LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Systemdokumentation

Författare

Warren Crutcher, warcr701@student.liu.se
Viktor Norlin, vikno856@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023
Version 1.0

2023-05-04

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första utkast av dokumentation	2023-05-04

Innehåll

Tabeller

2 Inledning

I kursen TDP019 Projekt: Datorspråk som läses under andra terminen av Innovativ Programmering vid Linköpings Universitet har vi skapat ett programmeringsspråk som heter Whisper.

2.1 Syftet och Målgrupp

Syftet med Whisper är att underlätta hanteringen av enkla matematiska uttryck i terminalen. Målgruppen för Whisper är användare som är bekanta med en Unix terminal men inte nödvändigtvis erfarna programmerare som också önskar sig ha en miniräknare med användardefinierade funktioner i terminalen.

3 Användarhandledning

Whisper fungerar som en terminalbaserad miniräknare. Först måste användaren ladda ner filerna från gitlab. Kopiera filerna med kommandot:

```
$ git clone https://gitlab.liu.se/warcr701/TDP019.git
```

För att installera Whisper, kör installationsskriptet med kommandot:

```
$ sudo ./whisper_install
```

Som gör att Whisper installeras och kan köras som ett exekverbart program vart som helst i terminalen efter att starta om datorn. För att sedan börjar använda Whisper, kör:

```
$ whisper
```

Om installation ej önskas, kan användaren från katalogen där filen `whisper.rb` står kör:

```
$ ruby whisper.rb
```

Terminalen borde se så här ut:

```
[Whisper]:
```

Nu kan användaren skriva olika matematiska uttryck som exekveras direkt i terminalen.

3.1 Syntax exemplar

Användaren kan skriva enkla vanliga matematiska uttryck, som exempel:

```
[Whisper]: 5+6*9
```

Whisper stödjer användarskapade aritmetiska uttryck. Exemplet nedan visar definiering av en funktion, variabel definition, utskrift av ett funktionsanrop till terminalen och iteration genom en while-loop:

```
[Whisper]: let f(x,y) = {x*y + x^y}; let a = 0; while a < 10 do {show(f(a,2)); let a = a+1}
1
4
8
14
24
42
76
142
272
530
=> 10
```

```
[Whisper]:
```

I exemplet ovan skrevs ut resultatet av funktionsanrop `f(a, 2)` medans värdet för `a` ändrades under vare iteration av loopen. Sista utskriften vid `=>` ger resultatet av det sist exekverade uttryck som kördes, nämligen `let a = a + 1`.

Whisper stödjer också att passa en funktion som argument till en annan funktion:

```
[Whisper]: let f(x,y) = {x*y + y^x}; let g(x) = {x^3}; let a = 0; while a < 10 do \
           {show(f(g(a), 2)); a += 1}
```

Det finns två olika sätt att öka värden i en while loop i Whisper. Uttrycket `let a = a + 1` fungerar som en variabel deklaration som gör att enligt räckviddhanteringen (som kallas för scopehanteringen) deklareras `a` i nuvarande scope. Detta gör att efter kodblocket avslutas, kommer `a` ha samma värde som innan kodblocket. I exemplet ovan, skulle `a` ha värdet `0` efter kodblocken avslutas. Med uttrycket `a += 1` kommer användaren undan scopehanteringens regler så att `a` kommer ha efter kodblocken exekveras värdet som den hade inuti kodblocken. På ett sådant sätt kan användaren välja hur de vill hantera variabler inuti loopar.

Funktioner kan också skapas inuti andra funktioner. Exempel:

```
[Whisper]: let f(x, y )  = { if x < 10 then {x^y} else {let g(c) = {c+2}; g(x)}}
[Whisper]: f(10, 3)
```

Anropas `f` med ett värde större än eller lika med `10` kommer `else` satsen exekveras där funktionen `g` skapas och exekveras med `x` värdet som argument. Resultatet av efter evalueringen av exemplet ovan ger oss:

```
=> 12
```

som förväntas.

3.2 Sammanfattning av syntax regler

Nedan följer en sammanfattning av syntaxregler för de grundläggande satserna som stöds i Whisper.

3.2.1 Variabler

Deklaration av variabler, d.v.s. tilldelningssatser:

```
let <variable name> = <value>
```

Där `<variable name>` måste börja med en liten bokstav och får innehåller bara bokstäver, siffror eller `_` tecken.

3.2.2 Funktioner

Funktioner definieras på följande sätt:

```
let <function name>() = {<statement>}
let <function name>() = {<statement_1>; ...; <statement_i>}
let <function name>(<argument>) = {<statement>}
let <function name>(<arg_1>, ..., <arg_i>) = {<statement_1>; ...; <statement_k>}
```

for hur många som helst naturliga heltal `i, k`. Namnet till en funktion följer samma regler som namnet till en variabel men en funktion utan argument måste följas av `()` och högersidan av operatören `=` måste börja med `{` och sluta med `}`, d.v.s. en eller flera `<statement>` måste vara inuti `{}` och flera `<statement>` måste separeras med `;` tecken.

3.2.3 If-satser

Whisper har stöd för `if` satser och `if - else` satser.

```
if {<condition>} then {<statements>}
if {<condition>} then {<statements>} else {<statements>}
```

Uttrycket för `<condition>` måste returnera antingen sant eller falsk vid evaluering och `<statements>` kan vara en eller flera satser separerade med `;` tecknet.

3.2.4 While-loopar

Whisper har stöd för while-loopar för att uppnå iteration och olika sätt att styra ett programs struktur.

```
while {<condition>} do {<statements>}
```

Precis som beskrevs med `if`-satser måste `<condition>` vara en sats som evalueras till antingen sant eller falskt och `<statements>` kan vara en eller flera `;`-separerade satser.

3.2.5 Operationer

Whisper har stöd till ett begränsat antal grundläggande aritmetiska operationer.

Operation	**Symbol**
Binär operatörer	
addition	+
subtraktion	-
multiplikation	*
division	/
modulär division	%
exponentiering	^
omfördela addition	+=
omfördela subtraktion	-=
omfördela multiplikation	*=
omfördela division	/=
och	&&
eller	||
Jämförelse operatörer	
mindre än	<
större än	>
mindre än eller lika med	<=
större än eller lika med	>=
lika med	==
ej lika med	!=
Unär operatörer	
ej	!
negation	-

Tabell 1: Binär och Unär operationer som stöds av Whisper

4 Systemdokumentation

Nedan beskrivs Whisper språket i sin helhet.

4.1 Översikt

Språket Whisper ligger ovanpå den lexikaliska scannern som används i kursen TDP007: Konstruktion av Datorspråk. Den lexikaliska parsern är skapad så att användaren kan skriva olika *regler* som exekveras beroende på nyckelord eller reguljära uttryck som läses av scannern. De olika regler som exekveras bygger ett *abstrakta syntax träd* med olika uttryck kopplad till varandra som sedan exekveras.

De olika delarna som gör ett abstrakt syntax träd består av olika klasser i språket Ruby. Whisper har också en klass som hanterar räckvidden av olika funktioner och variabler, oftast kallat *scopehantering*.

I och med att Whisper har som syfte att fungera som en miniräknare är det tänkt att det lämpligaste sättet att använda Whisper är direkt i en Unix terminal i interaktivt läge där varje uttryck evalueras direkt för att ge användaren snabba svar. Nedan finns ytterligare beskrivning i mer detalj av de olika delarna som tillsammans ligger bakom Whispers konstruktion.

4.2 Grammatik

Whispers grammatik definieras enligt BNF (Backus-Naur Form) och finns i slutet av detta dokument.

4.3 Lexikalisk analys och parsing

Whisper är byggt på parsern som användes i kursen TDP007, som finns i filen `rdparse.rb`. I Whisper projektet finns koden i filen `parser.rb` och innehåller klasserna `Rule` och `Parser`. De här två klasserna tillsammans innehåller ramverket för en så kallade *recursive descent parser* och tillåter programmerare att skapa regler enligt en grammatik för att exekvera kod enligt programmerarens önskemål. I Whisper projektet har författarna inte ändrat varken `Rule` eller `Parser` klasser.

4.4 Evaluering

Filen `whisper.rb` innehåller en implementation av vår BNF grammatiken som skapar instanser av olika klasser beroende på vilka regler som matchas med parsingen av filen `parser.rb`. När en match av en sats inträffas, skapas ett abstrakta syntax träd vems kod finns i filen `nodes.rb` där varje Nod klass har egen uppbyggnad och `eval` funktion. På grund av Whispers betoning på interaktivitet skapas, evalueras och förstörs en trädstruktur för varje körning av ett Whisper program. Ett Whisper program består av antingen en sats (statement) eller flera satser (en lista av statements). I och med att varje Nod klass är för sig ett abstrakta syntax träd länkas ihop olika abstrakta syntax träd där vissa satser blir 'barn' satser till olika abstrakta syntax träd. En 'körning' av ett Whisper program räknas som när användaren trycker 'Enter' tangent vid tangentbordet.

För att Whisper ska komma ihåg variabel och funktion definitioner mellan olika körningar under en session finns en klass som heter `scope.rb` som innehåller klassen `Scope_Manager` vems jobb det är att förvara definitioner av olika funktioner och variabler samt vilket värde de olika 'symboler' för både funktioner och variabler representerar.

4.5 Klasser

I och med att Whisper implementeras i det objektorienterat programmeringsspråket Ruby används klasser som egendefinierade datatyper. Det som klasserna hanterar är omfattning (scoping), parsing, konstruktion och evalueringen av uttryck i ett abstrakt syntax träd (AST).

4.5.1 Scopehanteringen

Klassen `Scope_Manager` hanterar en variabels eller funktions omfattning i Whisper. Sättet som `Scope_Manager` fungerar är som en hög (stack) av tabeller där varje tabell har som nyckelvärde ett namn till en funktion eller variabel och har som värde en referens till ett AST-objekt som representerar det givna uttrycket. I Ruby uppnås det här beteende med en datamedlem till `Scope_Manager` som heter `scope` som är en `Array` av `Hash` tabeller. `Scope_Manager` är statisk, d.v.s. att det finns bara en instans av `Scope_Manager` i Whisper och den nås av alla andra klasser. När programmet går in till en ny nivå av scope läggs en ny `Hash` till `Scope` arrayen i `Scope_Manager`. Klassen har funktionen `symbol_lookup()` som hämtar värdet av ett namn. Funktionen börjar alltid leta i nuvarande nivå i stacken för att leta fram värdet av ett namn. Om värdet inte finns fortsätter sökningen i de andra nivåer tills antingen variabelns värde hittas eller så returneras funktionen `nil`.

Förutom funktioner för att lägga till eller ta bort en omfattningsnivå finns en funktion `symbol_update()` som ändrar värdet av en symbol oavsett nuvarande omfattningsnivå. På ett sådant sätt kan en variabel ändras inuti en loop eller annat kodblock och beroende på användarens önskemål antingen behåller dåvarande värde eller uppdateras även efter kodblock exekveras.

4.5.2 AST Noder

Förutom klasserna som tillhör parsern som finns i filen `parser.rb` finns flera olika Node klasser som finns inuti filen `nodes.rb`. Dessa är:

- `Node_Arithmetic` som representerar ett abstrakta syntax träd med en operation och ett eller två *barn* noder beroende på om operatören är binär eller unär.
- `Node_Assignment` som representerar ett abstrakta syntax träd med operatören `=` som också föra in nya symboler till `Scope_Manager` klassens `stack` av symbol tabeller.
- `Node_Boolean` som representerar ett booleskt atom av antingen nyckelord 'true' eller 'false'.
- `Node_Built_In_Function` som representerar ett abstrakta syntax träd för inbyggda funktioner. I nuläget har whisper stöd för bara en funktion, nämligen funktionen `show()` som motsvarar en vanlig `print()` funktion i andra programmeringsspråk. Skulle Whisper utökas för att inkludera flera inbyggda funktioner (som var målet från början) skulle de hanteras av den här klassen.
- `Node_Comparison` som representerar ett AST med jämförelse operatörer, som `< > <= >= ==` och `!=`.
- `Node_Double` som representerar ett AST för ett värde som kan vara ett decimalt tal. Noden har inga barn.
- `Node_Function_Call` som representerar ett AST för hanteringen av ett funktionsanrop. Klassen använder också `Scope_Manager` klassen för att lägga till en ny omfattningsnivå vid funktionskropps start och omfattningsnivån tas bort efter evalueringen avslutas.
- `Node_Function_Define` som representerar ett AST för hanteringen av en funktionsdefinition. Evalueringen består av att lägga funktionsdefinition i symbol tabellen i nuvarande omfattningsnivån.
- `Node_If` som representerar ett AST för `if`-satser. Den här noden har som barn ett AST som är en `condition` ett annat AST nod `then_block` för uttryck som evalueras ifall `condition` är sann och antingen ett barn som är `nil` eller om en `else` sats finns ett AST nod som heter `else_block` som evalueras om `condition` evalueras till falskt.
- `Node_Integer` som representerar ett heltal som AST.
- `Node_List` som representerar ett AST och ärver från Rubys `Array` klass. En `Node_List` har som syfte att hantera antingen en lista av argument eller en lista av uttryck som ska evalueras. Att `Node_List` ärver från `Array` är för att ta nytta av de olika inbyggda funktioner som ingår i Rubys `Array` klassen.

- `Node_Logical_Op` som representerar ett AST för logiska operationer, som `&&` || `!` och `==`.
- `Node_Reassign` som representerar ett AST för operationer `+= -= *= /=`. De här operationerna undivkar `Scope_Manager` klassens hantering av tilldelnings operationer.
- `Node_Variable` som representerar ett AST för ett variabelnamn. Evalueringen av en `Node_Variable` instans består av hämtning av den AST noden som innehåller uttrycket med samma namn som variabelns instans.
- `Node_While` som representerar ett AST för en `while`-loop. En `Node_While` har som barn noder ett AST för `condition` och `block` där noden i `block` evalueras så länge satsen i `condition` returnerar sann.

4.6 Kodstandard

Whisper är skapad för att ha en minimal syntax. Som syns i grammatiken och kodexemplar finns inte så många nyckelord. Tanken är att ett program är ett uttryck eller en sekvens av uttryck och förutom `;` tecken som används för att skilja mellan olika uttryck bryr Whisper inte om ett uttryck se ut som

```
let          a =                                          18
```

eller

```
let a=18
```

så länge att nyckelord matchas. Användning av nyckelord `then` och `do` är mer för att ge en 'mänsklig' sätt att beskriva instruktioner.

4.7 Paketeringen av koden

Som beskrivs i del 3 Användarhandledning kan whisper installeras så att den finns exekverbar vart som helst i en Unix terminal eller så laddas ner de givna filerna och i katalogen som innehåller filen `whisper.rb` körs kommandot:

```
$ ruby whisper.rb
```

Terminalen borde visa:

```
[Whisper]:
```

och då kan användaren börja skriva uttryck som beskrivs i Användarhandledning. För att installera Whisper till systemet, kör:

```
$ sudo ./whisper_install
```

Efter installationen borde Whisper vara tillgänglig i terminalen med en medföljande manualsida.

5 Erfarenheter och reflektion

Innan projektet drog i gång så var gruppens båda medlemmar medvetna om att ambitionerna bör hållas på en låg nivå, då projektets omfattning är relativt stort. Vi hade aningar om att vi skulle stöta på problem som skulle göra att utvecklingen tog längre tid än tänkt. Trots att våra ambitioner var relativt låga redan innan projektet började så var vi tvungna att ytterligare sänka våra krav en kort tid efter projektet dragit i gång. Från början var tanken att vårt språk skulle vara statiskt typat och kompilerat, något som vi gav upp på omgående då det kändes som en alldeles för svår uppgift.

Det rådde stor frustration i gruppen under stora delar av projektets gång. De första veckorna av projektet lades mycket arbete på andra kurser, vilket gjorde att projektet fick lägre prioritet. När väl projektet var i

fokus så hade gruppen svårt att komma i gång och få ner någon konkret och användbar kod. Både Viktor och Warren kände att avsaknaden av dokumentation kopplad till rdparse försvårade arbetet markant. Detta gjorde att Warren på egen hand under sin fritid utvecklade en alternativ version av programspråket med hjälp av verktygen Flex och Bison i C. När gruppen hade två versioner i två olika verktyg så var det svårt välja en version av projektet att gå vidare med. Dessutom så gick utvecklingen i Flex/Bison vid denna tidpunkt snabbare än utvecklingen i rdparse. Detta var till stor del en följd av att gruppen inte än kommit fram till en bra struktur över hur parsingen ska gå till i rdparse. Under denna period så var båda gruppens medlemmar frustrerade och förvirrade, och moralen i gruppen var inte särskilt bra. Efter ett antal veckor fyllda med frustration så kom gruppen fram till en metod som löser parsingen i rdparse och ger ett förväntat resultat. När denna metod var på plats så gick utvecklingen framåt i rask takt i jämförelse med innan. Med facit i hand så hade gruppens initiala plan troligtvis gått att genomföra om inte flera veckors arbete hade gått till spillo på grund av osäkerhet gällande hur implementationen skulle genomföras.

<program>	::=	<statement_list>
<statement_list>	::=	<statement> | <statement_list> ';' <statement>
<statement>	::=	<built_in_evaluate> | <if_statement> | <while_statement> | <function_define> | <assignment_statement> | <expression>
<built_in_evaluate>	::=	'show' '(' <expression> ')'
<if_statement>	::=	'if' <expression> 'then' '{' <statement_list> '}' 'else' '{' <statement_list> '}' | 'if' <expression> 'then' '{' <statement_list> '}'
<while_statement>	::=	'while' <expression> 'do' '{' <statement_list> '}'
<function_define>	::=	'let' <id> '(' ')' '=' '{' <statement_list> '}' | 'let' <id> '(' <argument_list> ')' '=' '{' <statement_list> '}'
<argument_list>	::=	<id> | <argument_list> ',' <id>
<assignment_statement>	::=	'let' <id> '=' <expression> | <variable> <re_assign_op> <expression>
<id>	::=	/[a-z_][a-zA-Z0-9_]*/
<function_evaluate>	::=	<id> '(' ')' | <id> '(' <explist> ')'
<explist>	::=	<expression> | <explist> ',' <expression>
<expression>	::=	<term> | <variable> | /continue/
<term>	::=	<logical_term> | <arithmetic_term>
<arithmetic_term>	::=	<arithmetic_factor> | <arithmetic_term> <addop> <arithmetic_factor> | <arithmetic_term> <rel_op> <arithmetic_factor>
<arithmetic_factor>	::=	<arithmetic_exponential> | <arithmetic_factor> <mulop> <arithmetic_exponential> | <arithmetic_factor> <eq_op> <arithmetic_atom>
<arithmetic_exponential>	::=	<arithmetic_atom> | <arithmetic_exponential> <exop> <arithmetic_atom>
<arithmetic_atom>	::=	<function_evaluate> | <literal> | '(' <arithmetic_term> ')' | <unary_minus> <arithmetic_atom>
<logical_term>	::=	<logical_factor> | <logical_term> <orop> <logical_factor>
<logical_factor>	::=	<logical_atom> <logical_factor> <andop> <logical_atom>
<logical_atom>	::=	<bool> | '(' <logical_term> ')' | '!' <logical_atom>
<re_assign_op>	::=	<increment> | <decrement> | <re_divide> | <re_multiply>
<unary_minus>	::=	/\-/
<increment>	::=	/\+\=/
<decrement>	::=	/\-\=/
<re_divide>	::=	/\/\=/
<re_multiply>	::=	/*\=/
<add_op>	::=	/\+/ | /\-/
<mul_op>	::=	/*/ | /\// | /\%/
<ex_op>	::=	/\^/
<or_op>	::=	/'||'/
<and_op>	::=	/'&&'/
<rel_op>	::=	/\>/ | /\</ | /\<\=/ | /\>\=/
<eq_op>	::=	/\=\=/ | /\!\=/
<variable>	::=	/[a-zA-Z_][a-zA-Z_0-9]*/
<literal>	::=	<digit> | <variable>
<digit>	::=	/\-?\.?[0-9]+/ | /\-?[0-9]+\.[0-9]*/
<bool>	::=	'true' | 'false'

Tabell 2: BNF Grammatik för Whisper programmeringsspråk