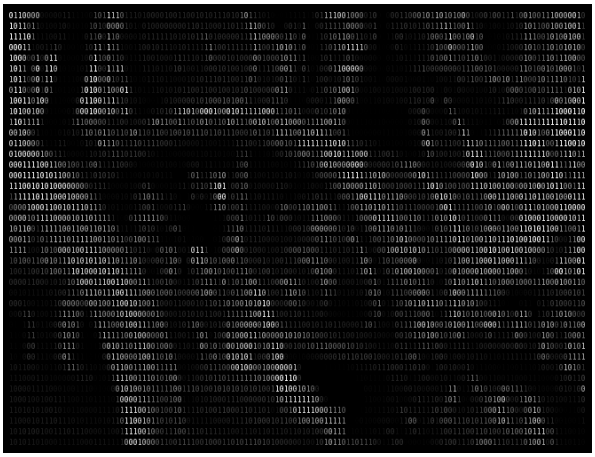
Virtuell maskin i Ruby



André Teintang och Pontus Welin
Linköpings universitet
Linköping
2010-05-17

Sammanfattning

Syftet med projektet var att implementera en virtuell maskin samt att skapa en översättare från ett högnivåspråk till VM. Vi valde att implementera C++ som vårat högnivåspråk.

Vi implementerade VM och kompilator i Ruby. För att testa koden under utvecklingen använde vi oss av testfall.

Målet var att den virtuella maskinen skulle ha så få operationer som möjligt för att vara enkel att implementera, medan dessa operationer, trots dess enkelhet, skulle vara nog för att implementera det övre språket som skulle kunna ha många och avancerad operationer.

Vi hade som mål att implementera funktioner, variabler, while, for, if, heltalsaritmetik och logiska operationer, samt i mån av tid listor, strängar och klasser.

Innehåll

1 Inledning

Detta projekt genom fördes under första året på kandidatprogrammet Innovativ Programmering i kursen ”Projekt: Datorspråk” (kurskod TDP019).

1.1 Syfte

För att lära oss mer om programmeringsspråk och hur de är uppbyggda ville vi skapa ett eget språk och en egen kompilator eller egen programtolk. Efter diskussion kom vi fram till att vi skulle skapa en virtuell maskin som kan exekvera kod skriven med en uppsättning enkla operationer som använder omvänd polsk notation. Samt att skapa en kompilator som kan översätta ett mer abstrakt språk till kod skriven med denna uppsättning enkla operationer.

1.1.1 Den enkla operationsuppsättningen

Huvudmålet med operationsuppsättningen var att den ska ha så få operationer som möjligt för att göra det enkelt att implementera en virtuell maskin. På grund av detta mål valde vi att alla operationer som kan implementeras av redan existerande operationer tas bort. Ett exempel är < (mindre än) och > (mer än) operatorerna, där ”a > b” är samma sak som ”b < a”, alltså kan operatorn < användas istället för > bara man byter plats på operanderna. Vi lägger alltså större tyngd på att språket ska ha få operationer, än att det ska vara lättläst eller intutivt. Vi kommer hädanefter kalla denna uppsättning operationer för Postfix.

1.1.2 Det abstrakta språket

Motsatt till Postfixspråket ska det abstrakta språket vara så lättläst och lätt skrivet som möjligt, där vi vill erbjuda den funktionalitet som finns i vanliga programmeringsspråk. Möjligheten att skapa funktioner, tilldela variabler, listor, klasser och kontrollsatser som variabel-scope, villkorssatser och loopar. För att ta upp ett tidigare exempel för att ytteligare visa skillnaden på språken ska operatorerna < och > båda finnas med, eftersom det finns tillfällen där det är mer intutivt och enklare (och kanske mer lättläst) att kunna använda dem båda. Vi kommer härdanefter kalla det abstrakta språket och koden som skrivs i det för Infix.

1.1.3 Kompilatorn och den virtuella maskinen

Kompilatorn och den virtuella maskinen ska vara helt fristående från varandra och ska inte på något sätt kommunicera med varandra. Kompilatorn ska bara konvertera Infix till Postfix, och den virtuella maskinen ska bara exekvera Postfixkod. Anledningen till detta är att det ska vara möjligt att implementerar andra abstrakta språk till den virtuella maskinen.

1.1.4 Begränsning av implementation

Att konvertera Infix till Postfix är en lång och komplicerad process. Därför valde vi att vi att i första hand implementera konvertering för heltalsaritmetik, logik, variabler, funktioner, kodblock (scope) samt kontrollsatserna if, while och for. Om det fanns tid över skulle listor, strängar och klasser implementeras.

1.2 Implementeringsplan

Implementationen delades upp i tre delmål. Postfixspråket och den virtuella maskinen, Infixspråket samt konverteringen från Infix till Postfix. Postfixspråket var tänkt att hämta inspiration från stackbaserade språk som Forth och Postscript. Infixspråket var tänkt att inspireras av Python och C++, de språk vi tidigare studerat på kandidatprogrammet för Innovativ Programmering.

I första hand skulle Postfixspråket och den virtuella maskinen skrivas. Detta för att vi skulle stå på en stabil grund med givna regler och för att undvika att vi flyttar ner funktionalitet från Infixspråket till Postfixspråket. Vi bestämde oss också för att använda mycket testfall, för att enklare spåra fel när vi bytte någon implementation i koden.

Efter det skulle en BNF skrivas för Infixspråket. Detta så att vi kunde starta implementationen av detta språk. Sen skulle vi göra kompilatorn för Infixspråket, baserat på BNFen, som skulle översätta till Postfix. Denna del var det som vi trodde skulle ta mest tid och jobb.

2 Användarhandledning

2.1 Stacken och den virtuella maskinen

Den virtuella maskinen arbetar genom att lägga operander på en stack, som man sedan manipulerar med olika operationer. Det finns också en heap där man kan allokera och avallokera minne för att enkelt kunna skapa referenser. Det ända man kan lägga på Postfixstacken är heltal och de booleska värdena true och false. Det är viktigt att veta att VM inte kan se om programmet är giltigt, utan förlitar sig på att koden är korrekt.

Något man måste tänka på i stackbaserade språk är ordningen som man lägger värden på stacken när man gör operationer, eftersom det är ordningen som avgör prioriteten t.ex. i aritmetik.

Exempel: ”1 + 2” i stackspråksoperationer.

1: Lägg 1 på stacken
2: Lägg 2 på stacken
3: Gör addition. På stacken ligger det nu 3.

Exempel: ”1 + 2 * 3” i stackspråks operationer.

1: Lägg 1 på stacken
2: Lägg 2 på stacken
3: Lägg 3 på stacken
4: Gör multiplikation, efter operationen ligger det 1 och 6 på stacken.
5: Gör addition, på stacken ligger det nu 7.

2.2 Postfixspråket

Postfixspråket är uppbyggt av 17 olika operationer och använder omvänd polsk notation. Detta innebär att operationen står efter operanderna, till skillnad mot den normala notationen med infix där operationen står mellan operanderna. Alla språkets operationer separeras med *whitespaces*, i sammanhanget ses också att lägga operander på stacken som en operation.

Exempel på postfixkod där man lägger 1, 2, 3, true och false på stacken.

”1 2 3 true false”

Exempel på postfixkod för ”1 + 2 * 3”, där operanderna + och * används.

”1 2 3 * +”

Postfixkod beskriver inte stackens utseende, utan operationerna man gör på den. Man kan alltså inte veta hur stacken ser ut genom att se på Postfixkoden, man måste följa alla operationer till den platsen för att veta det. Detta gör att det är väldigt svårt att skriva i Postfix för hand och ännu svårare att läsa.

För en mer ingående beskrivning av Postfixspråkets operationer, se bilaga 1.

2.3 Infixspråket

Infixspråket är den implementation av det abstrakta språk som användes för att visa hur kraftfullt Postfix är. Det är en mindre omfattande implementation av C++, där bara den mest grundläggande funktionaliteten finns med. Våran implementation kan användas som inspiration till hur man själv skulle kunna implementera sin egen kompilator till Postfix.

Språkets BNF finns i bilaga 2.

2.4 Översättning

För att kunna översätta från ett högnivåspråk till postifixspråket så måste man förstå konceptet med stacken och hur man kan konvertera de delar av högnivåspråket som inte finns i postifixspråket.

Det finns många saker man måste tänka på och det beror på vilket språk man ska översätta från. Allting som finns i högnivåspråket men inte finns i postfixspråket måste simuleras. Till exempel valde vi att implementera C++ och blev då tvungna att simulera bland annat variabler och scope samt funktioner och återhoppsadresser.

2.4.1 Expressions och Statments

I programmeringsspråk kan oftast operationer delas upp till expressions och statements, det är viktigt att man ser skillnaden på dessa typer av exekvering av kod.

Exempel på expression är ”1 + 1” och ”1 == 3”

Exempel på statement är ”puts 3”

Skillnaden mellan de båda är att expression lämnar något på stacken medan statement inte gör det.

2.4.2 Variabler

Sättet att hantera variabler i en stack är att lägga dess värde på stacken och sedan referera till dem relativt från stacktoppen med hjälp av referenser. Anledningen till att detta görs relativt är att man inte kan veta en variabels absoluta adress på stacken. Ett exempel på det är när en funktion anropas rekursivt. Då får varje enskilt funktionsanrop en egen uppsättning av lokala variabler som ligger på olika stackdjup.

Man kan tänka sig att en variabeldefinition är ett statement som också fungerar som ett expression, fast på en lägre nivå än vanliga beräkningar, eftersom de lämnar kvar något på stacken som efterkommande statments måste ha med i sina beräkningar.

Se bilaga 3 för ett ingående exempel.

2.4.3 Scope

Huvudidén med Scope handlar om att kontrollera att en variabel kan skapas och när de ska plockas bort. Man måste då på något sätt hålla reda på stackdjupet när ett scope börjar för att man sedan ska kunna återställa djupet när man går ur det. Svåra problem kan uppstå då man ibland vill hoppa ur flera scope samtidigt vilket görs med bland annat *continue*, *break* och *return*. Detta bidrar till att man även måste hålla reda på stackdjupet vid början och slut på loopar och funktioner.

Vid kontrollsatsen if kan kan man ta två olika vägar, en för sant och en för falskt. Det är då bra att använda sig av ett nytt scope vid varje gren. Eftersom det garanterar att man inte introducerar en ny variabel X på stacken i en gren men inte i en annan, vilket skulle göra det omöjligt att veta om X finns på stacken efter if-satsen.

Exempel med flera scope och överskuggning:

```
0:
1: {
2:      int a = 1;
3:      {
4:              int t = 2;
5:              int a = 3;
6:      }
7: }
8:
```

Rad 0: Antag att stacken är tom
[ ]
Rad 1: Markerar på något sätt att ett scope börjar, i detta exempel görs det med |-tecknet
[|]
Rad 2: Initerar variabeln a (Vi vill visa att det teoretiskt ligger a på stacken därför skriver vi a istället för värdet på a som är 1).
[| a]
Rad 3: Nytt scope
[| a |]
Rad 4: Initerar variabeln t
[| a | t]
Rad 5: Initerar variabeln a igen. Detta görs eftersom den nya a-variabeln nu överskuggar den gammal.
[| a | t a]
Rad 6: Slutet på andra scopet. Nu måste alla variabler som ligger i det scopet tas bort och med hjälp av |-markeringen vet vi hur många variabler som ska tas bort. I detta fall två.
[| a]
Rad 7: Slutet på första scopet, det enda som ligger i detta scope är a och det blir bara en variabel som måste tas bort. Stacken är nu tom.
[ ]

Postfixkoden för hela detta exempel blir ”1 2 3 2 pop 1 pop”.

Detta exempel visa också att möjligheten finns att göra en optimering, genom att ersätta ”2 pop 1 pop” med ”3 pop”.

2.4.4 Funktioner och återhoppsadresser

Konceptet man vill använda med funktioner är att kunna anropa funktioner på samma sätt som man gör med Postfixoperationerna. Man vill utöka det operationsset som redan finns. Dvs. man vill lägga argumenten på stacken, anropa en funktion som konsumerar dem och sedan lämna resultatet på toppen.

Det problematiska med funktioner är att man måste hoppa till ett kodavsnitt, som man sedan måste hoppa tillbaka från för att forsätta. För att lösa detta så läggs inargument och återhoppsadress på stacken innan man hoppar till funktionen och kör dess kod. Funktionen kan då använda sig av återhoppsadressen för att komma tillbaka. När hela funktionsanropet är klart förväntas man ha ersatt inargumenten med ett returvärde på stacken.

En sak att anmärka på är att funktioner både kan användas som ett expression och statement.

Exempel:

```
1: int one(){
2:      putchar('1');    // C++ funktion som skriver ut en char till stdout.
3:      return 1;
4: }
5: int main(){
6:      int a = 10 + one();
7:      one();
8:      return a;
9: }
```

På rad 6 används funktionen one för hämta ett värde för en beräkning. Medan den på rad 7 bara används för att skriva ut '1' till stdout, i detta fall måste man köra pop på returvärdet som ligger på stacken innan vi kan försätta med rad 8.

2.4.5 Dubbel exekvering

Att tänka på när man genererar Postfixkod från abstraktare funktioner, är att man alltid måste använda resultat av tidigare operationer.

Exempelvis den logiska operationen xor, ”a xor b” finns inte i Postfixspråket, men kan ersättas av ”((not a) and b) or (a and (not b))”. Den naiva implementationen av xor i postfix är

”a not b and a not b and or”.

Om a är false och b är true skulle det generera postfixkoden:

”false not true and false not true and or”

vilket är sant, Men byter man a till ett mer komplicerat uttryck som ”3 < 2” (”3 2 <” i Postfix) genereras postfixkoden

”3 2 < not true and 3 2 < not true and or”

Man utför då operationen ”3 < 2” flera gånger. Detta kan ge större konsekvenser bortsett från att det går långsamt, skulle man använda den tidigare exempel funktionen one() istället för 2 skulle utkriften '1' ske två gånger. För att lösa problement gör man operationerna a och b först, så att resultatet av dom ligger på stacken, dessa värden kopierar man sedan för att göra de logiska operationerna.

3 Systemdokumentation

3.1 Installation

Installationen sker manuellt. Filerna finns på http://github.com/AndTe/TDP019 och kan hämtas med Git (finns på http://git-scm.com/download). Alternativt kan man ladda ner varje fil manuellt från github. Filerna är beroende av Ruby.

Filerna som Infixkompilatorn är beroende av är:

compileinfix
InfixParser.rb
rdparse.rb
InfixNodes.rb

Filerna som Postfix VM är beroende av är:

runpostfix
PostfixParser.rb
Stack.rb

3.2 Kompilera Infixkod

Man kompilerar sin Infixkod med kommandot compileinfix som kan ta in ett eller två argument:

./compileinfix infix.cpp postfix.pf

Skapar en postfix.pf med den genererade Postfixkoden.

./compileinfix fil1.cpp

Skapar en fil1.pf med den genererade Postfixkoden.

Glöm inte att sätta compileinfix som en exekverbar fil.

3.3 Exekvera Postfixkod

Man exekverar sin Postfixkod med kommandot runpostfix med en postfixfil som inargument

./runpostfix postfix.fp

Glöm inte att sätta compileinfix som en runpostfix fil.

4 Diskussion

Vi valde C++ som det övre högnivåspråk vi översatte från. Detta val berodde på att vi ville ta ett språk som många känner till. Det är också ett språk som vi är bevandrade i. Det kunde egentligen ha varit vilket språk som helst, men för oss fungerade det bäst som exempel för hur den virtuella maskinen kan användas. Under projektet har vi lärt oss mycket om hur programmeringsspråk är uppbyggda och speciellt om hur stacken används i abstrakta språk som C++. Det har väldigt lärorikt.

Vi lyckades med större delen av de mål vi satte upp i början varav alla huvudmål. De mål vi inte implementerade var vector, string och class. Så vitt vi kunde se i våra testfall, fungerade allt precis som i GNU Compiler Collection för C++.

Det som har tagit mest tid har varit att rätta till misstag när det gäller stackdjup som medförde fel vid exekvering av postfixkoden. Detta hade varit lättare, om vi hade gjort en bättre debugger, eftersom det vid fel i C++-kod sägs bara vad som gick fel, men inte på vilken rad i koden. Vi kunde

ha löst det genom att göra en egen lexer och parser. Men vad vi kunde se gick det inte att lösa med rdparser över huvud taget.

Under projektet ändrades implementationsplanen något. Till stor del höll vi oss till det vi tänkte, men till exempel generaliserade vi bort duplicate och assign från postfixspråket. Vi hade i början två olika tilldelningar för heap och för stack. Men vi slog samman implementationen för att göra det enklare och mer generellt.

Bilaga 1 Postfixoperationer

Här följer en beskrivning av de operationer som finns i VM-språket.

I exemplen beskrivs ibland både referenser till kod och till stack. Referenser till kod används enbart vid goto och if.

När det är exempel på hur stacken kan se ut så används följande sätt att skriva:

[1 2 3]

Detta motsvarar en stack som har 3 överst på stacken och 1 underst i stacken.

När det är exempel på hur koden ser ut så används följande notation:

*”stacktop 5 3 * - goto”*

Programmet körs från vänster till höger och är indexerat från 1 och uppåt. Dessa index används vid goto och if som adress att hoppa till.

+, -, *, /

Normala matematiska operatorer. Konsumerar två operander på stacken och lägger en operand, resultatet av operationen, på stacken.

Exempel:

”7 5 +” lämnar 12 på stacken.

[12]

*”5 3 * 4 +” lämnar 19 på stacken..*

[19]

*”3 4 + 5 *” lämnar 35 på stacken.*

[35]

==

Jämförelseoperator för likhet. Konsumerar två operander på stacken och lägger en boolesk operand, resultatet av operationen, på stacken.

Exempel:

”5 5 ==” lämnar ”true” på stacken.

[true]

”5 7 ==” lämnar ”false” på stacken.

[false]

<

Jämförelseoperator för mindre än. Konsumerar två operander på stacken och lägger en boolesk operand, resultatet av operationen, på stacken. Den här operatorn används även för >, <= och >=.

Exempel:

”5 7 <” lämnar ”true” på stacken.

[true]

”7 5 <” lämnar ”false” på stacken.

[false]

stacktop

Lägger på stacken en referens till det översta elementet i stacken.

Exempel:

Om stacken ser ut så här [5 4 7] så returnerar ”stacktop” ”3” och efteråt så kommer alltså stacken se ut så här [5 4 7 3]

pop

Konsumerar en operand och tar bort så många element som dess värde från stacken.

Exempel:

Om stacken ser ut så här [5 4 1 7 2] före så kommer den se ut så här efter [5 4]. Den konsumerar 2:an och tar bort så många element.

goto

Konsumerar en operand och hoppar till den adressen i koden.

Exempel:

”1 5 goto exit 4 goto”

[1 5], goto konsumerar översta och går till motsvarande index i koden. Dvs platsen där det står ”4”. Och fortsätter därifrån.

[1 4], goto gör på samma sätt här, men nu är det till index 4, vilket är platsen för ”exit” och programmet avslutas alltså. Stacken ser då ut så här [1].

swap

Byter plats på de två översta elementen i stacken.

Exempel:

Före: [1 2 3]

swap körs

Efter: [1 3 2]

exit

Avslutar programmet.

output

Konsumerar en operand och skriver ut den till standard output som ett ASCII-tecken.

Exempel:

[65]

output

65 konsumeras och 'A' skrivs ut till standard output (eftersom 65 är ascii-värdet för 'A').

input

Tar in ett värde från standard input och lämnar dess ASCII-värde på stacken.

Exempel:

Om man trycker på A (versalt a) så läggs 65 på stacken

[65]

not
Konsumerar en boolesk operand och lägger dess motsats på stacken.

Exempel:

[true]

not

true konsumeras och false läggs på stacken

[false]

and, or

Normal logisk operator. Konsumerar två booleska operander på stacken och lägger en boolesk operand, resultatet av operationen, på stacken.

Exempel:

[true false]

”and”

de två operanderna konsumeras och operationen ”and” körs på dem. Enligt normal logik så returneras false som läggs på stacken.

[false]

if

If fungerar som en goto med ett villkor Konsumerar en operand (som adress) och en boolesk operand. Om den booleska operanden är false så hoppar VM till adressoperandens värde i koden. Om den booleska operanden är true så görs inget hopp.

Exempel:

”2 6 false if exit 1 pop 5 goto”

[2 6 false], if konsumerar de två översta och eftersom den booleska operanden är false så hoppar koden till det index som motsvaras av värdet från den andra operanden. Index 6 motsvarar den plats där det står ”1”. Koden fortsätter därifrån.

[2 1], översta operanden konsumeras (1) och pop tar bort så många element från stacken. 2an från stacken tas alltså bort, varpå stacken är tom. Sen läggs 4 på stacken.

[5], goto hoppar i koden till index 5, vilket är platsen för ”exit” och programmet avslutas alltså. Stacken ser då ut så här [] (den är tom).

assign_to_reference

Konsumerar två operander, där den första är referens och den andra är värde, och tilldelar värdet där referensen pekar på stacken eller heapen.

[0 2 1 4]

assign_to_reference. De två översta operanderna konsumeras och värdet som var överst på stacken skrivs på positionen som motsvaras av det värde som är näst överst på stacken. Det betyder att 4 skrivs till position 1.

[4 2]

reference_value

Konsumerar en operand (som referens) och hämtar värdet från den position som värdet motsvarar och lägger detta på stacken.

Exempel:

[4 2 1]

reference_value

Konsumerar den översta operanden på stacken (1) och lägger på stacken den operand på stacken som ligger på det index.

[4 2 4]

delete_reference

Konsumerar en operand som referens och frigör minnet för den position på heapen som referensen motsvarar.

referens
Konsumerar en operand som motsvarar antalet referenser som ska allokeras sekventiellt på heapen. Lägger sen den första referensens adress på stacken.

Bilaga 2 Infix BNF

<program> ::= [<variable_declaration> <function_declaration>]*

<nonzero_digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digit> ::= 0 | <nonzero_digit>

<letter> ::= "A"..."Ö" | "a"..."ö"

<alpha_numeric_character> ::= <digit> | <letter>

<string> ::= ""<stringitem>*""

<logical_value> ::= "true" | "false"

<stringitem> ::= <stringchar> | <escape_sequence>

<stringchar> ::= <any character except the quote>

<escape_sequence> ::= "\" <any ASCII character>

<integer> ::= ["-"] <digit> | ["-"] <nonzero_digit><digit>+

<float> ::= ["-"] 0"."digit+ | ["-"] <non-zero digit><digit>*"."<digit>+

<array> ::= "["<arrayitem>("," <arrayitem of the same type as the first>)*
"]"
| "[]"

<arrayitem> ::= <variable> | <string> | <integer> | <float>

<identifier> ::= <letter>+ <alpha_numeric_character>*

<datatype> ::= <identifier>

<variable> ::= <identifier>

<global_variable> ::= <variable>

<expression_list> ::= <expression>
| <expression> ("," <expression>)+

<expression> ::= <or_expr>

<or_expr> ::= <or_expr> "or" <and_expr>
| <and_expr>

<and_expr> ::= <and_expr> "and" <not_expr>
| <not_expr>

<not_expr> ::= "not" <not_expr>
| <comparison_expr>

<comparison_expr> ::= <plus_expr>
| <comparison_expr> "<=" <plus_expr>
| <comparison_expr> ">=" <plus_expr>
| <comparison_expr> "==" <plus_expr>
| <comparison_expr> "!=" <plus_expr>
| <comparison_expr> "<" <plus_expr>
| <comparison_expr> ">" <plus_expr>

<plus_expr> ::= <multiply_expr>
| <plus_expr> "+" <multiply_expr>
| <plus_expr> "-" <multiply_expr>

<multiply_expr> ::= <expression_value>
| <multiply_expr> "*" <expression_value>
| <multiply_expr> "/" <expression_value>

<expression_value> ::= <float>
| <integer>
| "(" <expression> ")"
| <function_call>

<variable_declaration> ::= <datatype> <variable> "=" <expression> <stmt_end>

<variable_assignment> ::= <variable> "=" <expression> <stmt_end>

<stmt_end> ::= ";"

<statement> ::= <simple_statement> | <compound_statement> | <block> |
<stmt_end>

<statement_list> ::= <statement>*

<simple_statement> ::= <variable_declaration>
| <return_stmt>
| <variable_assignment>

<compound_statment> ::= <for_stmt>
| <while_stmt>
| <if_stmt>

<function_identifier> ::= <identifier>

<function_declaration> ::= <datatype> <function_identifier> "(" <argument_list> ")"
<block>

<argument> ::= <datatype> <variable>

<argument_list> ::= <argument>? | <argument> ("," <argument>)+

<function_call> ::= <function_identifier> "(" <expression_list> ")"

<for_stmt> ::= "for" "(" <statement>? ";" <expression> ";" <statement>? ")"
<statement>

<while_stmt> ::= "while" "(" <expression> ")" <statement>

<if_stmt> ::= "if" "(" <expression> ")" <statement> <else-stmt>?

<else_stmt> ::= "else" <statement>

<return_stmt> ::= "return" <expression> <stmt_end>

<break_stmt> ::= "break" <stmt_end>

<continue_stmt> ::= "continue" <stmt_end>

<block> ::= "{" <statement_list> "}"

<class_identifier> ::= <identifier>

<class_declaration> ::= "class" <class_identifier> "{" <class_block> "}" <stmt_end>

Bilaga 3 Kodexempel

Vi tar ett enkelt program som bara initierar en variabel med ett värde och returnerar värdet av den variabeln.

```
1: int fn() {
2:      return 42;
3: }
4: int main() {
5:      int a = fn();
6:      return a;
7: }
```

Det programmet kan se ut på följande sätt i postfixkod.

”0 4 5 goto exit 0 9 22 goto stacktop 2 - stacktop 1 – reference_value assign_to_reference 1 pop goto 2 pop stacktop 1 – 42 assign_to_reference 0 pop goto”

Observera!
Adresser i koden räknas från och med 0 medan det understa värdet på stacken ligger på index 1.

0. **0** – Detta är en reserverad plats för return att lägga sitt värde i vid senare tillfälle.
   [0]
1. **4** – Detta är en adress som kommer användas senare. Som vi kan se i listan är det adressen till ”exit”.
   [0 4]
2. **5 –** Det här är adressen till den kod som körs när funktionen (main i det här fallet) körs.
   [0 4 5]
3. **goto** – operation för att hoppa dit det som ligger överst på stacken anger. I detta fall 5.
   [0 4]
4. **exit –** programmet avslutas.
   [42]
5. **0** – Reserverad plats för funktionen fn att spara returvärde.
   [0 4 0]
6. **9** – Återhoppsadress för funktionen.
   [0 4 0 9]
7. **22** – Adress till där själv funktionen fn börjar.
   [0 4 0 9 22]
8. **goto** – hoppar till koden för fn på rad 22.
   [0 4 0 9]
9. **stacktop**
   [0 4 42 3]
10. **2**
    [0 4 42 3 2]

11. -

[0 4 42 1]

12. **stacktop**
    [0 4 42 1 4]
13. **1**
    [0 4 42 1 4 1]
14. **-**
    [0 4 42 1 3]
15. **reference_value**
    [0 4 42 1 42]
16. **assign_to_reference**
    [42 4 42]
17. **1**
    [42 4 42 1]
18. **pop**
    [42 4]
19. **goto**
    [42]
20. **2** – kod som ej körs i detta sammanhang.
21. **pop** – kod som ej körs i detta sammanhang.
22. **Stacktop**
    [0 4 0 9 3]
23. **1**
    [0 4 0 9 3 1]
24. **–**
    [0 4 0 9 2]
25. **42**
    [0 4 0 9 2 42]
26. **assign_to_reference** - Värdet på index 2 skrivs över med 42.
    [0 4 42 9]
27. **0**
    [0 4 42 9 0]
28. **pop** – Tar inte bort något eftersom ingen variabel sparats i scopet, men det vet inte kompilatorn så den måste köra pop ändå.
    [0 4 42 9]
29. **goto** – hoppar till adress 9 i koden.
    [0 4 42]

Bilaga 4 Källkod för virtuell maskin

runpostfix

```
#!/usr/bin/env ruby

require 'PostfixParser.rb'
require 'Stack.rb'

if ARGV.length == 1
  filename = ARGV[0]
else
  puts "Wrong number of arguments"
  puts "./runpostfix <filename>"
  exit(1)
end

if not File.exist? filename
  puts "File \"#{filename}\" do not exist"
  exit(1)
end

pps = PostfixParseFile(ARGV[0])
wmStack = Stack.new
returnValue = wmStack.eat(pps)
exit(returnValue[1])
```

PostfixParser.rb

```
class PostfixParser
  def initialize
   @keywords = {
     "+" => :plus,
     "-" => :minus,
     "*" => :multiply,
     "/" => :divide,
     "==" => :equals,
     "<" => :less,
     "not" => :not,
     "and" => :and,
     "or" => :or,
     "goto" => :goto,
     "if" => :if,
     "exit" => :exit,
     "stacktop" => :stacktop,
     "pop" => :pop,
     "assign_to_reference" => :assign_to_reference,
     "reference_value" => :reference_value,
     "delete_reference" => :delete_reference,
     "reference" => :reference,
     "output" => :output,
     "input" => :input,
     "swap" => :swap,
     "true" => true,
     "false" => false}
  end

  def parse_string(str)
    tokens = str.split(/\s+/)

    # get Postfix instructions
    pi = tokens.map{|token|
      if @keywords[token] != nil
        @keywords[token]
      elsif token.match(/^\d+$/)
        token.to_i
      else
        raise "Not a Postfix command: \"#{token}\""
      end
    }
    pi
  end
end

def PostfixParseString(postfix)
  p = PostfixParser.new
  p.parse_string postfix
end

def PostfixParseFile(sourcefile)
  source = open(sourcefile)
  p = PostfixParser.new
  p.parse_string source.read
end
```

Stack.rb

```ruby
class Stack < Hash
  def initialize
    @programindex = 0
    @continueprogram = true
    @stackindex = 0
    @heapindex = 0
    @eaten = [] # Debug values

  end

  def viewlunch
    @eaten
  end

  def eat(program)
    while (@continueprogram)
      code = program[@programindex]
      @eaten << [@programindex, code]

      if @programindex >= program.length || @programindex < 0
        warn "Unexpected end of program, out of programbound!"
        p @eaten
        return self
      end

      @programindex += 1

      if code.class == Symbol
        self.method(code).call
      else # not a function call push value to stack
        self << code
      end
    end
    self
  end

  def reset
    self.clear
    @eaten.clear
    @programindex = 0
    @continueprogram = true
    @stackindex = 0
    @heapindex = 0
  end

  def << (value)
    @stackindex += 1
    self[@stackindex] = value
    self
  end

  def popStack
    value = self.delete(@stackindex)
    @stackindex -= 1
    value
  end

  def stacktop
    self << @stackindex
  end

  def pop
    value = popStack
    while (value > 0)
      popStack
      value -= 1
    end
```

```
end

def plus
  right, left = popStack, popStack
  self << (left + right)
end

def minus
  right, left = popStack, popStack
  self << (left - right)
end

def multiply
  right, left = popStack, popStack
  self << (left * right)
end

def divide
  right, left = popStack, popStack
  self << (left / right)
end

def less
  right, left = popStack, popStack
  self << (left < right)
end

def equals
  right, left = popStack, popStack
  self << (left == right)
end

def goto
   address = popStack
   @programindex = address
end

def if
  truthvalue, address = popStack, popStack
  if not truthvalue then
    self << address
    goto
  end
end

def swap
  value1, value2 = popStack, popStack
  self << value1 << value2
end

def exit
  @continueprogram = false
end

def reference_block(blocksize)
  to = @heapindex - 1
  @heapindex -= blocksize
  for i in (@heapindex..to)
    self[i] = nil
  end
  @heapindex
end

# references
def reference
  blocksize = popStack
  self << reference_block(blocksize)
end

def assign_to_reference
```

```
    value, ref = popStack, popStack
    self[ref] = value
  end

  def reference_value
    ref = popStack
    self << self[ref]
  end

  def delete_reference
    ref = popStack
    self.delete(ref)
  end

  # boolean operators
  def and
    value1, value2 = popStack, popStack
    self << (value2 and value1)
  end

  def or
    value1, value2 = popStack, popStack
    self << (value2 or value1)
  end

  def not
    value = popStack
    self << (not value)
  end

  def input
    self << getc
  end

  def output
    asciiValue = popStack
    print asciiValue.chr
  end
end
```

Bilaga 5 Infixkompilatorn

compileinfix

```
#!/usr/bin/env ruby

require 'InfixParser.rb'

if ARGV.length == 1
  infile = ARGV[0]
  outfile = infile.chomp(File.extname(infile)) + ".pf"
elsif ARGV.length == 2
  infile, outfile = ARGV
else
  puts "Wrong number of arguments"
  puts "./compileinfix <infile> <outfile>"
  puts "alt."
  puts "./compileinfix <infile>"
  exit(1)
end

if not File.exist? infile
  puts "Source file \"#{infile}\" do not exist!"
  exit(1)
end

if File.exist? outfile
  puts "Destination file \"#{outfile}\" already exist!"
  exit(1)
end

i = Iterator.new

i.newDatatype("void")
i.newDatatype("int")
i.newDatatype("bool")
i.newFunctionIdentifier("+",["int", "int"], "int", ["+"], true)
i.newFunctionIdentifier("-",["int", "int"], "int", ["-"], true)
i.newFunctionIdentifier("*",["int", "int"], "int", ["*"], true)
i.newFunctionIdentifier("/",["int", "int"], "int", ["/"], true)
i.newFunctionIdentifier("<",["int", "int"], "int", ["<"], true)
i.newFunctionIdentifier("<=",["int", "int"], "int", ["swap", "<", "not"], true)
i.newFunctionIdentifier("==",["int", "int"], "int", ["=="], true)
i.newFunctionIdentifier("!=",["int", "int"], "int", ["==", "not"], true)
i.newFunctionIdentifier("and",["bool", "bool"], "bool", ["and"], true)
i.newFunctionIdentifier("or",["bool", "bool"], "bool", ["or"], true)
i.newFunctionIdentifier("putchar",["int"], "int",
                        ["stacktop", "reference_value", "output"],
                        true)

ips = InfixParseFile(infile)
postfixCode = labelAddressing(ips.parse(i))
File.open(outfile, 'w') {|f| f.write(postfixCode)}

puts "Wrote file #{outfile}"
exit(0)
```

infixparser

```
require 'rdparse.rb'
require 'InfixNodes.rb'

class InfixParser
  def initialize
    @InfixParser = Parser.new("infix parser") do
      token(/\d+/) {|m| m.to_i}
      token(/\s+/)
      token(/;/) {|m| m}
      token(/<=/) {|m| m}
      token(/>=/) {|m| m}
      token(/\=\=/) {|m| m}
      token(/!\=/) {|m| m}
      token(/[\(\)\{\}]/) {|m| m}
      token(/\=/) {|m| m}
      token(/[\+\-\*\/]/){|m| m}
      token(/".*?[^\\]"/) {|m| m} # matches strings
      token(/' '/) {|m| m} # matches spacecharacter
      token(/'\S{1,2}'/) {|m| m}

      token(/'/) {|m| m}
      token(/</) {|m| m}
      token(/>/) {|m| m}
      token(/\w+/) {|m| m}


      start :program do
        match(:global_declaration_list) {|m| Node::Program.new(m)}
      end

      rule :global_declaration_list do
        match(:global_declaration, :global_declaration_list) {|a, b| [a] + b}
        match(:global_declaration) {|m| [m]}
      end

      rule :global_declaration do
        match(:global_variable_declaration) {|m| m}
        match(:function_declaration) {|m| m}
      end

      rule :function_declaration do
        match(:datatype, :function_identifier,
            "(", :argument_list, ")", :block) {|ret, id, _, arg , _, block|
          Node::FunctionDeclaration.new(id, ret, arg, block)}
      end

      rule :argument_list do
        match(:argument, ",", :argument_list) {|a, _, b| [a] + b}
        match(:argument) {|m| [m]}
        match() {[]}
      end

      rule :argument do
        match(:datatype, :variable) {|d, v| [d,v]}
      end

      rule :block do
        match("{", :statement_list, "}" ) {|_, sl, _|
          Node::Block.new(Node::StatementList.new(sl))}
      end

      rule :statement_list do
        match(:statement, :statement_list) {|a, b| [a] + b}
        match(:statement) {|m| [m]}
      end
```

```
rule :statement do
  match(:variable_declaration, :stmt_end) {|a, b| a}
  match(:assignment_statement, :stmt_end) {|a, b| a}
  match(:return, :stmt_end) {|r, _| r}
  match("break", :stmt_end) {Node::Break.new()}
  match("continue", :stmt_end) {Node::Continue.new()}
  match(:function_call, :stmt_end) {|m, _|
    Node::ExpressionStatement.new(m)}
  match(:while) {|m| m}
  match(:for) {|m| m}
  match(:block) {|m| m}
  match(:if) {|m| m}

end

rule :stmt_end do
  match(";")
end

rule :return do
  match("return", :expression) {|_, e| Node::Return.new(e)}
  match("return") {Node::Return.new(Node::Void.new)}
end

rule :while do
  match("while", "(", :expression, ")", :statement) {|_, _, e, _, s|
    Node::WhileStatement.new(e, s)}
end

rule :for do
  match("for", "(", :for_declaration, ";", :for_expression, ";",
        :for_assignment, ")", :statement) {|_, _, vd, _, ce, _, ie, _, s|
    Node::ForStatement.new(vd, ce, ie, s)}
end

rule :for_declaration do
  match(:variable_declaration) {|m| m}
  match() {false}
end

rule :for_expression do
  match(:expression) {|m| m}
  match() {false}
end

rule :for_assignment do
  match(:assignment_statement) {|m| m}
  match() {false}
end

rule :if do
  match("if", "(", :expression, ")", :statement, "else", :statement) {
    |_, _, e, _, trues, _, falses|
    Node::IfStatement.new(e, trues, falses)}
  match("if", "(", :expression, ")", :statement) {|_, _, e, _, s|
    Node::IfStatement.new(e, s, false)}
end

rule :variable_declaration do
  match(:datatype, :variable, "=", :expression) {|t, v, _, e|
    Node::VariableDeclaration.new(t,v,e, true)}
end

rule :assignment_statement do
  match(:assignment_expr) {|m| Node::ExpressionStatement.new(m)}
end

rule :assignment_expr do
  match(:variable, "=", :expression){|v, _, rh|
    Node::AssignExpression.new(v, rh)}
```

```
end

rule :function_identifier do
  match(:identifier) {|m| m}
end

rule :datatype do
  match(:identifier) {|m| m}
end

rule :variable do
  match(:identifier) {|m| m}
end

rule :expression do
  match(:or_expr) {|m| m}
end

rule :or_expr do
  match(:or_expr, "or", :and_expr) {|lh, _, rh|
    Node::FunctionCall.new("or", [lh, rh])}
  match(:and_expr) {|m| m}
end

rule :and_expr do
  match(:and_expr, "and", :not_expr) {|lh, _, rh|
    Node::FunctionCall.new("and", [lh, rh])}
  match(:not_expr) {|m| m}
end

rule :not_expr do
  match("not", :not_expr) {|_, a| Node::LogicalNot.new(a)}
  match(:comparison_expr) {|m| m}
end

rule :comparison_expr do
  match(:comparison_expr,"<=", :plus_expr) {|lh, _, rh|
    Node::FunctionCall.new("<=", [lh, rh])}
  match(:comparison_expr,">=", :plus_expr) {|lh, _, rh|
    Node::FunctionCall.new("<=", [rh, lh])}
  match(:comparison_expr,"==", :plus_expr) {|lh, _, rh|
    Node::FunctionCall.new("==", [lh, rh])}
  match(:comparison_expr,"!=", :plus_expr) {|lh, _, rh|
    Node::FunctionCall.new("!=", [lh, rh])}
  match(:comparison_expr,"<", :plus_expr) {|lh, _, rh|
    Node::FunctionCall.new("<", [lh, rh])}
  match(:comparison_expr,">", :plus_expr) {|lh, _, rh|
    Node::FunctionCall.new("<", [rh, lh])}

  match(:plus_expr) {|m| m}
end

rule :plus_expr do
  match(:plus_expr, "+", :multiply_expr) {|lh, _, rh|
    Node::FunctionCall.new("+", [lh, rh])}
  match(:plus_expr, "-", :multiply_expr) {|lh, _, rh|
    Node::FunctionCall.new("-", [lh, rh])}
  match(:multiply_expr) {|m| m}
end

rule :multiply_expr do
  match(:multiply_expr, "*", :expression_value){|lh, _, rh|
    Node::FunctionCall.new("*", [lh, rh])}
  match(:multiply_expr, "/", :expression_value){|lh, _, rh|
    Node::FunctionCall.new("/", [lh, rh])}
  match(:expression_value) {|m| m}
end

rule :expression_value do
```

```
      #match(:assignment_expr) {|m| m}
      match("(", :expression, ")") {|_, m, _| m}
      match("true") {Node::Boolean.new(true)}
      match("false") {Node::Boolean.new(false)}
      match(Fixnum)  {|m| Node::Integer.new(m)}
      match(:function_call) {|m| m}
      match(:char) {|m| m}
      match(:variable) {|m| Node::PushVariable.new(m)}
    end

    rule :char do
      match(/'\S'/) {|c| Node::Integer.new(c[1])}
      match("' '") {Node::Integer.new(?\s)}
      match("'\\n'") {Node::Integer.new(?\n)}
      match("'\\r'") {Node::Integer.new(?\r)}
      match("'\\t'") {Node::Integer.new(?\t)}
      match("'\\0'") {Node::Integer.new(?\0)}
    end

    rule :function_call do
      match(:identifier, "(", :function_call_arguments, ")") {|id, _, as, _|
        Node::FunctionCall.new(id, as)}
      match(:identifier, "(", ")") {|id, _, _|
        Node::FunctionCall.new(id, [])}
    end

    rule :function_call_arguments do
      match(:expression, ",", :function_call_arguments) {|e1, _, es|
        [e1] + es}
      match(:expression) {|m| [m]}
      match() {[]}
    end


    rule :identifier do
      match(/\w+/) {|m| m}
    end
  end
end

def parse_string(str)
  @InfixParser.parse str
end
end


def InfixParseString(infix)
  p = InfixParser.new
  p.parse_string infix
end

def InfixParseFile(sourcefile)
  source = open(sourcefile)
  p = InfixParser.new
  p.parse_string source.read
end
```

InfixNodes.rb

```
class Operand
  attr_accessor :datatype, :variableName
  def initialize(datatype, variableName=nil)
    @datatype = datatype
    @variableName = variableName
  end
end

class FunctionIdentifier
  attr_accessor :id, :argumentTypes, :returnType, :functionBlock, :inline
  attr_accessor :address
  def initialize(id, argumentTypes, returnType, functionBlock, inline)
    @id = id
    @argumentTypes = argumentTypes
    @returnType = returnType
    @functionBlock = functionBlock
    @inline = inline
    args = argumentTypes.join(",")
    @address = Address.new("#{id}(#{args})")
  end
end

# Iterates over the nodes, validates the node constructions and
# generates the postfix code
class Iterator
  attr_accessor :functions, :datatypes, :stack, :returnType
  def initialize
    @functions = {}
    @datatypes = []
    @stack = []
    @globalvariables = []
    @uniqueid = 0
    @continues = []
    @breaks = []
    @returnType = nil
  end

  def pushScope
    @stack.unshift([])
  end

  def popScope
    @stack.shift.size
  end

  def pushOperand(item)
    if item.class != Operand
      raise "Can't push #{item.class}, must be an Operand"
    end

    if not validDatatype(item.datatype)
      raise "Datatype #{item.datatype} not defined"
    end
    @stack.first.unshift(item)
  end

  def popOperand
    @stack.first.shift
  end

  def bindTopToVariable(name)
    @stack.first.each {|i|
      if i.variableName == name
        raise "Variable #{name} already defined in current scope"
      end
    }
```

```
  @stack.first.first.variableName = name
end

def pushGlobal(item)
  @globalvariables << item
end

def newFunctionIdentifier(id, argumentTypes, returnType, functionBlock,
                          inline)
  at = argumentTypes.join(",")

  if @functions[[id, argumentTypes]] and
     @functions[[id, argumentTypes]].functionBlock != []
    raise "Function already defined: #{id}(#{at})"
  end

  fnLabel, fnAddress = getGotoIds("#{id}(#{at})")
  if not inline and functionBlock != []
    functionBlock.unshift(fnLabel)
  end
  @functions[[id, argumentTypes]] = FunctionIdentifier.new(id,
                                                           argumentTypes,
                                                           returnType,
                                                           functionBlock,
                                                           inline)
end

def newDatatype(name)
  if @datatypes.include?(name)
    raise "Datatype #{name} already defined"
  end
  @datatypes << name
end

def validDatatype(name)
  @datatypes.include?(name)
end

def findFunctionIdentifier(name, args)
  @functions[[name, args]]
end

def getVariable(name)
  index = nil
  flat = stack.flatten
  flat.each_index {|i|
    if flat[i].variableName == name
      index = i
    end
  }
  if not index
    nil
  else
    [flat[index], index]
  end
end

def getGlobalVariable(name)
  index = nil
  @globalvariables.each_index {|i|
    if @globalvariables[i].variableName == name
      index = i
    end
  }
  if not index
    nil
  else
    [@globalvariables[index], index]
  end
end
```

```
  def getStackDepth
    @stack.flatten.size
  end

  def pushContinueAddress(address)
    @continues << [getStackDepth, address]
  end

  def pushBreakAddress(address)
    @breaks << [getStackDepth, address]
  end

  def popStackedAddresses
    @continues.pop
    @breaks.pop
  end

  def topContinueAddress
    @continues.last
  end

  def topBreakAddress
    @breaks.last
  end

  def getUniqueId
    @uniqueid += 1
  end

  def getGotoIds(id=getUniqueId)
    [Label.new(id), Address.new(id)]
  end

end

# Registers all Label locations and replaces all Address instances with
# a correct absolute address.
def labelAddressing(preprogram)
  preprogram.flatten!
  labels = {}
  addresses = {}

  i = 0
  while(i < preprogram.size)
    if preprogram[i].class == Label
      labels[preprogram.delete_at(i).id] = i
      next
    elsif preprogram[i].class == Address
      if addresses.has_key?(preprogram[i].id)
        addresses[preprogram[i].id] << i
      else
        addresses[preprogram[i].id] = [i]
      end
    end
    i += 1
  end

  addresses.each_pair{|key, value|
    value.each{|i|
      if preprogram[i]
        preprogram[i] = labels[key]
      else
        raise "Label #{i} is missing"
      end
    }
  }
  preprogram.join(" ")
end
```

```
# Label is a marker placed within a postfix code, used inconjunction
# with Address.
class Label
  attr_accessor :id
  def initialize(id)
    @id = id
  end
end

# Address is a place holder for addresses, used inconjunction with
# Label.
class Address
  attr_accessor :id
  def initialize(id)
    @id = id
  end
end

# Namespace for nodes to avoid conflicts
module Node
  # Program is the root node, it's make sure the program jumps to the
  # main function.
  class Program
    def initialize(globals)
      @globals = globals
    end

    def parse(iter)
      programreturn = [0, Address.new(:endprogram), Address.new("main()"),
                       "goto", @globals.map{|s| s.parse(iter)},
                       Label.new(:endprogram), "exit"]
      iter.functions.each_value{|fid|
        if not fid.inline
          programreturn << fid.functionBlock
        end
      }
      programreturn
    end
  end

  # VariableDeclaration generates code that places the expression
  # result on the stack and associate it with a variable name.
  class VariableDeclaration
    def initialize(datatype, variable, expression, local)
      @datatype = datatype
      @variable = variable
      @expression = expression
      @local = local
    end

    def parse(iter)
      ep = @expression.parse(iter)
      e = iter.popOperand

      if not iter.validDatatype(@datatype)
        raise "Undefined datatype: #{@datatype}"
      end

      if @datatype != e.datatype
        raise "Incompatable datatypes: #{@datatype} and #{e.datatype}"
      end

      if @local
        iter.pushOperand(e)
        iter.bindTopToVariable(@variable)
      else
        iter.pushGlobal(e)
      end
      ep
    end
```

```
end

# FunctionDeclaration associate the function code with the function
# identifier. The generated function code expects that the arguments
# is located on the top of the stack, with a reserved slot for
# return value and a return address.
class FunctionDeclaration
  def initialize(id, returntype, argumentlist, block)
    @id = id
    @returntype = returntype
    @argumentlist = argumentlist
    @block = block
  end

  def parse(iter)
    iter.returnType = @returntype
    typelist = @argumentlist.map{|arg| arg[0]}
    typeliststring = typelist.join(" ")

    if @id != "main"
      label = Label.new("#{@id}(#{typeliststring})")
    else
      if @argumentlist.size != 0
        raise "Main function argument list should be empty"
      end
      label = Label.new(:main)
    end

    iter.newFunctionIdentifier(@id, typelist, @returntype, [], false) #

    iter.pushScope
    @argumentlist.map{|arg|
      iter.pushOperand Operand.new(arg[0], arg[1])
    }

    iter.pushOperand Operand.new(@returntype, :returnValue)
    iter.pushOperand Operand.new("int", :returnAddress)

    programreturn = @block.parse(iter)

    iter.popScope

    iter.newFunctionIdentifier(@id, typelist, @returntype, programreturn,
                               false) #
    iter.returnType = nil
    []
  end
end

# Block creates a new local scope in the stack, making it possible
# to override a variable. At the end of the scope it removes the
# variables declared in the scope.
class Block
  def initialize(statementlist)
    @statementlist = statementlist
  end

  def parse(iter)
    iter.pushScope
    programreturn = @statementlist.parse(iter)

    popsize = iter.popScope

    programreturn + [popsize, "pop"]
  end

end

class StatementList
  def initialize(list)
```

```
    @list = list
  end

  def parse(iter)
    @list.map{|s| s.parse(iter)}
  end
end

# IfStatement generates code jumping to truebrach or falsebranch.
class IfStatement
  def initialize(expression, truebranch, falsebranch=nil)
    @expression = expression
    @truebranch = truebranch
    @falsebranch = falsebranch
  end

  def parse(iter)
    falseLabel, falseAddress = iter.getGotoIds
    endLabel, endAddress = iter.getGotoIds

    iter.pushOperand(Operand.new("int")) # push false address to stack
    programreturn = [falseAddress]       #
    programreturn << @expression.parse(iter)
    programreturn << "if"
    iter.popOperand
    iter.popOperand
    programreturn << @truebranch.parse(iter)
    iter.pushOperand(Operand.new("int")) # push end address to stack
    programreturn << endAddress          #
    programreturn << "goto"
    iter.popOperand
    programreturn << falseLabel
    if @falsebranch
      programreturn << @falsebranch.parse(iter)
    end
    programreturn << endLabel
    programreturn
  end
end

# WhileStatement generates code that corresponds to a while-loop.
class WhileStatement
  def initialize(expression, statement)
    @expression = expression
    @statement = statement
  end

  def parse(iter)
    startLabel, startAddress = iter.getGotoIds
    endLabel, endAddress = iter.getGotoIds

    iter.pushContinueAddress(startAddress)
    iter.pushBreakAddress(endAddress)

    programreturn = [startLabel]
    iter.pushOperand(Operand.new("int")) # push end address to stack
    programreturn << endAddress          #
    programreturn << @expression.parse(iter)
    programreturn << "if"
    iter.popOperand
    iter.popOperand
    programreturn << @statement.parse(iter)
    iter.pushOperand(Operand.new("int")) # push start address to stack
    programreturn << startAddress        #
    programreturn << "goto"
    iter.popOperand
    programreturn << endLabel
    iter.popStackedAddresses
    programreturn
  end
```

```
end

# WhileStatement generates code that corresponds to a for-loop.
class ForStatement
  def initialize(declaration, continueexpr, iterationexpr, statement)
    @declaration = declaration
    @continueexpr = continueexpr
    @iterationexpr = iterationexpr
    @statement = statement
  end

  def parse(iter)
    startLabel, startAddress = iter.getGotoIds
    endLabel, endAddress = iter.getGotoIds
    continueLabel, continueAddress = iter.getGotoIds
    breakLabel, breakAddress = iter.getGotoIds

    programreturn = []
    iter.pushScope
    iter.pushBreakAddress(breakAddress)
    if @declaration
      programreturn << @declaration.parse(iter)
    end

    iter.pushContinueAddress(continueAddress)

    programreturn << startLabel
    iter.pushOperand(Operand.new("int")) # push end address to stack
    programreturn << endAddress         #
    if @continueexpr
      programreturn << @continueexpr.parse(iter)
    else
      programreturn << "true"
    end
    programreturn << "if"
    iter.popOperand
    iter.popOperand
    programreturn << @statement.parse(iter)

    programreturn << continueLabel
    if @iterationexpr
      programreturn << @iterationexpr.parse(iter)
    end
    iter.pushOperand(Operand.new("int")) # push start address to stack
    programreturn << startAddress       #
    programreturn << "goto"
    iter.popOperand
    popdepth = iter.popScope
    programreturn << endLabel
    programreturn << popdepth
    programreturn << "pop"
    programreturn << breakLabel
    iter.popStackedAddresses
    programreturn
  end
end

# Generates code that stores expression result to the reserved
# return value slot, removes all variables in the function scope and
# consume the return address on the stack.
class Return
  def initialize(expression)
    @expression = expression
  end

  def parse(iter)
    # args, ret_val, ret_addr
    depth = iter.getStackDepth - 2
    operand, rindex = iter.getVariable(:returnValue)
```

```
    iter.pushOperand Operand.new("int")
    e = @expression.parse(iter)
    item = iter.popOperand
    if item.datatype != iter.returnType
      raise "Missmatched return type expected \"#{iter.returnType}\", " +
        "was \"#{item.datatype}\""
    end

    ["stacktop", rindex, "-", e, "assign_to_reference", rindex - 1, "pop",
     "goto"]
    # args, ret_val
  end
end

# Removes all declared variables sinse the loopstart, then jumps to
# the end of the loop.
class Continue
  def initialize()
  end

  def parse(iter)
    previousDepth, address = iter.topContinueAddress
    [iter.getStackDepth - previousDepth, "pop", address, "goto"]
  end
end

# Generates code that removes all declared variables sinse the
# loopstart, then exiting the loop.
class Break
  def initialize()
  end

  def parse(iter)
    previousDepth, address = iter.topBreakAddress
    [iter.getStackDepth - previousDepth, "pop", address, "goto"]
  end
end

# Generates code that removes the value that an expression leaves on
# the stack.
class ExpressionStatement
  def initialize(ae)
    @ae = ae
  end

  def parse(iter)
    programreturn = [@ae.parse(iter)]
    iter.popOperand
    programreturn << 1 << "pop"
    programreturn
  end
end

# Generates code for assigning a value to a variable and leaves a
# copy of the value on the stack.
class AssignExpression
  def initialize(variableId, rh)
    @variableId = variableId
    @rh = rh
  end

  def parse(iter)

    rh = @rh.parse(iter)
    programreturn = [rh]

    item, index = iter.getVariable(@variableId)
    if not item
      raise "Undefined variable: #{@variableId}"
    end
```

```
    programreturn << "stacktop" << index << "-"
    programreturn << "stacktop" << 1 << "-" << "reference_value"
    programreturn << "assign_to_reference"
    programreturn

  end
end

# FunctionCall generate code that places the arguments results on the stack.
# If the function is a non-inline function its also reserves a slot for the
# return value and the return address, then jumps to the function code. When
# returning from the function, FunctionCall expects the return address is
# consumed and that the return value is stored in reserved slot, removes
# arguments results and places the return value on top.
# If the function is a inline it paste the function code after the arguments
# code and expects that the function code removes the arguments results and
# leaves a result on the stack.
class FunctionCall
  def initialize(id, argumentNodes)
    @id = id
    @argumentNodes = argumentNodes
  end

  def parse(iter)
    programlist = @argumentNodes.map{|n|
      n.parse(iter)
    }
    #[args1..argsN]

    #get the operand types from the iterator stack
    operands = []
    1.upto(@argumentNodes.size) {
      operands << iter.popOperand
    }

    datatypes = operands.reverse.map{|o|
      o.datatype
    }

    functionId = iter.findFunctionIdentifier(@id, datatypes)

    typeList = datatypes.join(",")

    if not functionId
      raise "FunctionId not declared: #{@id}(#{typeList})"
    end
    #[args1..argsN]
    if functionId.inline
      programlist << [functionId.functionBlock]
      #[ret_val]
    else
      returnValue = Operand.new("void") #reserve return value slot

      iter.pushOperand(returnValue)
      iter.pushOperand(Operand.new("int"))  #return address

      fnAddress = functionId.address
      returnLabel, returnAddress = iter.getGotoIds


      if @argumentNodes.size == 0
        #[ret_val, ret_add fn_add]
        programlist << [0, returnAddress, fnAddress, "goto", returnLabel]
      else

        programlist << [0, returnAddress, fnAddress, "goto"]
        #[args,  ret_val, ret_add]
```

```
      #[arg1, arg2, ret_val] =>          #[ret_val, arg2]
      programlist << [returnLabel,
                "stacktop", @argumentNodes.size, "-", "swap",
                "assign_to_reference", @argumentNodes.size - 1, "pop"]
    end
    iter.popOperand
    iter.popOperand
  end


  iter.pushOperand(Operand.new(functionId.returnType))
  programlist
 end
end

# Generates code that copyes variable value to the top of the stack.
class PushVariable
 def initialize(name)
  @name = name
 end

 def parse(iter)
  item, index = iter.getVariable(@name)
  if item
   iter.pushOperand(Operand.new(item.datatype))
   ["stacktop", index, "-", "reference_value"]
  else
   item, index = iter.getGlobalVariable(@name)

   if item
    iter.pushOperand(Operand.new(item.datatype))
    [index, "reference_value"]
   else
    raise "Undefined variable: #{@name}"
   end
  end
 end
end

# Integer generates code that leaves an integer value on the stack.
class Integer
 def initialize(value)
  @value = value
 end

 def parse(iter)
  iter.pushOperand(Operand.new("int"))
  @value
 end
end

# Boolean generates code that leaves a boolean value on the stack.
class Boolean
 def initialize(value)
  @value = value
 end

 def parse(iter)
  iter.pushOperand(Operand.new("bool"))
  if(@value)
   "true"
  else
   "false"
  end
 end
end

# Void generates code that leaves a value on the stack. (It only
# used when a function returns a void, to solve a special case of
# "return;")
```

```
  class Void
    def parse(iter)
      iter.pushOperand(Operand.new("void"))
      0
    end
  end
end
```