2017

Stijn Klarenbeek
Frank Grevelink

COMPILERS EN OPERATING SYSTEMS

Inhoud

Voorwoord

Door de jaren heen zijn er duizenden programmeertalen ontworpen en ontwikkeld.
Een aantal daarvan zijn redelijk bekend (Java, C++, Javascript, PHP, etc.), vele andere heb je nog nooit mee gewerkt of nog nooit van gehoord (Haskell, Algol, Prolog, Brainfuck, Lisp, Python, Piet, Whitespace, FALSE, LOLCODE,Basic, Ruby, Pascal).

Voor deze opdracht ga je, je eigen programmeertaal ontwikkelen die vertaald kan worden naar Java Bytecode zodat deze kan worden gelezen door de JVM.

Iedere programmeertaal bevat bepaalde functies. Aan de volgende eisen moest minimaal voldaan worden:

1. Minimaal 2 datatypen.
2. Ondersteuning voor variabelen.
3. Rekenkundige expressies.
4. Logische expressies.
5. Conditional branching (bijv: if statements).
6. Looping statements (while/for).
7. Functie met parameters en return type.
8. Ondersteuning voor globale variabelen.
9. Ondersteuning voor het printen van expressies en variabelen naar de consoles.
10. Ondersteuning voor het lezen van gebruikersinvoer.

Interessante features

Hieronder staan een paar interessante features die onze taal bevat, deze worden verderop uitgelegd:

- Het direct kunnen schrijven van een (+=, /=, *=, -=) expressie die tegelijk deze waarde in de variabele opslaat.
- Het gemakkelijk lezen van gebruikers invoer.
- For en while loops zijn beide geïmplementeerd.
- If-else statements kunnen in elkaar worden gezet.
- Het direct kunnen wegschrijven van een conditie.
- Ondersteuning voor variabel toekenning (kan d.m.v. methode).
- Het kunnen veranderen van een variabele waarde (kan ook d.m.v. methode).
- Ondersteuning voor de volgende typen: leegte, draad, nummer, vliegend.
- Ondersteuning voor globale variabelen (alleen niet voor methodes).
- Declarering van variabelen zonder deze toe te kennen.

Niet geïmplementeerde functies

De volgende functies zijn niet geïmplementeerd:

- De OR, AND en NOT logische expressies zijn niet geïmplementeerd vanwege tijdgebrek.

Ontwerp van de taal

Onze taal is zeer simpel ontworpen, dit om te zorgen dat je zo min mogelijk code nodig hebt om toch veel te kunnen doen. Bij onze taal moeten eerst alle functies aangemaakt worden, voordat je verder kan programmeren, waarom dit zo is, lees je in “implementatie issues”.

Het inlezen van gebruikersinvoer

Om gebruikersinvoer in te kunnen lezen moeten de volgende twee dingen gelden:

1. Er moet een variabele bestaan waarin je het wil inlezen
2. De datatypen moeten overeenkomen.

Voorbeeldcode

```
draad stijn;
stijn < (draad);
```

Methode uitvoering

Voor het uitvoeren van een methode zijn er ook een paar regels, zoals het feit dat return types niet mogen bij een “leegte” functies. Verder moeten de datatypes van het gereturnde en de functie overeenkomen.

Voorbeeldcode

```
leegte functie ikPraat(draad gepraat, nummer aantalKeerGepraat){
   nummer index = 0;
   zolang(index < aantalKeerGepraat){
       schrijf gepraat;
       index++;
   }
}
```

Declarering van een variabele

Voor het declareren van een variabele geldt er iets heel belangrijks, namelijk het feit dat een ID niet mag beginnen met een cijfer. Dit laat onze grammatica niet toe.

Voorbeeldcode

```
draad stijn;
```

Toekenning van een variabele

Voor het toekennen van een variabele geldt hetzelfde als hierboven en moet hetgene overeenkomen met het type variabele

draad	strings, bijv: “blabla”
nummer	Hele getallen zoals: 20
vliegend	Komma getallen zoals: 20,5
booleaans	waar of onwaar

Voorbeeldcode

```
draad stringNaam = "Ik ben een string";
nummer nummerGetal = 20;
vliegend vliegendGetal = 20,5;
booleaans booleaansDingetje = onwaar;
```

Verandering van een variabele

Voor het veranderen van een variabele geldt dat de datatypes moeten overeenkomen zoals hierboven en dat de variabele bestaat.

Voorbeeldcode

```
draad stijn;
stijn = "Ik ben stijn";
```

Toekenning van een variabele d.m.v. een methode

Voor het toekennen van een variabele d.m.v. van een methode gelden er een aantal dingen. Zoals hetgene wat boven staat beschreven bij methode uitvoering. En het datatype dat gereturned wordt moet overeenkomen met de variabele.

Als er parameters worden meegegeven aan de methode, moet je ervoor zorgen dat de hoeveelheden correct zijn en dat de types verder overeenkomen. Als je een variabele meegeeft als parameter moet deze bestaan. Dit geldt bij alle aanroepingen van methodes, dus ook bij verandering en normale aanroepen.

Voorbeeldcode

```
draad functie geefTerugFunctie(){
      draad string = "Terug gegeven draad";
      geefterug string;
}
draad geefTerug = geefTerugFunctie();
```

Verandering van een variabele d.m.v. een methode

Hier gelden dezelfde regels als bij “toekenning van een variabele d.m.v. een methode” alleen bestaat hier de variabele al.

Voorbeeldcode

```
draad functie geefTerugFunctie(){
      draad string = "Terug gegeven draad";
      geefterug string;
}
draad geefTerug = "draad";
geefTerug = geefTerugFunctie();
```

Methode aanroep

Hier gelden dezelfde regels als bij eerder “toekenning van een variabele d.m.v. toekenning” met het verschil dat return types hier niets uitmaakt.

Voorbeeldcode

```
ikPraat("Dit is een leuk praatje",30);
```

Als statements

Onze compiler biedt alleen ondersteuning voor vergelijking van getallen. Wel zou je een booleaans kunnen aanmaken en dat laten vergelijken met een 0 of 1, maar er is geen ondersteuning voor: als(waar){ //etc }.

Voorbeeldcode

```
nummer getal1;
getal1 < (nummer);

nummer getal2;
getal2 < (nummer);

als(getal1 < getal2){
      schrijf "getal 1 is kleiner dan getal 2";
}alsanders (getal1 == getal2){
      schrijf "getal 1 is gelijk aan getal 2";
}anders{
      schrijf "getal 1 is groter dan getal 2";
}
```

Voor loop

Voor de variabele binnen de voor loop kunnen de volgende dingen gelden. Het kan bijvoorbeeld slechts een verandering van een al bestaande variabele zijn of een toekenning van een nieuwe. Hieronder staan twee voorbeelden om te demonstreren wat er wordt bedoeld. Verder geldt voor de vergelijking van de getallen hetzelfde als bij “als statements”.

Voorbeeldcode met toekenning

```
voor(nummer i = 0; i < 20;i++){
      schrijf "Ik ben een werkende programma";
}
```

Voorbeeldcode met verandering

```
nummer i = 20;
voor(i = 0; i < 20;i++){
      schrijf "Ik ben een werkende programma";
}
```

Zolang loop

Voor de vergelijking van de getallen hetzelfde als bij “als statements”.

Voorbeeldcode

```
nummer zolangNummer;
zolangNummer < (nummer);
zolang(zolangNummer > 0){
```

```
      schrijf "\r\nAantal keer geschreven ";
      schrijf zolangNummer;
      zolangNummer--;
}
```

Print statement

Binnen onze taal kunnen er verscheidene dingen naar de console geprint worden, zoals: een conditie, een string, een variabele en een rekenkundige expressie.

Voorbeeldcode

```
@@Defined variable printing
nummer getal1 = 12-3;
nummer getal2 = 12*12;

nummer getal6 = 20;

schrijf "Getal 1 (12-3 reken expressie)";
schrijf getal1;

schrijf "\r\nGetal 2 (12*12 reken expressie)";
schrijf getal2;

@@ Straight print
schrijf "\r\n20 kleiner dan 30:";
schrijf 20<30;

@@ Schrijf rekenkundige expressies

schrijf "\r\n20%15=";
schrijf 20%15;

@@ schrijf expressies die opgeslagen worden
schrijf "\r\n20*=10";
schrijf getal6*=10;
```

De bovenstaande dingen zijn slechts voorbeelden, er kan nog meer geprint worden. Voor alle mogelijkheden, bekijk WorkingFile5.djpp in het mapje “Werkende programma’s”.

Code checker

In onze implementatie zijn de scope checker en type checker in één gebouwd. Dit was voor ons een stuk makkelijker en efficiënter. Hierdoor hoef je niet verschillende keren de scope op te halen om een paar dingen te checken.

Binnen de code checker wordt gebruik gemaakt van een enum genaamd Type. Deze bevat verschillende soorten types, een methode om een type te krijgen op basis van een string en een methode om een mnenonic op te halen op basis van type.

De code checker is een visitor class die allemaal visitor methodes heeft. Sommige methodes returnen een bepaald type voor het type check gedeelte. In iedere methode wordt de current scope opgeslagen. Verder visit iedere methode zijn kinderen wanneer dit moet. In bepaalde methodes wordt de scope stack opgehoogd, dit om te berekenen hoe groot je uiteindelijke stack moet zijn (werkt niet optimaal, stack vaak te groot).

Methodes
Als eerst worden alle methodes bezocht (omdat deze altijd bovenaan in de file staan) met statements daar binnen en daarna komen pas de losse statements. Dit is gedaan zodat je bij het aanroepen van een methode, deze al hebt en dan kun je makkelijker vergelijkingen doen op basis van parameters en namen (eventueel return types).

Toekenning van een variabele
Bij een toekenning van een variabele wordt eerst gekeken of hij al bestaat in deze scope of een ouder scope, als dat zo is komt er een foutmelding. Vervolgens worden de typen vergeleken en indien correct, wordt de variabele opgeslagen. Als laatst wordt een positie toegevoegd aan de variabele op basis van de positie die binnen die scope staat gedeclareerd. Dit om een zo'n efficiënt mogelijk aantal locals te hebben met de correcte variabele posities binnen de code generator. Ook wordt de variabele toegevoegd aan de "varMethTree". Deze parsetree houdt binnen bepaalde methodes bij, welke methode of variabele is aangemaakt.

Verandering van een variabele
Bij de verandering van een variabele wordt gekeken of hij wel bestaat. Als dat niet zo is, kijkt hij of de types overeenkomen. Als dit allemaal klopt slaat hij ook hier de variabele op in de varMethTree.

Toekenning en verandering van variabele d.m.v. methode
Bij toekenning en verandering van variabelen door middel van methodes wordt gekeken naar eerder genoemde regels. Zo moet het aantal parameters dus overeenkomen met het aantal dat bij de uitvoering van een methode is aangevraagd en moeten de typen overeenkomen. Ook moet de methode daadwerkelijk bestaan (net zoals bij methode aanroep).

Rekenkundige expressies
Bij het visiten van rekenkundige expressies wordt gekeken of de types wel een INT of een FLOAT zijn. Op het moment dat beide getallen een INT zijn, wordt er een nieuwe value

expressie aangemaakt met het type INT. Deze stopt hij in een parsetree genaamd “valExprTypeTree” en returned hij het type INT. Op het moment dat één van de twee een float is, wordt er een value expressie aangemaakt met het type FLOAT en toegevoegd aan diezelfde tree en returned de methode een FLOAT.

Bij sommige rekenkundige expressies zoals: expr--, expr++ en expr+=expr. Moet aan de linkerkant altijd een variabele staan die bestaat. Hier wordt ook op gechecked, in deze methodes worden de linker variabelen ook toegevoegd aan de varMethTree.

Conditionele expressies
Bij conditionele expressies wordt gekeken of het linkertype en rechter type, een INT of een FLOAT of een BOOL zijn. De reden voor BOOL is omdat we toen nog met && expressies bezig waren die er uiteindelijk niet zijn in gekomen. Als de types kloppen returned hij een BOOL.

Als statements
Bij een als statement wordt een nieuwe scope geopend. Hieraan wordt een naam gegeven. Vervolgens wordt de ouder opgehaald van de scope en daar de nieuwe scope aan toegevoegd. Vervolgens lopen we door alle conditie blokken heen waar de bovenstaande conditionele expressies weer van toepassing komen. Als een conditie block niet in het type BOOL resulteert, komt er een InvalidTypeException. Vervolgens wordt er gekeken of er een “anders” blok bestaat en hier door heen gelopen. Elke keer wordt er een counter opgehoogd voor de als statements zodat iedere scope een eigen naam heeft. Dit is belangrijk voor de code generator. Als laatst wordt de scope geclosed en wordt de parent dus de current scope.

While loop
Bij een while statement gebeurd precies hetzelfde als bij een als statement.

Voor loop
Een voor statement is een combinatie van de while statement en het toekennen of veranderen van een variabele.

Schrijf statement
Bij een schrijf statement hoeft alleen bij het schrijven van een variabele gekeken te worden of deze bestaat. Ook wordt hier die variabele opgeslagen in de parsetree.

Gebruikersinvoer lezen
Bij het lezen van gebruikersinvoer wordt gekeken of de datatypes overeenkomen en of de variabele waar het aan wordt toegekend, daadwerkelijk bestaat. Ook wordt hier de variabele toegekend aan de varMethTree.

Verder zijn er nog kleine visitors die alleen hun eigen type teruggeven. Zoals de visitAtomString die het type STRING teruggeeft en de visitAtomInt die het type INT teruggeeft.

Code generation

De code generator kan op meerdere manieren worden aangepakt. Wij hebben ervoor gekozen om een visitor te maken waar alles naar een Jasmin file wordt geprint. In de code generator worden de parsetrees meegegeven waaruit de nodige data kan worden gehaald.

Toekenning/verandering variabelen
Voor toekenning van variabelen wordt eerst de data weggeschreven naar de stack die je wilt opslaan en vervolgens wordt deze dan opgeslagen in een locale variabele. Het eventuele pushen naar de stack en vervolgens opslaan wordt gedaan op basis van het type en positie. Ditzelfde geldt voor het veranderen van een variabele.

Voorbeeld
```
bipush 12 ;Push naar stack
bipush 3  ;Push naar stack
isub      ;Haal van elkaar af
istore 1  ;Store naar een positie
```

Methode aanroep
Bij het aanroepen van een methode wordt eerst het “this” object geladen. Daarna worden alle variabelen naar de stack gepushed en wordt de methode aangeroepen door middel van invoke. Voor methode aanroepen gebruiken wij invokevirtual.

Voorbeeld
```
aload 0 ;Laad het this object
ldc "Dit is een leuk praatje" ;Push het eerste object naar de stack
bipush 30 ; Push tweede object naar de stack
invokevirtual WorkingFile2/ikPraat(Ljava/lang/String;I)V ;Aanroep
```

Toekennen en veranderen d.m.v. methode
Voor het toekennen en veranderen van variabelen door middel van methode maak je gebruik van een combinatie van bovenstaande elementen. Eerst push je alles naar de stack en roep je de methode aan. Vervolgens sla je het op op basis van positie en datatype.

Voorbeeld
```
aload 0 ;Laad het this object
invokevirtual WorkingFile4/geefTerugFunctie()Ljava/lang/String; ;Aanroep
astore 4 ;Store resultaat in object
```

++ en -- rekenkundige expressie
Voor -- en ++ expressies gebruiken wij iinc voor INTs en fsub/fadd voor FLOATs. Iinc pakt een getal op basis van positie en increased hem met 1 of -1 (ligt eraan wat jij wilt).

Voorbeeld
```
iinc 2 -1 ;Increase positie 2 met -1
```

+=, *=, /=, -= rekenkundige expressies

Voor += expressies laden wij eerst het linker deel op de stack. Dan kijken wij wat voor type value expressie het is. Als de value expressie een FLOAT is en het geladen getal een INT, zetten wij deze eerst om naar een FLOAT. Daarna gebeurd hetzelfde met de rechterkant. Vervolgens kijken we op basis van de operator wat voor een operatie moet worden gebeuren en voeren deze uit. Daarna wordt het getal opgeslagen op basis van de positie van het linker kind van de expressie.

Voorbeeld

```
iload 6 ;Laad de variabele
bipush 10 ; Push 10 naar de stack
imul ; Keer operatie
istore 6 ;Store het terug naar de variabele
```

Overige rekenkundige expressies

Voor de andere reken expressies maakt het niet uit of het linker kind een variabele is of niet. Dan wordt naar de stack gepushed op basis van datatype en wordt op basis van de operator de correcte operatie uitgevoerd. Ook hier is de value expressie aanwezig om eventueel het linker of rechter kind om te zetten naar een float.

Voorbeeld

```
bipush 12 ;Push 12 naar stack
bipush 3 ;Push 3 naar stack
isub ;Haal ze van elkaar af
```

Als statements

Voor als statements komen de namen van de scopes weer te pas. Een scope naam wordt dan gebruikt als label waar naar verwezen kan worden. Deze gebruik je om efficiënter en correct door het statement heen te lopen. Iedere als anders blok heeft zijn eigen naam waar naar wordt verwezen als aan een if statement niet wordt voldaan.
Voorbeeld:

Voorbeeld

```
alsstate_1: ;Eerste als state label
iload 1 ;Laad positie 1 op de stack
iload 2 ;Laad postitie 2 op de stack
if_icmpge alsstate_2 ;Conditie vergelijken
// Als code hier
goto als_1_end ;Ga naar einde van de if statement

alsstate_2:
iload 1
iload 2
if_icmpne alsstate_3
//Als anders code hier
goto als_1_end

alsstate_3:
//Anders code hier
als_1_end:
```

Zoals hierboven is te zien wordt doorverwezen naar het volgende label als niet aan de conditie wordt voldaan. Zoals ook te zien is, gebruiken wij een omgekeerd stelsel voor condities, dit maakt vergelijken makkelijker en maakt het handiger om snel door te verwijzen.

Voor en while loops
Labels gebruiken wij ook voor “zolang” en “voor” loops. Eerst schrijf je het label, dan een vergelijking die verwijst naar een label aan het eind van de loop als de conditie niet voldoet. Voldoet hij nog wel, dan wordt alles uitgevoerd tot hij bij een label aankomt die ervoor zorgt dat hij weer naar het begin gaat. Bij een voorloop wordt ook nog de index opgehoogd (dit gebeurd voordat hij naar het begin springt).

Voorbeeld
```
for_loop_1:      ;Voor loop begin label
iload 1          ; Laadt variabel
bipush 20        ; Push 20 naar de stack
if_icmpge for_loop_1_end ;Check conditie en voer code uit

getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Ik ben een werkende programma"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

iinc 1 1         ; Hoog op de index
goto for_loop_1  ; Ga terug naar het begin

for_loop_1_end:  ; Naar verwezen als conditie niet voldoet
```

Gebruikersinvoer
Voor het inlezen van gebruikersinvoer wordt gebruik gemaakt van de scanner class in Java. Als eerst laden we de variabele in waarnaar geschreven moet worden. Daarna wordt de scanner class aangeroepen en naar de top van de stack gekopieerd. Vervolgens halen we de inputstream op en geven we deze mee aan de scanner. Als laatste wordt dan ingelezen op basis van het datatype.

Voorbeeld (string)
```
aload 1            ;Variabele laden
new java/util/Scanner ;Scanner laden
dup ;push scanner naar top van stack
getstatic java/lang/System/in Ljava/io/InputStream; ;laad inputstream
invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V ;Geef IS mee
invokevirtual java/util/Scanner/nextLine()Ljava/lang/String; ;Lees lijn in
astore 1   ;Store de ingelezen string
```

Schrijven
Er zijn vier verschillende dingen die geschreven kunnen worden. Hieronder zie je een voorbeeld van alle vier.

Voorbeeld string
```
getstatic java/lang/System/out Ljava/io/PrintStream; ;Haal outstream op
```

```
ldc "Getal 1 (12-3 reken expressie)" ;push wat jet wilt zeggen naar stack
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V ;printen
```

Voorbeeld ID

```
getstatic java/lang/System/out Ljava/io/PrintStream; ;Haal outstream op
aload 1 ;Laad de variabele naar de stack
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V ;printen
```

Voorbeeld conditionele expressie

```
bipush 30  ;Push 30 naar stack
bipush 20  ;Push 20 naar stack
if_icmpge schrijfexpr_2_else ;Conditie vergelijking, print uitkomst

getstatic java/lang/System/out Ljava/io/PrintStream;
iconst_1 ;Push 1 naar stack
invokevirtual java/io/PrintStream/println(Z)V ;Print een boolean
goto schrijfexpr_2_end ;Ga naar einde

schrijfexpr_2_else: ;Anders voer dit uit
getstatic java/lang/System/out Ljava/io/PrintStream;
iconst_0
invokevirtual java/io/PrintStream/println(Z)V
goto schrijfexpr_2_end
schrijfexpr_2_end:
```

Voorbeeld rekenkundige expressie

```
getstatic java/lang/System/out Ljava/io/PrintStream; ;Haal output op
iload 6 ;Laad positie 6 op stack
bipush 10 ;Push 10 naar stack
isub ;Haal van elkaar af
istore 6 ;Store naar positie 6 (Dit is een -= expressie)
iload 6 ;laad 6 terug op de stack
invokevirtual java/io/PrintStream/println(I)V ;Printen maar
```

Op deze manier schrijven wij dus alles weg naar de Jasmin files. Wij geven geen lijst met strings terug, maar schrijven alles direct weg.

Implementatie issues

Tijdens het maken van de code zijn we tegen verschillende problemen aangelopen. Hieronder staat een overzicht van deze problemen en hoe ze zijn opgelost.

If-else statements
Wij wisten niet precies hoe je if en else statements moest wegschrijven naar een file. We begrepen dat je labels moest gebruiken maar niet precies hoe. Uiteindelijk hebben wij de manier van Tristan gebruikt.

Jasmin “this”
Wij wisten niet Jasmin een “this” object nodig had voor het aanroepen van methodes binnen onze klasse. Dankzij Tristan weten we dit nu wel en is ook het aanroepen van methodes gelukt.

Scoping
Wij hadden zeer veel verschillende problemen met scoping. Dit kwam doordat wij niet goed de kinderen toevoegden en op verkeerde plekken de scope toevoegden aan de parsetree. Verder maakten we gebruik van onnodige dingen.

Parameter
Wij wisten wisten eerst niet precies hoe we parameters binnen methodes konden checken of ze bestonden. Dit hebben we opgelost door parameters als symbolen binnen de methode toe te voegen als locale variabelen.

Positionering van elementen
Er waren een aantal problemen met het vinden van de correcte positie van elementen. Dit kwam ook voornamelijk omdat wij een hashmap implementatie gebruiken om de symbolen op te slaan. Wij hebben redelijk wat recursieve methoden geprobeerd. Uiteindelijk was het het handigst om per symbool een positie op te slaan.

Aantal locals
Er waren wat problemen met het uitrekenen van de hoogst nodige stack. Dit probleem is door een recursieve methode die alle kinderen van kinderen hun positie ophaalt. Uiteindelijk komt dan de hoogst mogelijke getal terug.

Stack size
Wij hebben nog steeds een te grote stack size naar ons idee, wel is het wat efficiënter dan geen stack berekenaar.