dem. Det här programspråket har skruvar, spik, plåt och brädor i alla dimensioner och det är bara att pussla ihop det man vill ha.

Anledningen är nog att en vanlig programmerare inte skulle klara av att använda det. Det skulle sakna färdiga mallar för att hantera programmeringsproblemen, vilket är vad programspråk ofta består av. Ju mer komplicerade problem desto mer komplicerade mallar behövs, och desto ohanterligare och mer statiskt blir språket. Ett exempel på detta är objektorienterade språk och deras färdiga klasser och metoder. Hela konceptet är ett enda stort verktyg, som man använder för att snabbt och säkert lösa ett problem. En skicklig programmerare undviker stora verktyg eftersom han tappar kontrollen över koden, eller så modifierar han klassen vilket egentligen inte var meningen. Själva idén med objekt behåller han, eftersom det är ett av många sätt att lösa problemen.

Programspråket C är ett populärt språk. Det är lätt att skriva nästan vilka program och system som helst, och det har även en särskild preprocessor. Språket har ganska små byggstenar och det är en av anledningarna till att det blivit så användbart. Med hjälp av preprocessorn sker en del evaluering i kompileringsfasen och med denna kan man då automatisk anpassa programmet till olika plattformar och förutsättningar [Sal96]. Här ligger en del av kraftfullheten. Nackdelen är att man får två programspråk i ett system, ett makrospråk och språket C.

Tanken med C++ var att man skulle skriva standardmetoder och använda dessa. Men programmerarna använder istället den kraftfullare kompileringen istället för att använda standardklasser.

Det borde vara möjligt att konstruera ett språk som exekveras i själva kompileringen och inte producerar en statisk maskinkod. Det borde leda till ett interpreterande språk som inte kan kompileras i den meningen att exekverbar maskinkod produceras. Språkets motsats borde vara ett språk som går snabbt och enkelt att kompilera, till exempel assembler.

Detta hypotetiska språk kommer jag att kalla K, för att det är kompileringsexekverande och kodevaluerande.

1.2 Varför ett nytt språk?

Det finns inte många programspråk som liknar K. En del språk har vissa egenskaper som kan finnas i K, eller har likheter med K. Preprocessorn i C är ett exempel, och det kommer att nämnas fler i den här rapporten.

Jag har försökt, eller tror mig ha konstruerat, ett språk som kommer väldigt nära K. Anledningen till detta är att visa att språket K borde finnas. Detta nya programspråk har jag kallat Rnd.

Det handlar inte om att jag gjort det perfekta språket. Resultatet har däremot imponerat mig själv flera gånger och har både förvirrat och imponerat andra. Efter att ha gjort ett antal små program, så blir jag faktiskt lite irriterad över vanliga programspråks otymplighet.

En annan anledning till att jag konstruerade språket Rnd, var att jag ville utforska och pröva hur pass lätt det är att programmera i sådant språk. I slutet av den här rapporten kommer ett kapitel med min utvärdering.

Titeln på den här rapporten kunde lika gärna vara “Implicit och Explicit Evaluering”. Just den här typen av evaluering ses ofta med oblida ögon av andra programspråksexperter främst därför att den är ganska långsam. Men det skall ju inte hindra någon från att undersöka den här egenskapen.

1.3 Litteratur

I mina sökningar i litteraturen efter liknande språk eller liknande teori, har jag inte hittat mycket. Inte ens något språk i samma klass eller av samma typ. Det som kommer närmast är nog makrospråket m4.

Det finns en del teori om kompilering i [ASU86] och jag har hämtat en del definitioner därifrån, men den bygger mycket på redan etablerade kompileringsmetoder och teori. Den är låst vid att parsa språk och sedan generera maskinkod.

[Kam90] har flera exempel på interpreterande språk, och har även en del teori och definitioner på egenskaper hos dessa språk. Jag har använt flera ideer och exempel från den här boken. Den innehåller även en enkel evaluator, men denna är mest avpassad för Lisp-liknande språk.

[JGS93] handlar om partialevaluering vilket gett mig en del idéer om evaluering och kompilering. Den innehåller även en beskrivning av lambdakalkyl som kanske ligger närmast språket K. [Sal96] handlar om partialevaluering med hjälp av preprocessorer.

För lite mer komplex analys och klassificering av språket kan man läsa [HU79] som handlar om formella språk och automatateori. Någon mer avancerad analys kommer dock inte att göras i den här rapporten. Däremot kommer den formella grunden för Rnd att analyseras och en del teori kring denna kommer att behandlas.

Övriga referenser är dels några proceedings, [ACM87], och manualer över perl [WS90], tcl [Ous94] och m4 [Sei94] samt texter från Usenet News.

1.4 Grundläggande begrepp

I den här rapporten finns en del begrepp som kan behöva förklaras och definieras. De kanske skiljer sig en del från normala datavetenskapliga definitioner men det är så här de kommer att användas i den här rapporten.

Uttryck

Ett uttryck är en sekvens av identifierare, operatorer och parenteser. En identifierare är vanligtvis bokstäver och anger ett namn på en variabel eller tabell. En operator är de fyra räknesätten och andra liknande funktioner, ofta bestående av ett tecken. Parenteser är vanliga parenteser `()`, hakparenteser `[]`, måsvingar `{}`, apostrofer `` ` ´ `` eller liknande tecken med höger och vänstertyp.

Subuttryck

Ett subuttryck är ett uttryck som står innanför parenteser av vilken typ som helst. Man kan även kalla det för term. Ett subuttryck kan förstås också evalueras.

Evaluering

Evaluering av ett uttryck ger som resultat ett värde. Evalueringen använder sig av de globala variablerna och tabellerna, och anropar eller hämtar dessas värde, samt beräknar ett värde. Under evalueringen kan variablers värde och tabellers innehåll ändras. I den här rapporten förekommer även begreppen implicit och explicit evaluering, vilka förklaras noggrannare i kapitel 4.2.

Kodevaluering

Kodevaluering är evaluering av programkod. Det är ungefär samma sak som interpreterande programexekvering, men skillnaden är att koden finns i uttryck och inte i den vanliga typen av programrader.

Substituering

Ersätta ett uttryck eller subuttryck med det evaluerade uttrycket. Detta sker genom att ta bort uttrycket ur indatasträngen och stoppa in resultatet på samma plats i strängen. Ett liknande begrepp är termomskrivning.

Tabell

En tabell är en lista med par av (index,entry). Indexvärdet är ett heltal, och entryt är en sträng. Man kan slå upp ett entry i tabellen genom att ange ett index. Detta kallas på en del ställen i den här rapporten för funktionsanrop. Tabellen själv kan även ha ett heltalsvärde, vilket är det senast uppslagna indexet.

Självmodifierande

Ett program är självmodifierande om det ändrar, lägger till eller tar bort kod eller datastrukturer i programmet självt under exekveringen.

Kompilering

Att översätta hela källkoden till intermediär-, assembler- eller maskinkod, utan att exekvera den. Varje källkodsegment, som kan bestå av en rad, sats eller uttryck, översätts till ett ekvivalent kodsegment. Inget av den ursprungliga källkoden får finnas kvar. Program där bara vissa satser eller uttryck kompileras, kan man kalla partiellt kompilerade.

Interpretering

Översättning, tolkning och exekvering av kod som inte är kompilerad eller förkompilerad. Källkoden körs direkt utan någon transformation först.

Konkatenering

Slå ihop två strängar till en sträng, genom att sätta den ena direkt efter den andra. Strängen kan förstås innehålla kod, tal eller vanlig text.

Quotning

Att sätta högerapostrof ` och vänsterapostrof ´ runt ett uttryck eller text, vilket gör att det inte kommer att bli evaluerat.

1.5 Historisk bakgrund

Det hela började med att jag hittade ett program, skrivet av Marcus J. Ranum, på News för åtta år sedan. Programmet kunde generera slumpmässiga texter och tärningslag till rollspel. Det läste in en enkel beskrivning av vad som skulle genereras, tog en slumpmässigt text ur en tabell, och skrev ut resultatet. Detta "beskrivningsspråk" var inte särskilt flexibelt och jag jobbade länge på att få det att generera den text jag ville ha. Jag kom fram till att det måste vara fullt programmerbart för att det skulle vara användbart.

Den första versionen hade ett BASIC-liknande språk. Det fanns `repeat` och `until` för loopar, `set` för att tilldela en variabel, `stat` för att skriva ut en variabel, `table` för att läsa in tabell från en annan fil, `#<num>` för en numerisk label, `jump` för att hoppa till en label, `select` för att göra switch till flera labels, och `subtable` för att skriva en tabell. Man kunde även anropa en tabell med måsvingar . Det fanns en ännu tidigare version med tvådimensionella tabeller och med en annan syntax på anrop.

Tidigare hade jag även jobbat på en smart och snabb evaluator för aritmetiska uttryck. Jag kom då på idén att utveckla en slags super-evaluator, som då skulle likna ett funktionellt språk, och evaluera avancerade uttryck.

Detta resulterade i en sammanslagning av slumpspråket och idéerna till det funktionella språket. Det var tänkt att det skulle användas till rollspel, men sen upptäckte jag förvånat att det gick bra att programmera i och att det kunde användas som ett mer allmänt programspråk.

Den första och inte särskilt flexibla versionen var ganska lik tcl. Nästa version var den evaluerande varianten, men den hade väldigt invecklad scanner. Den tredje versionen fick en enklare och snabbare tillståndsmaskin. Den tredje slutliga beskrivs mer utförligt i implementationskapitlet.

När den sista versionen var klar, började jag skriva den här rapporten, som egentligen angriper problemet från ett annat håll. Alltså att hitta ett bra kodevaluerande språk. Det designmål och den beskrivning av språket kan man se som skriven innan den här rapporten.

1.6 Designmål

Jag har strävat efter att få språket så rent, ortogonalt och enkelt som möjligt, utan onödigt syntaktiskt socker, samtidigt som man bör kunna konstruera vilka funktioner som helst. Det innehåller en del experimentanda och jag har tryckt in många ideer, vissa är kanske befängda medan andra kanske revolutionerande. Jag vet inte var språket hamnar.

Jag har inte haft som mål att det skulle vara ett snabbt språk eller att det skulle vara enkelt att lära sig. Ej heller att det skulle vara kompatibelt med något annat programspråk.

1.7 Udda egenskaper

Rnd har en del karakteristiska saker i språket som man sällan ser i andra språk. Det här är en översikt av dem:

- Programkoden ligger i "strängar" som i sin tur ligger i listor.
- Syntax kollas under programkörning då programsträngarna ofta konkateneras till exekverbar kod.

- Kan modifiera koden under exekvering (i vissa fall nödvändigt).
- Väldigt svårt att kompilera.
- Styrsatser saknas helt (IF och LOOP får simuleras på annat sätt).
- Ingen skillnad på arrayer, funktioner eller variabler (första klassens objekt).
- Arrayerna är egentligen länkade listor, som jag kallar tabeller.
- Oändligheten är definierad som konstant och kan indexera i tabeller.
- Operatorn $ generar slumptal.
- Implicit evaluering, vilket innebär att allting normalt evalueras.
- Flerpassubstituering.
- Fördefinierade ord saknas. Det finns alltså inga biblioteksfunktioner och språket är bara syntax.

Flera av de här egenskaperna får lite speciella följder och man får programmera på ett lite annorlunda sätt.

Kapitel 2

Programspråket Rnd

2.1 Språkbeskrivning

Det här kapitlet beskriver de grundläggande uttrycken i språket. De är text, variabler, tabeller och aritmetiska uttryck. Dessutom kommer quotning, syntax och semantik, och scanning i Rnd att beskrivas.

Under varje underrubrik kommer designidéen bakom den funktionaliteten att motiveras och förklaras. Det visas också en BNF-grammatik för varje typ av uttryck, men det kommer ingen fullständig BNF för hela språket. Något som man kanske inser efter att ha förstått Rnd.

2.1.1 Syntax och Semantik

Ett programspråk måste ha en syntax för att man skall kunna ange hur programkoden skall representeras. Den består av mängden giltiga tecken och hur de kan kombineras till ett uttryck. Det är vanligt att man behöver något sätt att avskilja programsatser eller uttryck. Eftersom Rnd inte har programsatser i egentlig mening (i tabeller kan det se ut som satser) utan bara uttryck, så finns det bara olika slags parenteser.

Rnd har egentligen flera olika syntaxnivåer. Toppnivån är bara en följd av uttryck och har därför en enkel semantik och bara olika parentestyper. För varje parentestyp finns sedan egen syntax och semantik. På toppnivån i Rnd är det en enkel syntax och man kan dela upp den i tre olika typer av subuttryck. Hakparenteser `[]` används för aritmetiska uttryck och de har normal infix syntax. Måsvingar `{}` används för tabeller, som fungerar som funktioner och arrayer och de har en särskild semantik. Apostrofer `` ` ´ `` används för text som inte skall evalueras. Det normala för alla uttryck är att allting evalueras.

```
There are three types {atable 1 foo} `quoted´ and [x+1]
```

Exempel 1: Ett exempel på Rnd kod

Aritmetiska uttryck och tabeller har en särskild semantik, och denna beskrivs under respektive underkapitel. Lägg märke till att vanliga parenteser inte används på toppnivån. En anledning till detta var att jag inte ville blanda ihop Rnd:s utseende med Lisp.

Figur 2.1: BNF-grammatiken för toppnivån

$$
\begin{array}{lll}
<\mathit{prog}> & ::= & <\mathit{expr}>\ <\mathit{prog}> \mid \epsilon \\
<\mathit{expr}> & ::= & <\mathit{text}> \mid <\mathit{table}> \mid <\mathit{quoted}> \mid <\mathit{arithmetic}>
\end{array}
$$

2.1.2 Text

I ett språk med implicit evaluering, och där koden finns inuti subuttryck, kan man även ha implicit “print”. Den här tekniken används ofta av typsättningspråk, som till exempel LaTeX, där text som skall typsättas inte markeras på något speciellt sätt. Det minsta tänkbara språket har alltså en interpretator som läser in koden och skriver ut den igen utan att göra något. Kommandot “cat” i Unix fungerar just på det sättet och kan därför kallas interpretator i vid definition.

Grundtanken med den första versionen av språket, var att behandla och skriva ut text. Det är anledningen till att Rnd har implicit “print”.

Om interpretatorn läser in något som inte är ett subuttryck eller ligger inuti ett subuttryck, så skrivs det ut på `STDOUT` direkt. Nyrad-tecken och tabulatortecken ignoreras, så om man vill skriva ut dessa måste man skriva `\n` eller `\t`. Även en del andra specialtecken måste skrivas ut med backslash, som `\{`, `\[` och `\‘`, eftersom dessa markerar början av subuttryck.

Figur 2.2: BNF-grammatiken för text

$$
\begin{array}{lll}
<\mathit{text}> & ::= & <\mathit{chars}>\ <\mathit{text}> \\
<\mathit{chars}> & ::= & \mathbf{anyother} \mid \backslash\ <\mathit{special}> \mid \epsilon \\
<\mathit{special}> & ::= & \mathbf{n} \mid \mathbf{t} \mid \boldsymbol{\{} \mid \mathbf{[} \mid \text{\textasciigrave}
\end{array}
$$

där **anyother** betyder vilket tecken som helst utom specialtecknen.

2.1.3 Variabler

Ett programspråk som ska vara användbart behöver naturligtvis variabler eller någon slags identifierare. I Rnd finns det en typ av identifierare som används både till variabler och funktioner. Det blir enklare av den anledningen att man slipper markera på något sätt om det är en variabel eller funktion man menar.

I Rnd finns det enbart globala heltalsvariabler som allokeras dynamiskt och samma identifierare används även till tabeller/funktioner. Variabelvärdet och tabellen är nära bundna till varandra. Det innebär att indexering till tabeller och anrop till funktioner även kan sätta värdet på variabeln. Exempel: {x =1} sätter variabeln x till värdet 1, men gör inget anrop. {x 1} sätter värdet och anropar x, vilket förstås kan ge sidoeffekter.

Variabler kan även ha värdet "oändlighet" och "minus oändlighet", och de skrivs som ~ och minus oändlighet som -~. Variabler kan även vara odefinierade, vilket de är innan de har fått något värde tilldelat. Detta får särskild effekt i funktionsanrop, vilket förklaras i kapitel 2.2.1.

Namn på variabler kan vara vilka tecken som helst, men vissa tecken behöver förstås quotas för att de inte ska evalueras. Man kan alltså ha variabelnamnen \}, ` ´ eller (). Det kan bli problem med vissa namn i evaluering av aritmetiska uttryck, och de måste då quotas även där.

Variabler är egentligen tabeller, och därför förklaras syntax och semantik mer utförligt i kapitel 2.1.8.

Implementationsdetalj:
När man startar interpretatorn kan man skicka med heltalsargument som hamnar i variablerna arg1, arg2, ...argN. Exempel: Startar man Rnd med rnd -1 --2 prog så sätts variabeln arg0 till 1 och arg1 till -2.

2.1.4 Quotning

Rnd evaluerar all kod eller text som läses in. Det måste finnas en konstruktion som förhindrar detta. Det går förstås att skriva \ framför varje tecken, men detta blir otympligt. Andra programspråk brukar sätta citationstecken runt texten.

Om man vill att interpretatorn inte ska evaluera en viss text, så kan man sätta den innanför apostrofer `´. Detta är mest praktiskt i funktioner där man vill att uttrycken ska evalueras först efter anrop till funktionen. Texten kan förstås innehålla flera nivåer av quotning.

Dessutom blir det ett specialfall när interpretatorn läser in quotes och befinner sig på toppnivån. Då plockas den yttre nivån av apostrofer bort och texten skrivs ut som vanligt. Exempel: `` `abc`def´ghi´ `` skrivs ut som ``abc`def´ghi``. Annars kan man även använda backslash \ för att undvika evaluering.

Figur 2.3: BNF-grammatiken för quotning

$<\mathit{quoted}>$ ::= ` $<\mathit{text}>$ $<\mathit{quoted}>$ $<\mathit{text}>$ ´ | ϵ

2.1.5 Aritmetiska uttryck

Även om Rnd var tänkt att användas till att behandla text, så är det nödvändigt att kunna utföra aritmetik, vilket också behövs för index och indexering i tabeller. Man kunde låta Rnd vara utan aritmetiska uttryck och simulera dem med funktioner istället, men det blir för otympligt

Aritmetiska uttryck sätts innanför hakparenteser. Operatorerna är de vanliga, addition, subtraktion, multiplikation, division och mod, och de skrivs `+ - * / %`. De logiska operatorerna är större än, mindre än, lika med, bitvis "och", och bitvis "eller", och de skrivs `> < = & |`. De ger 0 för falskt och 1 för sant.

Det finns även en : operator för att hämta värden på stacken, men denna beskrivs mer utförligt i kapitel 2.2.2. Dessutom finns operatorn `$` som jag kallat "die", och den ger slumpmässiga tal precis som om man kastat tärning. Exempel: `[2$6]` ger summan av två sexsidiga tärningar.

Prioriteten för dessa operatorer, där : har högst prioritet:

1. `:`
2. `$`
3. `/ * %`
4. `+ -`
5. `< > =`
6. `&`
7. `|`

Det går förstås att använda parenteser inuti uttrycken. Syntaxen för de aritmetiska uttrycken är den normala infixnotationen som andra programspråk brukar använda.

Uttrycken kan förstås innehålla variabler också. Exempel: `[x+1]` ersätts med 2, om x har värdet 1. Variabelnamn som innehåller andra tecken än bokstäver och siffror måste quotas. Exempel: ``[(`##´+1)*2]`` där `##` är ett variabelnamn

Division med 0 ger inte fel, utan kommer att ge `~` (oändligheten) som resultat och 0 dividerat med 0 kommer att ge en tom sträng. Detta kan vara praktiskt för indexering i tabeller och man slipper en del felhantering vid division med 0. Aritmetik med oändlighet beskrivs i kapitel 6.1.

Figur 2.4: BNF-grammatiken för aritmetiska uttryck

$$
\begin{array}{lll}
<\!\mathit{arithmetic}\!> & ::= & \mathbf{[}\ <\!\mathit{term}\!>\ <\!\mathit{expr}\!>\ <\!\mathit{term}\!>\ \mathbf{]} \mid \epsilon \\
<\!\mathit{term}\!> & ::= & <\!\mathit{chars}\!>\ <\!\mathit{term}\!> \mid <\!\mathit{chars}\!>
\end{array}
$$

2.1.6 Datatyper

Det finns alltså bara två sorters datatyper i Rnd, heltal och strängar. En variabel (tabell) kan bara ha heltalsvärde och innehållet i entry kan bara vara strängar. Heltalen har värden från -MAXINT till MAXINT, där MAXINT beror på implementationen. Konvertering mellan heltal och sträng sker automatiskt eftersom heltalen skrivs ut som strängar när man hämtar dem.

2.1.7 Scanning

Vanliga programspråk gör en eller två genomläsningar, makroexpansioner och liknande, av koden som den kompilerar eller interpreterar. Rnd använder sig av en lite speciell variant, som kan kallas flerpassubstituering, när den läser in (scannar) koden. Det liknar någon slags termomskrivning [DJ90], eller den reduceringsmetod som lambdakalkyl använder. Skillnaden är att Rnd:s metod följer inte ett strikt schema, där enbart grammatiskt korrekta uttryck accepteras och reduceras till nya korrekta uttryck, utan läser in och evaluerar efter hand på ett friare sätt.

Rnd läser in programmet tecken för tecken och när den har läst in ett fullständigt subuttryck, utan inre subuttryck, så evalueras det och resultatet stoppas in där subuttrycket stod. Exempel: `a{x}c` kan bli `ayc` eller `{{foo}` kan bli `{bar}` efter ett pass. När det är klart börjar interpretatorn att scanna från början av programmet igen, men den kan skriva ut och sedan ignorera text som inte behöver scannas mer. Den fortsätter med scanningen så länge det finns tecken kvar att läsa. Interpretatorn exekverar alltså samtidigt som den läser programmet.

Ett program ser ut så här: `abc{x =[2+2]}def`. När scanningen börjar står buffertpekaren först i programmet.

```
abc{x =[2+2]}def
↑
```

Tecknen `a` `b` och `c` läses in och skrivs ut och försvinner från bufferten.

```
{x =[2+2]}def
↑
```

Nu pekar den på ett tabell-subuttryck, men fortsätter läsa och skriver inte ut något.

```
{x =[2+2]}def
    ↑
```

Ett aritmetiskt subuttryck påträffas och interpretatorn slutar följa det första tabell-subuttrycket och följer istället det aritmetiska.

```
{x =[2+2]}def
        ↑
```

En avslutande hakparentes innebär att det aritmetiska subuttrycket är klart att evalueras. Resultatet stoppas in i bufferten och scanningen börjar om igen från början.

```
{x =4}def
↑
```

Subuttrycket `{x =4}` påträffas och x får värdet 4. När man använder sig av likhetstecken = i tabellanrop så evalueras uttrycket till en tom sträng.

```
def
↑
```

Slutligen skrivs `d` `e` och `f` ut.

Exempel 2: Scanning

2.1.8 Tabeller

Det sista som behövs nu är funktioner och ett sätt att uttrycka dem. Någon form av funktioner är nödvändigt i ett programspråk, eller någon form av "labels" eller radnummer. Rnd har istället tabeller som fungerar som funktioner. För att visa hur en tabell ser ut kommer här ett exempel på en annan slags tabell.

Svenska Kungar	
1905-1950	Gustav V
1950-1973	Gustav VI Adolf
1973-	Carl XVI Gustaf

Om man nu undrar vem som var kung 1940 slår man upp det i tabellen och får "Gustav V" som svar. När det står "1905-1950" i tabellen betyder det att alla år mellan 1905 och 1950 var han kung, så även 1940. Man har så att säga klumpat ihop de åren till en rad. Denna tabelluppslagning kan ses som anrop till funktionen "Svenska Kungar" med parametern 1940. Rnd:s funktioner är konstruerade på samma sätt, och kallas därför tabeller.

En tabell i Rnd innehåller en lista med par av index och entry. Indexen måste vara heltal men entry kan vara vilken text som helst. Nummerordningen på paren index-entry har ingen betydelse. Man deklarerar tabellen genom att skriva tabellens namn och listan innanför måsvingar `{}`, och separerar namnet, indexen och entryn med blanksteg, tab eller nyrad. Formellt sett blir det `{`*`namn index entry index entry ...`*`}`

Exempel: `{foo 1 red 2 blue 3 green}`, där tabellens namn är foo och på index 1 hamnar `red`, index 2 hamnar `blue` och så vidare. Tabellen är nu deklarerad och blir förstås kvar under resten av evalueringen av programmet.

Om man sist i listan har ett index utan tillhörande entry tolkas det som en indexering (eller anrop) till tabellen, och det entry som ligger på det indexet hämtas och evalueras på nytt. Exempel: Subuttrycket `{foo 2}` ersätts med `red`, som den sedan evaluerar, eller skrivs ut i det här fallet. Samtidigt sätts variabeln foo till 2, som beskrevs i kapitel 2.1.3 om variabler.

Om ett visst index saknar entry när man anropar tabellen så kan en av två saker hända. Antingen får man tillbaka en tom sträng om indexet är större än det största indexet i tabellen, eller så får man tillbaka det entry som har ett index som är större eller lika med anropsindexet. Indexen är alltså sorterade i nummerordning så att rätt index kan slås upp. Exempel: `{foo 4}` ger en tom sträng, men `{foo 0}` ger tillbaka `red` eftersom 1 är större än 0.

En tabell påminner om en array, speciellt en associativ array eller gles array. En array som har värdena `a,b,b,b,c,c,d` kan alltså skrivas i Rnd som `{x 0 a 3 b 5 c 6 d}` och sedan indexeras med `{x `*`index`*`}`.

Man kan även sätta ett entry med index "oändligheten", som representeras med ett tilde `~`. Det blir ett index som så att säga ligger längst ut i änden på listan. Exempel: Om man gör `{foo ~ the_end}` och anropar med `{foo 4}` får man tillbaka `the_end` och det får man även med `{foo 99}`, `{foo 4711}` eller vilket index som helst.

För att visa hur tabeller fungerar kommer ett lite större exempel. Så här ser input-buffern ut vid start:

```
{foo 1 fo 10 `{{foo 1}o [foo+10]}' ~ `foo = [foo]' 5}
```

Efter första läsningen i buffern så hittar den ett subuttryck, en tabell med 3 par och ett anrop med argumentet 5. Det finns inga andra subuttryck inuti, utom quotade, så hela uttrycket evalueras. Index `1` sätts till `fo`, index `10` till strängen `{{foo 1}o [foo+10]}` och index oändligheten till `foo = [foo]`. Inget mer evalueras just nu.

Tabellen `foo` anropas nu med argumentet 5, vilket innebär att innehållet på index `10`, hämtas ut och placeras i buffern, och variabeln `foo` sätts till 5. Nu ser bufferten ut så här:

```
{{foo 1}o [foo+10]}
```

Interpretatorn börjar om från början av buffern. Där hittar den ett inre subuttryck `{foo 1}`, vilket evalueras först. Den anropar `foo` med argumentet 1, alltså hämtar innehållet på index `1`, som är `fo`, och stoppar in det i buffern, och sätter variabeln `foo` till 1.

```
{foo [foo+10]}
```

Nu hittar den ett inre aritmetiskt uttryck, och evaluerar det. Variabeln foo har värdet 1 och resultatet blir 11.

```
{foo 11}
```

Variabeln foo sätts till 11, och innehållet på index ~ (oändligheten) stoppas in i buffern.

```
foo = [foo]
```

`foo =` skrivs ut och det aritmetiska subuttrycket evalueras. I buffern står nu:

```
11
```

`11` skrivs ut och interpretatorn slutar exekvera eftersom input-buffern är tom.

Exempel 3: Exekvering av tabell

Om man bara har en typ av anrop får man inte tillräcklig flexibilitet. Det är inte alltid man vill sätta variabeln samtidigt som man anropar tabellen, eller man vill inte anropa och enbart sätta variabeln. Den senare typen har redan beskrivits i kapitel 2.1.3 om variabler.

En annan viktig anropstyp är då man vill anropa men inte evaluera entryt. Man vill alltså quota resultatet. Detta är en nödvändighet i det här språket som har implicit evaluering. Det måste därför finnas en spärr mot evaluering.

Det finns alltså tre andra typer av anrop. Om man sätter ett särskilt tecken, en modifierare, framför indexet får man andra effekter. Det finns även två andra modifierare som inte har med anrop att göra och dessa beskrivs mer i detalj längre fram. Dessa modifierare är:

`=` variabeln sätts till indexerade värdet och inget entry hämtas. Detta är speciellt användbart då man vill sätta en variabel eller om tabellen ska initieras med ett startvärde. Exempel: `{foo =1}` sätter variabeln `foo` till 1.

`*` variabeln sätts inte, men entryt hämtas. Detta används i samband med parametrar. Exempel: `{foo *5}` ger `fo` som resultat men variabeln har fortfarande sitt gamla värde kvar.

`"` entryt hämtas men det quotas, (sätts inom apostrofer). Detta behövs i de fall man vill förhindra evaluering. Exempel: `{foo "6}` ger `` `fo´ `` som resultat och `foo` sätts till 6.

`#` markerar att detta är ett index och inte ett entry.

`:` markerar att detta är ett entry och inte ett index.

Man kan kombinera dessa, men `"=` blir samma som `=`, och `*=` blir samma sak som att inte ha någon modifierare alls. En annan fördel med att använda dessa modifierare är att man kan använda dem inne i listan. Exempel: `{foo =1 1 black}`. Detta beskrivs närmare i kapitel 2.2.2 om parametrar.

Om variabeln är satt och man inte anger något anropsindex, så kommer tabellen ändå att anropas med variabelns nuvarande värde. Om man skriver enbart `{foo}` i det större exemplet ovan kommer index 11 att hämtas igen och evalueras. Men om variabelvärdet på foo varit odefinierat så hade inget anrop skett.

Man kan förstås definiera fler index och entrys i en befintlig tabell. Om man lägger till ett nytt entry på ett befintligt index så kastas det gamla bort.

Om man vill ange att några tecken skall vara ett index kan man sätta en `#` framför indexet. Om man vill markera att några tecken skall vara ett entry, ett vanligt problem är siffror, sätter man ett kolon `:` framför tecknen.

`#` är egentligen inte nödvändigt att ha med eftersom det räcker med `:` för att markera ett entry. Det kan dock behövas i framtiden när index kan vara vilka tecken som helst. Exempel: `{foo #4 :yellow}`

Man kan även utelämna indexet när man sätter ett entry. Då tas det senaste index-numret, ökar med 1, och lägger entryt på den positionen. Om man inte har angett något index tidigare så tas index 0. I exemplet tidigare kunde man skrivit `{foo 1 red blue green}` och då skulle `blue` hamna på index 2 och `green` på 3.

Designen av syntax och semantik för tabeller kan tyckas vara märklig. Framför allt att man kan ha olika ordning på index, entry och anrop. Alternativet är att ha särskilda skiljetecken mellan dem, som till exempel `{foo,4-yellow;1}`, men det skulle bli mer svårläst, strikt och inflexibelt. Med den syntaxen kan man inte detaljstyra evaluering inuti listan heller.

Figur 2.5: Tabellers syntax skriven i BNF-grammatik

<table>	::=	**{** *<identifier>* *<list>* **}**
<list>	::=	*<entry>* *<list>*
	|	*<index>* *<entry>* *<list>*
	|	*<call>*
	|	ϵ
<call>	::=	*<modifier>* *<index>*
<index>	::=	`~` | `-~` | *<index>*
<entry>	::=	`:` *<text>* | *<text>*
<modifier>	::=	= | `*` | `"` | # | ϵ

2.2 Övriga konstruktioner

Föregående kapitel beskrev hela språket. Det som saknas i beskrivningen är metoder för att hantera typiska konstruktioner som förekommer i andra språk.

2.2.1 Funktioner

Det finns ingen särskild syntax för att deklarera funktioner. En funktion definierar man helt enkelt genom att lägga koden i entryn i en tabell. Indexen blir då "ingångar" till funktionen, men det brukar vara lämpligt att lägga koden på index `~`. Då kan man anropa funktionen med vilka argument som helst. Koden bör förstås vara quotad så att den inte evalueras.

Exempel: ``{printx ~ `[x]\n´}``, definierar en funktion `printx` som kommer att skriva ut innehållet i x för alla argument. Intressantare blir funktionen ``{print ~ `[print]\n´}`` som skriver ut argumentet till funktionen.

Anropar man med `{print 3}` så skrivs 3 ut. Intressantast är funktionen `printvar` i kapitel 2.2.2.

Att man kan lägga till andra ingångar till funktionen kan vara praktiskt när man definierar rekursiva funktioner. För fakultetsfunktionen kan man lägga ett index på 1 som returnerar 1, och ett index på ~ som returnerar fakultetsberäkningen.

```
{fac
  1 :1
  ~ '[[fac]*{fac [fac-1]}]'
}
```

Exempel 4: Fakultetsfunktionen

Lägg märke till att konstruktionen `` `[fac*{fac [fac-1]}]´ `` inte hade fungerat. Evaluering av det inre subuttrycket sker före multiplikationen, och `fac` sätts till nytt värde, innan det aritmetiska uttrycket beräknas. `fac` måste alltså stå innanför hakparenteser.

2.2.2 Parametrar

Det finns två sätt att skicka parametrar till funktioner. Det första sättet måste man använda om man vill skicka strängar eller kod som argument. Man sätter ett entry i tabellen på index 0, som man sen använder som parameter. Vill man skicka flera argument så kan man sätta dem på index 0, 1, 2 ... etc.

```
{printvar ~ '{printvar "*0} = [{printvar *0}]' =~}
{y =42}
{printvar y}
```

Argumentet `y` läggs på index 0 och som resultat skrivs 42 ut.

Exempel 5: En funktion som skriver ut innehållet i en variabel

Det andra sättet är lite enklare att använda. Man tilldelar funktionen ett antal värden i rad. Exempel: `{x =1 =2 =3}`, och dessa kan sedan hämtas i aritmetiska uttryck med `:` operatorn. `x:0` refererar till det senaste värdet, `x:1` till det nästa senaste, och så vidare.

Dessa värden lagras undan i en stack, och ligger kvar där tills man man gör en ny tabell-tilldelning. I exemplet ovan så kastas de gamla värdena bort om man gör `{x =0}`.

2.2.3 Villkors-satser

Det finns ingen styrsats för if-konstruktionen. Istället måste man skapa en tabell som simulerar if-satsen. Ett aritmetiskt uttryck som innehåller ett logiskt uttryck evalueras till 0 eller 1. På de indexen i en tabell kan man då lägga kod för “then-gren” och “else-gren”.

Exempel: `"if (a>b) then max=a else max=b"` kan skrivas som ``{if `{max =[a]}´ `{max =[b]}´ [a<b]}``. En annan variant är att skriva `{{if a b}{max =[{if *[a<b]}]}}`, vilket kan vara praktiskt om båda grenarna har likartad kod. Man behöver förstås inte använda "if" som tabellnamn, men det blir lättare att läsa. En tredje variant är `{max ={if [a] \[a]    \[if] [b]}}`, vilken inte innehåller något villkor. Denna variant är inte så bra eftersom entryt stannar kvar i tabellen `if`, men det visar hur pass implicit Rnd är. Det borde dock finnas ett sätt att göra temporära tabeller.

En case-sats konstruerar man på liknande sätt, men man får se upp med indexen för anrop till tabeller som faller igenom till närmast högre index.

2.2.4 Loopar

Styrsatser för loopar saknas också. Man måste konstruera dem med hjälp av rekursion. Fakultetsfunktionen i exempel 4 är ett exempel på detta.

Anta att vi vill ha en loop som skriver ut `i=[i]` tio gånger. Funktionen som gör loopen måste då skriva ut `i=[i]`, räkna upp i med 1, och kolla om i är mindre än 10, och i så fall köra loopen ett varv till. Följande kod gör detta:

```
{i 0 ‘i=[i]\n’
   ~ ‘{i *0}{i =[i+1]}{i *[i<10]}’
1}
```

Det här kan man generalisera till en loop-konstruktor som tillverkar en sådan loop, och till konstruktorn skickar man loopvariabel, start, stopp och kod som parametrar.

```
{loop
~ ‘{{loop *0}
    0 {loop *"3}
    ~ \{{loop *0}‘ *0}
{’{loop *0}‘ =[’{loop *0}‘+1]}
{’{loop *0}‘ *[’{loop *0}\<{loop *2}‘]}’
```

```
 {loop *1}}'
=~}
```

Det här visar även hur man hämtar parametrar, som i detta fall är `loop *0`, `loop *1`, och så vidare.

Nu kan man göra en loop genom att skriva `{loop i :1 :10 'i=[i]\n'}` och få samma resultat. Man kunde förstås ha skickat parametrarna 1 och 10 som `=1 =10`, men då skulle konstruktorn se rörigare ut. Den här loopen måste ha start och stopp större än 1, och fungerar inte för negativa parametrar.

Man kan alltså göra funktioner som tillverkar loopar, och även andra typer av styrsatser. Se vidare i kapitel 3.6.7 om funktionskonstruktion.

2.2.5 Saknas

I Rnd saknas många grundläggande funktioner som flyttal, stränghantering, eller pipe- och filhantering. Det är inte riktigt användbart som det ser ut nu utan är mer ett teoretiskt språk. I kapitel 5 om utvidgningar och kapitel 6.1 om begränsningar beskrivs dessa mer utförligt.

2.3 Ett större programexempel

För att visa hur man programmerar i Rnd följer här ett större programexempel. Det visar hur man bygger konstruktorer och en datastruktur. Det visar även hur man lägger kod i datastrukturer, och hur praktiskt detta kan vara. Beskrivning av funktionerna följer efter koden.

```
{assign ~ '{{assign *0} ~ {assign "*1} =~}' =~}

{loop
~ '{{loop *0}
    0 {loop *"3}
    ~ \{{loop *0}' *0}
{'{loop *0}' =['{loop *0}'+1]}
{'{loop *0}' *['{loop *0}\<{loop *2}']}'
 {loop *1}}'
=~}

{assign name :0}
{assign job :1}
{assign where :2}
```

```
{assign salary :3}
{assign company-name IDA}
{assign xjobb-sal :12000}

{make-person ~ `
{nrp =[nrp+1]}
{person[nrp] {name} {make-person "*0}
             {job}  {make-person "*1}
             {where} {make-person "*2}
             {salary} {make-person "*3}
}
{assign {make-person "*0} person[nrp]}'
=~}

{write ~ `
Name:   {{write *0} *{name}}\n
Job:    {{write *0} *{job}}\n
Where:  {{write *0} *{where}}\n
Salary: {{write *0} *{salary}}\n\n'
=~}

{make-person Mats xjobb `{company-name}' :15000}
{make-person Johan dokt `{company-name}'
`[{{Mats} *{salary}}+1000]'}

---alla personer\n
{loop i :1 :[nrp-1] `{write person[i]}'}

---uppdatera\n
{{Mats} {salary} `{xjobb-sal}'}
{write {Johan}}

{eqx ~ `
{{eqx *0} =1}
{{eqx *1} =0}
[{eqx "*0}={eqx "*1}]
{{eqx *0} =0}
' =~}
{eq ~ `{eqx \!{eq *0} \!{eq *1}}' =~}

Har {if `olika' `samma'
{eq :{{Mats} {salary}} :{{Johan} {salary}}}} l|n\n
Har {if {eq :{{Mats} {where}} :{{Johan} {where}}}} arbetsplatser\n
```

En körning av detta program ger följande utskrift:

```
---alla personer
Name:   Mats
Job:    xjobb
Where:  IDA
Salary: 15000

Name:   Johan
Job:    dokt
Where:  IDA
Salary: 16000

---uppdatera
Name:   Johan
Job:    dokt
Where:  IDA
Salary: 13000

Har olika l|n
Har samma arbetsplatser
```

För att kunna förklara vad som händer kommer först en beskrivning av de olika funktionerna i exemplet ovan:

`assign` Den här funktionen tar två parametrar, dels variabelnamnet och dels vad det ska tilldelas. Den tillverkar allstå en tabell som har entryt på index ˜ och där kan man lagra strängar. Man hämtar sedan värdet med `{variabelnamn}`. Man kan likna den vid Pascals ":=" operator.

`loop` Detta är loop-konstruktorn som beskrivits tidigare.

`make-person` Denna funktion konstruerar en person-post som innehåller fyra `fält, name, job, where, salary`. Den tillverkar också en referenstabell som har samma namn som name-fältet, som senare används istället för `person000`, så att man kan referera posten med namnet.

`write` Skriver ut innehållet i en post.

`eq` Jämför två strängar och returnera 1 om de är lika. Den är egentligen inte bra, men eftersom det inte finns stränghantering blir den så här ful.

Det som händer är att två personer skapas och "Mats" arbetsplats sätts till `'{company-name}'` vilket innebär att när man hämta värdet på det fältet hämtas innehållet i `company-name` istället. Sen sätts "Johan's" lön till "Mats" lön plus 1000 kronor. Det fältet pekar alltså ut “Johan’s” fält. Han kommer alltid att 1000 mer i lön.

Sen ändras “Mats” lön och efter det skrivs effekten av detta ut. Det går förstås att sätta lönen till vilket uttryck som helst.

I Rnd behövs inte pekare eller referenser. Man skriver in programkod rakt av i variablernas värden och sedan evalueras de. Detta gör det hela väldigt flexibelt och tyvärr långsammare. Exemplet visar också lite på att det kan bli ganska kraftfullt efter att man definierat upp sina tabeller.

Kapitel 3

Jämförelser med andra liknande språk

För att visa Rnd:s flexibilitet när det gäller evalueringar, så har jag valt ut några andra interpreterande språk att jämföra med. Rnd liknar mest tcl och m4, men jag tog med perl för att det har ganska kraftfull evaluering. Ett vanligt språk som LISP har också tagits med för att det i viss mån liknar Rnd, men att jämföra med något imperativt språk som C blir svårt eftersom det saknar eval-funktioner, och det är ett kompilerande språk. Man kan skriva koden till fil kompilera den och köra, men då får man inte med hela omgivningen. LPC är ett C-liknande språk och har en del interpreterande egenskaper, så det har tagits med i en del jämförelser.

Andra språk som man kunde jämföra med är ML och Erlang. De har en del likheter med Rnd med avseende på funktioner. De kan ha olika "ingångar" för funktionerna och detta gäller speciellt Erlang. Man kunde även jämföra med makrospråket cpp men det språket är mindre kraftfullt än m4.

Nya språk som till exempel Java saknar precis som C möjligheten att evaluera. Det saknar även makroprocessor, så det blir ännu ointressantare att jämföra med. Java är ur evalueringssynvinkel ett steg bakåt. Det har ingen ny intressant funktionalitet.

Egentligen kan man simulera en evalueringsmekanism i alla normala programspråk, men då blir det ett språk i språket, en slags eval-funktion som blir ett mellanting.

Nu följer en kort översikt av språken. En viss kännedom om språken förutsätts.

3.1 m4

m4 är en enkel makroprocessor som används som preprocessor till andra programspråk. Den använder nästan samma princip som Rnd vad det gäller scanning av programkoden. Skillnaden är att den inte skriver ut vanlig text direkt utan skriver tillbaka texten i inputbuffern.

Precis som Rnd är ett m4-program inte syntaktiskt korrekt om man betraktar hela programmet. Det är under exekveringen som interpretatorn upptäcker om programmet är korrekt.

m4 har egentligen inte variabler, utan dessa får simuleras med funktioner. En funktion deklareras med `define(f,‘text’)` och när ett `f` påträffas ensamt i resten av programmet ersätts det med `text`. Det finns en del fördefinierade funktioner som till exempel `ifelse` och `eval`.

Nackdelen med m4 är att man inte kan konkatenera ihop variabelnamn eller funktionsnamn. Till exempel:

```
define(a,‘oo’)
define(foo,‘hej’)
f‘’a
```

Borde skriva ut `hej`, men skriver istället `foo`. En annan nackdel med m4 är att det saknar arrayer eller någon form av listhantering. Men rekursion är däremot möjlig, till skillnad mot cpp.

För programspråk har man ofta en omgivning (maskintyp, version, funktionsbibliotek etc.) som påverkar hur programmet ska kompileras och det är just för sådana användningsområden som makroprocessorer är kraftfulla.

3.2 Tcl

Tcl är ett programspråk som används tillsammans med fönsterverktyget Tk för att skriva scripts. Det används ofta som ett generellt scriptspråk istället för sh och perl, och anledningen är nog dess enkelhet och praktiska evaluering. Det är ett interpreterande språk som scannar och evaluerar koden endast en gång. Man kan dock få den att evaluera flera gånger med eval kommandot.

Normalt gör tcl tre olika slags substitutioner: för variabler, för kommandon och backslash. Variabler ersätts med sitt innehåll, på samma sätt som för perl. Men `set x "puts {foo}"; $x`, som borde skriva ut `foo`, fungerar inte eftersom koden bara scannas en gång. Om man vill att kommandon skall evalueras vid första scanningen så kan man sätta dem innanför hakparenteser. Ett typiskt exempel är `set i [expr $i+1]`. Begreppet backslashsubstitution är helt enkelt att sätta \ framför tecken som inte ska tolkas, `\$x` ersätts alltså med `$x`

Det har visat sig att nybörjare på tcl har trott att evalueringen i scanningen är kraftfullare än vad den egentligen är, och att den skulle fungera på samma sätt som Rnd.

Om man vill att tcl ska evaluera flera gånger använder man kommandot `eval`. Exemplet ovan funkar bättre med `set x "puts {foo}"; eval $x` vilket skriver ut "foo". Man kan även få den att rekursera i scanningen med till exempel:

```
set i 10
set f {if {$i>0} then {puts $i; set i [expr $i-1]; eval $f}}
eval $f
```

En annan trevlig sak med tcl är att man kan definiera sina egna kommandon, som då fungerar på samma sätt som de andra. Man kan alltså göra egna typer av styrsatser.

3.3 Perl

Perl är egentligen en kombination av de bästa bitarna ur C, sed, awk och sh. Det är väldigt användbart till att läsa textfiler och extrahera information ur dem. Det finns ingen egentlig kompilator till perl, utan det är ett interpreterande språk, eller snarare semikompilerande.

Perl används ofta vid systemarbete för att snabbt göra små scripts som hanterar och bearbetar textfiler. Det är tyvärr inte så lämpligt att använda i större programsystem eftersom det är långsammare än kompilerad kod, och har en tendens att använda mycket minne.

Perl har egentligen två evalueringssätt. Det ena är när man skriver ut strängar eller tilldelar strängar ett värde. Ett exempel är `print "x=$x"`, här stoppas innehållet i `$x` in i strängen och skrivs ut. En variant på detta är att man som namn på en variabel, kan ge ett uttryck som kan evalueras innanför måsvingar. `print ${"f"."oo"};` är alltså samma sak som `print $foo;` De här evalueringarna sker dock bara i ett pass, eller snarare evalueras en gång, men uttrycken kan vara ganska komplexa. Exempel: `print "$a[1+1]";` kommer att addera och skriva ut element 2 i `$a`.

Det andra sättet är att använda funktionen `eval`. I perl version 4 var den begränsad till en nivås evaluering. I perl5 kan man göra eval på eval med godtyckligt djup. Nackdelen med den här sortens eval är att man för att få tillräcklig funktionalitet måste göra eval på alla tilldelningar och beräkningar. Dessutom kan man inte skicka argument till eval.

3.4 LISP

Lisp är ett vanligt språk inom AI och datavetenskap. Att det har samma representation för både data och kod gör att det blir enkelt men samtidigt väldigt kraftfullt. Det har flera likheter med Rnd. Man evaluerar uttryck och har rekursion som kontrollstruktur, i motsats till imperativa språk där man exekverar program och loopar.

Lisp, liksom perl och många andra språk, accepterar bara syntaktiskt korrekta program. Det sker alltså ingen preprocessning av uttrycken, undantaget är `'` som ersätts med `(quote` och i en del Lisp-dialekter ersätts `]` med ett antal parenteser.

Även om kod och data kan ha samma representation i Lisp, och hela Lisp är en enda stor evaluator, så kan det inte evaluera partiella uttryck.

I Lisp finns förstås eval-funktionen, men den är lika begränsad som andra eval. Dessutom kan man definiera makron som är ett sätt att simulera substituering. Makron är det närmaste man kan komma Rnd:s evaluering.

Lisp har arrayer och något som kallas associationslistor, eller tabeller. Det är en lista med par av nycklar och attribut. De är ganska lika Rnd:s tabeller.

3.5 LPC

LPC är ett objektorienterat språk som liknar C, [PH90]. Det utvecklades för att användas i MUD-system, ett slags fleranvändarspel där spelarna i en fiktiv värld kan bygga egna rum och objekt. För just detta språk kommer bara dess egenskap att omvandla strängar till funktionsanrop att undersökas.

3.6 Jämförelser

För att visa några av Rnd:s fördelar kan det vara bra med några jämförelser med andra språk. Exemplen är ganska konstruerade och det beror delvis på att Rnd saknar vissa saker, som är nämnda i kapitlet om utvidgningar. Alla språken kommer inte att jämföras i alla exemplen.

Egenskaper som kommer att undersökas:

1. sträng $\rightarrow$ funktionsanrop
2. eval-funktionen
3. implicit/explicit evaluering

4. direktexekvering
5. omprogrammering

3.6.1 Implicita villkor

Ett väldigt enkelt exempel på implicita villkor är i små funktioner som till exempel abs-funktionen. Den skrivs i Rnd som: `{abs `[-abs]´ ~ `[abs]´}` och har då varken villkor eller if-sats.

Det kanske ser ut som om detta skulle vara otydligare och mer svårläst, men det har mer att göra med sättet att programmera. Det kan bli mindre att skriva och i vissa fall enklare.

Motsvarande kod i perl blir:

```
sub abs { return $_[0]>0?$_[0]:-$_[0]; }|
```

Vilket blir mer svårläst. Det blir det även i m4 där samma funktion ser ut så här med ifelse-sats.

```
define(abs,‘ifelse(eval($1<0),0,$1,eval(-$1))’)
```

Lisp och tcl har inte heller implicita villkor, men i de flesta språk går vissa villkor att simulera med arrayer. Till exempel i perl: `eval($case[$x])`.

3.6.2 Stora arrayer

I Rnd kan man rent teoretiskt ha oändliga arrayer. Få programspråk har denna funktionalitet och det är egentligen bara dataflödes-språk, som Lucid [Wad85], som har något liknande.

Man kan fylla stora delar av en array i Rnd med ett värde, genom att sätta index och entryn i en tabell med lämpliga värden. Ett exempel: `{x 1 a 100 b}` kommer att ha ett `a` i position 1 och `b` i positionerna 2 - 100. Positionerna $-\infty$ till 0 blir förstås även dem `a`. Att lägga till ett `c` i positionerna 2 - 50 är lätt: `{x 50 c}`

I perl kan man göra motsvarande genom att sätta alla element för sig. Det blir som vanligt i perl ganska kort:

```
$x[1] = "a";
@x[2..100] = ("b") x 99;
```

I tcl och Lisp får man skriva en loop som sätter värdena från 2 till 100. Fast man kan ju initiera arrayen till `b` i Lisp när man deklarerar den och sedan sätta första elementet till `a`, men mer flexibilitet får man inte. m4 saknar arrayer, men man kan simulera det med funktioner, vilket blir väldigt otympligt.

Man kan lägga in funktionsanrop i Rnd:s arrayer, och det är bara perl som kan simulera motsvarande med tie-konstruktionen. Annars kan man förstås göra `eval` varje gång man hämtar något i arrayen.

Det praktiska med denna typ av arrayer är att de tar mycket mindre minne, och går snabbare att initialisera.

3.6.3 Implicit/explicit evaluering

I Rnd görs implicit evaluering på all data som hämtas, tilldelas eller flyttas. Detta är i motsats till perl, tcl och många andra där man måste göra explicit evaluering.

För att visa effekten och det praktiska med detta ges ett enkelt exempel. Antag att variablerna x och y innehåller värden vilka som helst. Det kan vara en konstant eller ett subuttryck eller kombinationer av detta. Man vill göra två saker med variabeln: 1) Enbart tilldela variabeln z värdet av x, och inte evaluera. 2) Evaluera x och tilldela variabeln z resultatet.

Antag sedan att x innehåller värdet `{y}` och y har värdet `[a+b]`. I tcl blir det `$y` och `expr a+b`. I perl blir det `$y` och `$a+$b`.

Tilldelningarna av variabeln z gör man på följande sätt i Rnd. Om man har en tabell som en strängvariabel är det praktiskt att lägga värdet på oändligheten. Den första är utan evaluering (med ") och den andra är med evaluering.

```
{z ~ {x "~} =~}
{z ~ {x} =~}
```

Det finns en lat variant, se kapitel 3.6.6, där värdet av x inte hämtas förrän z används:

```
{z ~ `{x}' =~}
```

I tcl blir motsvarande:

```
set z $x
set z [eval eval $x]
```

Men problem uppkommer om man sätter `$y` till `"$t"` och `$t` innehåller `"expr 2+3"`. Då måste man göra tre eval för att få fram resultatet.

Ett sätt att lösa det är att sätta `$x` till `"eval $y"`, och då vet värdet själv när det skall evalueras. Men då får man problemet vid print att texten `eval` skrivs ut.

I perl gör man något liknande med dess eval-funktion. m4 har implicit eval precis som Rnd, och man kan hämta definitionen med `defn`.

```
define(x,y)
define(y,`eval(a+b)')
define(z,defn(`x'))
define(z,x)
```

Detta beskrivs även i kapitel 4.2.

3.6.4 Filter

I Unix kan man koppla ihop flera program med hjälp av pipes. Mellan dessa program brukar man för det mesta skicka data, men ibland kan man skicka programkod. Den koden evaluerar eller kompilerar man sedan, men det kan vara knepigt om man vill skicka kod och data blandat. Om man sedan har väldigt stora data eller program som sänder dessa under lång tid, kan det vara svårt att evaluera. I perl måste man skicka hela koden eller fullständiga programsatser på varje rad som skickas. Ett exempel på ett perl-program som läser in kod och exekverar den:

```
while (<>) {
  eval $_;
}
```

Man kan förstås göra ``eval `otherprog`;`` men då försöker den läsa in hela programmet innan den börjar evaluera. Någonting motsvarande kan man göra i tcl, men med samma nackdelar.

I Rnd behöver man inte skriva någonting alls eftersom allting som tas emot evalueras direkt och skickas vidare på `STDOUT`. m4 fungerar på liknande sätt. Med Rnd kan man då koppla ihop flera processer som kontinuerligt evaluerar sina indata.

3.6.5 Praktiskt

Rnd:s evalueringsprincip gör det enkelt och praktiskt att ändra i koden. Man slipper hålla reda på om data behöver evalueras eller inte. Normalt evalueras ju allt, och om man inte vill det måste man specificera det. Ett exempel är om man har en strängvariabel för felmeddelanden:

```
{felmed ~ 'Ett fel har uppstatt.' =~}
```

Om man då vill lägga in en felkod i strängen är det enkelt att stoppa in den i koden.

```
{felmed ~ 'Fel nummer [e] har uppstatt.' =~}
```

Man slipper också stoppa in eval på andra lämpliga ställen i koden. I de andra språken måste man göra `eval { print $felmed }` eller något liknande.

Rnd gör alltså ännu mindre skillnad på kod och data jämfört med de andra språken. I andra programspråk kan man nästan alltid peka ut kod och data i programmet, och de blandas inte. En sträng är inte kod, och vice versa.

3.6.6 Lat evaluering

Lat evaluering innebär att man inte evaluerar ett uttryck förrän man behöver det, och max en gång. "Call-by-need" är ett exempel på detta. Om man har ett funktionsanrop med en parameter så evalueras denna normalt före anropet, men inte med lat evaluering. Man tjänar tid men förlorar om programmet har sidoeffekter, det vill säga parametern får olika värden beroende på när den evalueras. Med lat evaluering är det också möjligt att göra "if-satser" och liknande kontrollstrukturer.

I Rnd kan man skriva:

```
{x ~ 'foo' =~}
{y ~ '{x}' =~}
```

där man då får lat evaluering på `y`. Man kan till och med tilldela det lata värdet till `z` genom att hämta det quotade värdet.

```
{z ~ {y "~} =~}
```

Men Rnd har inte äkta lat evaluering, utan man måste specificera när det skall ske, som i exemplet med `z` ovan. I annat fall får man "call-by-name", då parametrarna evalueras varje gång de refereras. Ta till exempel en funktion skriven i perl som tar tre parametrar och returnerar den andra om den första är mindre än 1, annars den tredje:

```
sub foo {
  if ($_[0]<=0) {     # $_[0] är första parametern
    return $_[1];
```

```
  } else {
    return $_[2];
  }
}
```

Om man anropar med `$x = foo(1,2+3,abs(-10))` evalueras argumenten normalt före anrop. Om perl hade haft lat evaluering hade bara ett av dem evaluerats och bara när det gjorde return. Eller kanske inte ens då. `$x` värde är kanske inte ens evaluerat efter anropet. Först när man gör till exempel `print $x` evalueras det.

Motsvarande kod i Rnd blir

```
{foo ~ ‘{if :{foo *"1} ~ :{foo *"2} {foo *0}}’ =~}
```

Den här Rnd koden evaluerar inte sina parametrar förrän man har hämtat dem. `{foo :0 ` + "`" + `[1+2]´ ` + "`" + `[2*3]´}` ger alltså `[1+2]` som returvärde, och detta evalueras direkt efter anropet. Man kan hämta värdet quotat med `"{foo *0}` vilket ger lat evaluering i följande exempel:

```
{foo ~ ‘{if :{foo *"1} ~ :{foo *"2} "{foo *0}}’ =~}
{x ~ {foo :0 ‘[1+2]’ ‘[2*3]’} =~}
{x}
```

`x` får då värdet `[1+2]` och det evalueras när det hämtas på sista raden i exemplet ovan.

3.6.7 Funktionskonstruktorer

Ett exempel med loop-konstruktorn har visats tidigare. I programspråk med eval, såsom perl och tcl är detta ganska lätt att göra. Samma sak är det för språk med makron. Det går att göra en konstruktor för att deklarera funktioner i Rnd. Den koden blir ganska rörig och tas inte med här, men då kan man deklarera en funktion `foo` med tre parametrar `a, b, c` så här:

```
{func foo :3 a b c `kod´}
```

3.6.8 Funktionell programmering

Rnd har vissa likheter med funktionella programspråk och ett antal jämförelser med dessa språken borde göras.

Kapitel 4

Teori

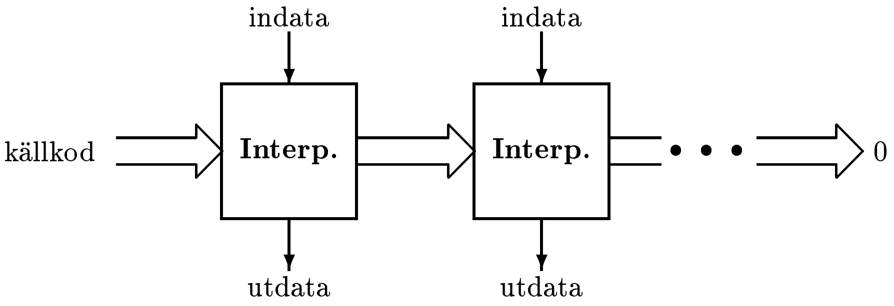
Det här kapitlet tar upp lite teori och en del saker som relaterar till programutveckling och kompilering. Det tar också upp en del om implicit/explicit evaluering. Några delar i det här kapitlet har inte mognat än och behöver utvecklas mera, , och är mer som öppna dörrar till mer teori.

4.1 Grundläggande teori

I andra programspråk som till exempel C, sker en del exekvering under själva kompileringen. Det är makro-expansion med cpp (preprocessorn) och det sker en del optimering gjorda av kompilatorn, till exempel att addera två heltalskonstanter. Man kan göra ganska avancerade beräkningar under kompileringen. C++ har ännu mer förexekvering när den räknar ut referenser. Ju mer man förexekverar ett program desto snabbare går det förstås.

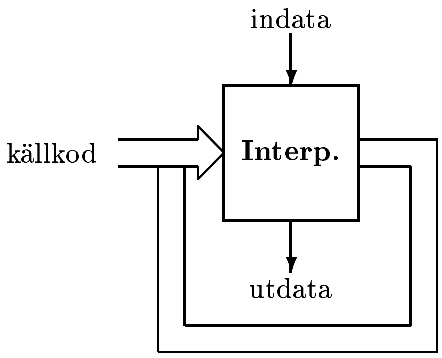
Om man driver denna förexekvering till sin spets och gör all exekvering under kompilering, och om programmet inte har indataberoenden, så får man bara utdata som resultat, och ingen maskinkod. Rnd är just ett sådant språk, och är därför omständligt att kompilera. Anledningen är att förexekvering till sin natur är svår att kompilera.

Ett annat sätt att beskriva Rnd är att se interpretatorn som en serie interpretatorer. Det som skickas mellan dem är delvis exekverad kod, till exempel ett uttryck som är evaluerat.



Figur 4.1: Serie av interpretatorer

Detta kan man skriva som en interpretator som terminerar när det inte finns någon mer källkod.



Figur 4.2: Interpretator *(terminerar när källkoden är tom)*

Eller uttryckt som C-kod:

```
while (source) {
  source = interpret(source);
}
```

Man kan göra en jämförelse mellan den första representationen i figur 4.1 och hur övriga programspråk utvecklats. Från början skrev man direkt i maskinkod. Sedan konstruerade man kompilatorer för att förenkla programmerandet. Efter det kom preprocessorer och så vidare. I utvecklingsordning blir det alltså:

1. maskinkod

2. källkod $\stackrel{kompilator}{\Longrightarrow}$ maskinkod
3. källkod $\stackrel{preprocessor}{\Longrightarrow}$ kod $\stackrel{kompilator}{\Longrightarrow}$ maskinkod
4. konfigureringsfiler $\stackrel{config}{\Longrightarrow}$ källkod $\stackrel{preprocessor}{\Longrightarrow}$ kod $\stackrel{kompilator}{\Longrightarrow}$ maskinkod

Varför inte göra alla stegen i en kraftfullt språk som Rnd, och kanske låta det generera maskinkod. Men då blir maskinkoden i och för sig ett slutmål och det är kanske inte så bra eftersom man vill ha ett dynamiskt program som ändrar sig efter omgivningen och programmerar om sig själv.

I framtiden vill man kanske ha amorfa program som man kan modifiera under körning och slippa hela kompileringsprocessen och omstart av program. Man kan se att dynamisk laddning av bibliotek i program eller dynamiska devicedrivers i operativsystem blir allt vanligare.

Om man ser på vanliga programutvecklingsprocesser så är de att man skapar programmet, testar, ändrar, kompilerar om, testar ... osv. Detta steg med omkompilering kan man kanske ta bort och ersätta med program som anpassar sig efterhand automatiskt till användaren och systemet.

4.2 Implicit/Explicit evaluering

Explicit evaluering är vad vi normalt brukar mena med evaluering. Ett typiskt exempel på detta är eval-funktionen som finns i en del språk. Man kan även kalla en hel kompilering av ett program för explicit evaluering eftersom det genereras ett resultat. Explicit evaluering är alltså när man säger till att det ska ske.

Implicit evaluering är när alla uttryck och värden som refereras automatiskt körs genom en evaluator innan de används. Denna evaluering upprepas tills ett konstant uttryck erhålls. I de flesta andra programspråk sker detta bara en gång.

En av de viktigaste egenskaperna hos Rnd är dess implicita evaluering. Få andra språk har detta och de har ofta eval-funktioner istället, som ofta rekommenderas att inte användas. Istället kan programspråk ofta ha olika beteende beroende på typ på parametern, *overloading*, eller som i C där man har funktionspekare för att få olika beteenden.

En annan variant på detta är språket LPC, där man kan konvertera strängar till funktionsnamn och anropa dem. Där kan man ha en eval som ser ut så här:

```
eval(s) {
  if (sscanf(s,"@%s",f)) {
```

```
    return call_other(this_object(),f);
  }
  return s;
}
```

Denna något förenklade eval anropar funktionen med namnet `f` om det kommer ett `@` följt av en sträng. Denna eval görs sedan på alla ställen där man vill hämta värdet på en variabel. På detta sätt får man nästan en implicit eval. Det här exemplet är hämtat från kod som används, och betraktas av programmerare som kraftfullt och praktiskt.

Min tes är alltså att genom att ha fullständig implicit evaluering får man ett mycket kraftfullare språk. Detta ger många fördelar men också flera nackdelar. Bland annat att koden helt enkelt blir krångligare och svåröverskådlig.

I kapitel 3.6.3 görs en jämförelse med andra programspråk.

4.3 Teori om kompilering

Den här teoridelen är bara ett första försök att analysera Rnd och försöka avgöra om det går att kompilera. Denna del är inte komplett.

Man kan säga att språket innehåller sju primitiva funktioner.

```
text(<text>)
calc(<aritmetiskt uttryck>)
quote(<any>)
table(<name>,<index>,<entry>)
call(<name>,<index>)
set(<name>,<value>)
concat(<any>,<any>)
```

Om man på något sätt vill kompilera koden i Rnd så måste man transformera den till någon kanonisk notation eller form som det är lätt att vidare transformera eller generera objektkod från. De primitiva funktionerna skrivs då:

```
{x 0 foo} skrivs x 0 foo table
{y}       skrivs y y call

Kod                kompilerat
{y z =0}           y[0]= z
{y ‘x z}{z’ =0}    y[0]= x 0 z table z
```

```
{{y} =1}          y y call 1 set
{x {y}z}          x 0 y y call z concat table
```

Efter ett evalueringspass i Rnd kan man se att det resulterar i giltiga uttryck.

```
                {y z =0}     {y 'x z}{z' =0}
{{y} =1}         {z =1}        {x z}{z =1}
{x {y}z}         {x zz}        {x x z}{zz}
```

För `{x z}{z =1}` blir postfixkoden `x 0 z table z 1 set`, vilket är korrekt.

För `x x zzz` skulle det normalt bli `x 0 x table x 1 z table zz zz call` i postfix form. Men enligt den förkompilerade koden ovan blir det `x 0 x 0 z table z z concat table` vilket är fel. Problemet är förstås att ett entry kan innehålla ändringar i syntaxen för det uttryck som ligger utanför. Koden är inte kontextfri.

Det är förstås möjligt att ha flera olika förkompilerade varianter för varje entry, och sedan använda den variant som passar i syntaxen. Det kan innebära att den kompilerade koden blir flera gånger större än källkoden. Det blir enklare att låta bli att kompilera.

Automaten som tolkar Rnd har 38 tillstånd. Det innebär att man måste ha 38 olika kompilerade varianter av samma entry. Den kompilerade koden har då starttillstånd, sluttillstånd och en serie primitiva anrop. Under körning kopplar man sen ihop koden i kedjor där länkpunkterna har samma tillstånd. Exempel: `'{x '` kan ha start 0, slut 4 och primitiv `x 'z}'` kan ha start 4 och slut 0 och primitiv `0 z table` Dessa kopplas ihop eftersom start och slut är samma tillstånd.

I praktiken har automaten fler tillstånd eftersom den har flera tillståndsflaggor.

Ett problem är att man inte kan kontrollera att programmet är syntaktiskt korrekt med en scanning genom koden. Man måste exekvera programmet för att kontrollera syntax. Om sedan användaren kan mata in kod blir det omöjligt.

Ett enkelt exempel på kod som måste exekveras för att kolla om det är syntaktiskt korrekt är: `{x \}}{y =1{x 0}`

4.4 Kompilering

Många frågar sig varför perl, sh, tcl och andra interpreterande språk inte har någon kompilator. De är ju trots allt väldigt användbara språk och kompilering skulle snabba upp dem. Anledningen är just kraftfullheten i den evalueringsmetod och den kontextkänslighet de har och problemet med att kompilera denna på något vettigt sätt. Visserligen är flera av dessa språk semi-kompilerande, men problemet att göra fullständig kompilering finns ändå kvar.

Rnd är också väldigt svårt att kompilera på grund av att det kan programmera om sig själv och att källkoden till programmet inte är statisk. Programmets utseende kan alltså variera under interpreteringen.

Enklare Rnd-program med rättfram kod skulle nog inte vara så svåra att kompilera. Det är bara att kompilera varje sträng i entryna för alla tabeller, och generera kod för dem. Ett uttryck som `{foo `[x+1]´}` kan vara lätt att kompilera eftersom det finns ett fullständigt uttryck i den tabellen.

Kapitel 5

Utvidgningar

Rnd är ett enkelt programspråk som saknar många funktioner som normalt finns i andra språk. Det är lätt att hitta på nya utvidgningar och mer syntaktiskt socker. Jag har strävat efter att minimera mängden inbyggd funktionalitet och typer av subuttryck, för att försöka få ett så rent språk som möjligt. Mycket tid har gått för att tänka igenom om en viss funktionalitet behövs eller om man kan simulera den på något annat sätt. Till exempel är det egentligen då onödigt att man kan skicka parametrar på två sätt men det är svårt att undvika.

En annan anledning till att följande utvidgningar inte blivit implementerade är att det tar extra tid och förmodligen ökar underhållet exponentiellt med antalet funktioner.

Utvidgningarna är uppräknade efter hur pass viktiga de är att implementera för att få ett mer användbart språk.

5.1 Felhantering

Egentligen finns det ingen felhantering. Det som inte kan evalueras skrivs bara ut. Så ett uttryck som `]1+2]` evalueras inte. Ett vanligt fel är att glömma `*` i tabellanrop, till exempel `{foo 1}` istället för `{foo *1}`, när man inte vill sätta variabeln `foo`.

5.2 Stränghantering

Rnd kan inte hantera eller söka i strängar. En ny typ av subuttryck kunde introduceras. Till exempel `<xyz,2>` kunde ersättas med det andra tecknet i strängen, det vill säga `y`. Man kan dock simulera eq, en test om två strängar är exakt lika, med hjälp av följande funktion:

```
{eqx ~ ‘
{{eqx *0} =1}
{{eqx *1} =0}
[{eqx "*0}={eqx "*1}]
{{eqx *0} =0}
’ =~}
{eq ~ ‘{eqx \!{eq *0} \!{eq *1}}’ =~}
```

Egentligen är stränghantering en av de viktigaste sakerna att implementera.

5.3 Fil- och pipehantering

Läsning och skrivning från och till fil eller pipe finns inte heller. Här skulle man även kunna använda `<>`-uttrycket för att läsa en rad från filen.

5.4 Start-stop evaluering

Om man vill använda Rnd som ett riktigt makrospråk är det inte särskilt svårt att implementera ett tecken för start av evaluering. Till exempel kunde man kunna skriva ett snabel-a, `@{f 1}`, för att starta evaluering.

En annan variant är att som i andra makrospråk evaluera identifierare om de är definierade. Om man till exempel definierar `{foo ~ `bar´ =~}`, kan man skriva enbart `foo` istället för `{foo}` och det skulle ersättas med `bar`. Som Rnd är konstruerat kan man ha variabelnamn som två blanka, `` `  ´ ``, och man kan då få Rnd att evaluera två blanka.

Frågan är dock om den skulle göra den här typen av substitueringar, eftersom den kommer att ersätta alla förekomster. Till exempel skulle `{foo}` först ersättas med `{bar}` som sedan skulle evalueras. En möjlig begränsning skulle vara att endast tillåta detta utan subuttryck.

5.5 Dumpar

Just nu kan man bara hämta ett element i en tabell. Det kunde vara praktiskt om man kunde hämta hela definitionen av tabellen. Sen kan man skriva ut den eller skicka den vidare på `STDOUT` till en annan process. Eller så vill man kanske få ut en lista på alla index i en tabell eller antal index och entry i tabellen.

5.6 Exakt get

Att kunna hämta ett entry på ett specifikt index vore också praktiskt. Det skulle man kunna göra med ett kommatecken. Exempel: `{foo 10 bar}` skulle göra att `{foo ,5}` inte skulle returnera någonting.

5.7 Flyttal

Det skulle vara ganska lätt att implementera flyttal, men då får man de vanliga problemen med konvertering och olika datatyper. Man skulle även kunna införa en generell numerisk typ.

5.8 Inbyggda funktioner

Loop-konstruktorn och andra hjälpfunktioner kunde vara fördefinierade. Även aritmetiska funktioner som kvadratrot, exponent, sinus ochcosinus kunde vara fördefinierade om man hade flyttal.

5.9 Komplexa argument vid programstart

En annan utvidgning skulle kunna vara att ge kod som argument vid start av interpretatorn. Till exempel:

```
rnd -'{foo =1}' prog
```

Eller att man kunde ange flera filer som ska evalueras efter varandra.

5.10 Ta bort entryn

För att ta bort ett entry i en tabell skulle man kunna sätta det till tomma strängen. Alltså med `´, men problemet blir här att om man verkligen vill ha ett tomt entry får man göra det på ett annat sätt. En möjlighet är att sätta det till `` `[]´ `` som ändå evalueras till tom sträng.

Det enklaste vore att ha en modifierare som nollställer eller tar bort ett entry och kanske sätter tabellens värde till odefinierat. Som till exempel `!`.

5.11 Lokala variabler och namn på parametrar

För att ett programspråk ska bli riktigt användbart bör man ha möjlighet att ha lokala variabelnamn och parameternamn. En möjlighet är att lösa det som perl gör, med `local` funktionen, eller helt enkelt införa normal parametersyntax.

5.12 Strängvariabler

Variabler borde kunna innehålla strängar. Det är nästan nödvändigt om man vill ha strängar som entry till tabeller. Just nu kan man simulera strängvariabler genom att sätta ett index till en sträng, till exempel `{x ~ `foo´ =~}`, och man sedan skriva `{x}` om man vill hämta värdet. Men om man har strängvariabler skulle man kunna skriva `[x]` precis som vanligt.

5.13 Strängar som index

En intressant utvidgning är att kunna ha strängar som index och inte bara heltal. Interpretatorn skulle sortera in strängarna i bokstavsordning i tabellen och vid indexering skulle det fungera som med heltal. De skulle ligga i en separat lista och inte i heltalslistan. Exempel: `{foo #bar :gazonk}`

En ännu intressantare utvidgning är att kunna ha reguljära uttryck som index. * skulle då motsvara heltalsoändligheten och rentav sammafalla med den.

5.14 Flerdimensionella arrayer

Tabeller med mer än en dimension får man ju om man inför strängar som index. `#1,1` blir då strängen `"1,1"`. Men äkta flerdimensionalitet med samma syntax är kanske att föredra. Man skulle kunna införa tupler eller eller möjlighet att definiera poster med fältnamn.

5.15 Externa funktioner

Det kunde finnas en möjlighet att länka med ett externt program bibliotek. Dessa kunde då ta en länkad lista av entryn som parametrar. `{exfoo :x :y}` skulle skicka med strängarna "x" och "y" till C-funktionen "exfoo", som kan vara deklarerad som `char *exfoo(char *a, char *b);`. Funktionen kan sedan returnera en sträng som sätts in som vanligt.

De här funktionerna kunde laddas in dynamiskt från biblioteket när man använder dem första gången.

5.16 Reguljära funktionsnamn

En kraftfull utvidgning är om man kunde deklarera eller anropa tabeller med reguljära identifierare. Man skulle allstå ha ett vanligt reguljärt uttryck och detta skulle då matcha när man gör anrop. Exempel: `{foo* ~ bar}` skulle göra att `{foo}`, `{foobar}` och `{fooo}` skulle ge `bar` som resultat.

5.17 Temporära tabeller

Ofta när man har if-satser vill man inte att definitionen ska finnas kvar när man exekverat den. Då kunde det vara praktiskt med en slags temporära tabeller som bara finns för ett anrop, och som sedan kastas bort när de använts klart.

Kapitel 6

Implementation

Det här kapitlet tar upp implementationen av Rnd och begränsningar i den.

6.1 Begränsningar

Det finns flera implementationsmässiga begränsingar i Rnd. Att parametrar stackas och sen kastas bort vid ny tilldelning är kanske onödigt. Man kunde behållit dem på stacken, men då finns risken att minnet tar slut om man gör oändliga loopar.

En annan begränsning är storleken på inläsningsbufferten och antalet variabler. Buffertstorleken är just nu 5000 tecken och max antal variabler är 2113. Man borde egentligen låta bufferten och variabelantalet växa dynamiskt.

Oändligheten representeras just nu som MAXINT. Man kan alltså ha ett max antal index i en tabell. Med en annan representation av index borde man kunna ha många fler. Dessutom är behandlingen av oändligheten i aritmetiska uttryck inte helt korrekt.

Det går inte att göra några avancerade beräkningar med oändligheten, utan man får nöja sig med att så fort den får oändligheten eller minus oändligheten som resultat någonstans i det aritmetiska uttrycket så blir det resultatet. Det innebär att `[-(1/0)]` blir ~. Rnd borde egentligen följa IEEE-standarden för beräkningar med oändligheten.

Egentligen borde även den aritmetiska evaluatorn byggts som en DFA, och inte som en rekursiv funktion. Den vanliga DFA'n kunde hanterat positioner för tal, identifierare och siffror precis som för tabeller.

6.2 Nackdelar

Det finns inte bara fördelar med Rnd, utan också en del ganska betydande nackdelar. Alltså inte implementationsmässiga begränsingar eller saker som saknas i Rnd, utan nackdelar som ligger i språkets design.

Den allvarligaste är nog att man ibland inte kan upptäcka felen förrän man exekverar koden. Fel kan alltså ligga gömda i koden tills man vid ett udda tillfälle aktiverar den delen av koden. Det är förstås en följd av att Rnd inte kompilerar och kontrollerar koden.

En annan nackdel är att programspråk som Rnd, som exekverar med hjälp av strängsubstituering, ofta blir väldigt långsamma. För till exempel en loop substitueras och evalueras villkor och kod varje varv utan mellanlagring av resultat.

6.3 Implementation av interpretatorn

Interpretatorn är gjord i C under Unix. Den är implementerad som en DFA (deterministisk finit automat). Egentligen är den konstruerad utifrån en DFA(ϵ), alltså med epsilonförflyttningar, men kompileras till en DFA. Varje tillstånd i DFA'n har sedan en funktion bunden till sig. Till exempel har för `}` (end-bracket) bundits kod för evaluering av tabeller.

DFA'n ligger i en matris, och när ett tecken läses in indexeras arrayen med tecknet och nuvarande tillstånd. Efter det anropas den funktion som är bunden till det tillståndet, varefter den börjar om igen. Detta liknar den lexikaliska analys som finns i kapitel 3 i [ASU86]. Källkoden för detta ser ut så här:

```
c = get_buf();
while(c != EOF) {
  prev_state = state;
  state = dfa1[state][map_dfa1[c]];
  (*action[state])();
  c = get_buf();
}
```

När en tabell läses in, dvs interpretatorn hittar `{`, läses indexen och entryn in och lagras i en tabell buffert och positionerna för entry läggs i en stack. När `}` hittas så hämtas positionerna från stacken och texten hämtas från bufferten och tabellen allokeras. Det kunde ju verka lämpligare om tabellentry allokeras direkt, men en tabell kan ju innehålla subuttryck som evalueras till andra värden. När den träffar på ett sådant subuttryck får den börja om från början med tabellstackningen.

Aritmetiska uttryck i Rnd har den vanliga syntaxen, och därför har en särskild evaluator för sådana uttryck byggts för interpretatorn. Den evaluerar uttrycken rekursivt.

Identifierare lagras i en stor hashtabell med poster, som innehåller ett namn, en pekare till stackade gamla värden, en flagga som anger om variabeln har ett värde och ett fält för värdet. De stackade värdena i en variabel lagras i en enkel länkad lista.

Tabellerna allokeras dynamiskt och är helt enkelt en länkad lista med poster som innehåller index, en sträng med text/kod och en pekare till nästa entry. För varje uppslagning i tabellen måste det göras en sökning i listan, vilket kan ta en del tid för stora tabeller.

Kapitel 7

Övrigt

7.1 Användningsområden

Vad kan man använda Rnd till? Till stora programsystem passar det kanske inte, men till mindre kan det kanske vara användbart. Man kan ha det som subspråk inuti ett större system. Ett möjligt område är en databashanterare som evaluerar med hjälp av Rnd.

7.2 Utvärdering

Det här är ett projekt som tagit flera år, men det har varit roligt och intressant. Att skriva den här rapporten har tagit lång tid, men det beror nog på en del andra faktorer eller kanske att det här är ett område som det inte skrivits mycket om. Själva interpretatorn var i princip klar för ett år sedan. Efter det har det bara varit smärre buggfixar när jag testat exempel i den här rapporten.

Det jag har varit mest rädd för är att folk skulle såga idén helt, men det har aldrig hänt. En del har ifrågasatt och kritiserat vilket förstås är bra, men många har blivit förvirrade över det nya konceptet och senare imponerade. Det här hade de inte tänkt så mycket på. – “Vad menar du nu? Exekvera medan man makroexpanderar. Går det?”

Litteraturförteckning

[ACM87] ACM SIGPLAN. *Interpreters and Interpretive Techniques*, 1987.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. North-Holland, 1990.

[Gri82] Ralph E. Griswold. The evaluation of expressions in Icon. *j-TOPLAS*, 4(4):563–584, October 1982.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Kam90] Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley, Reading, Massachusetts, 1990.

[kW87] Åke Wikström. *Functional Programming using Standard ML.* Prentice-Hall, 1987.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[PH90] Lars Pensjö and Fredrik Hübinette. *LPC manual*, 1990.

[Sal96] Daniel J. Salomon. Using partial evaluation in support of portability, resusability, and maintainability. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 208–222. Springer Verlag, April 1996.

[Sei94] René Seindal. *GNU m4 manual, version 1.4*, 1994.

[Wad85] William W. Wadge. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, 1990.