Vertalerbouw

Het verslag moet inzicht geven hoe de taal gedeﬁnieerd is, en hoe de problemen die zich voordeden

bij het maken van de vertaler opgelost zijn. Vermeld ook wie voor welk onderdeel verantwoordelijk is en welke delen samen gemaakt zijn. Het verslag van de practicumopdracht dient in ieder

geval de volgende onderdelen te bevatten:

# Inleiding - Kim

Korte beschrijving van de practicumopdracht.

In dit project wordt een complete vertaler voor een zelf ontworpen programmeertaal ontwikkeld. Dit wordt gedaan met behulp van ANTLR. De zelf ontworpen programmeertaal zal vertaald worden naar Java ByteCode.

Er moet een parser, een lexer en een checker gemaakt worden om de zelf ontworpen taal te kunnen inlezen en te controleren op fouten. Ook moet er een codegenerator gemaakt worden om van de ingelezen informatie Java ByteCode te maken (met behulp van Jasmin).

In dit verslag zal als eerste de zelf ontworpen programmeertaal worden gedefinieerd. Er zal beschreven worden welke problemen zich voordeden en hoe ze zijn opgelost. Daarna wordt uitgelegd wat de syntax, context-beschrijving en semantiek van de programmeertaal zijn. Als volgt worden de vertaalregels voor de taal uiteengezet. Vervolgens wordt er een beschrijving gegeven van Java-programmatuur. Het testplan en de testresultaten worden daarna getoond en als laatste zullen er conclusies neergezet worden ter afsluiting van het project.

# Beknopte beschrijving - Kim

van de programmeertaal (maximaal e´en A4-tje)

De programmeertaal heeft de naam Chocolate omdat de taal veel *begrippen/termen?* bevat die gerelateerd zijn aan chocola. De volgende (chocolade) termen hebben de volgende betekenissen:

reep : constant | bounty : +

bonbon : variable | mars : -

| nuts : !

milka : assign | milkyway : \*

droste : read | dove : /

verkade : print | maltesers : %

| snickers : <

wit : integer | kinder : <=

melk : char | bueno : >=

puur : boolean | lion : >

| kitkat : ==

cacao : comment | bros : !=

| rolo : &&

| twix : ||

Een programma geschreven in Chocolate bestaat uit declaraties en statements, die door elkaar heen gebruikt kunnen worden, er wordt alleen vereist dat een constante of variabele gedeclareerd moet zijn voordat het gebruikt kan worden. Elke declaratie en statement wordt afgesloten met een puntkomma.

Voor het declareren van een constante moet er eerst worden aangegeven dat het gaat om een constante. Dit wordt gevolgd door het type, een identifier (een identifier kan een combinatie zijn van letters en cijfers, maar begint altijd met een letter) en de ‘assignment’ waardoor een waarde wordt toegekend aan de constante. Het declareren van een variabele gaat op vrijwel dezelfde wijze, alleen is het bij een variabele niet nodig om er een ‘assignment’ te doen. De taal heeft drie basis typen: integer (wit), char (melk) en boolean (puur).

Er zijn vijf verschillende soorten statements, namelijk de assign statement, de read statement, de print statement, de if-then-else statement en de while-do statement. De keywords van de if-then-else statement en de while-do statement hebben geen speciale termen gekregen om de taal Chocolate leesbaar te houden.

cacao Dit is een stukje voorbeeld code

bonbon wit a; cacao Een variabele wordt hier gedeclareerd

reep wit b milka 5; cacao Een constante wordt hier gedeclareerd

droste(a); cacao Er wordt een integer gelezen en in a opgeslagen

a milka a milkyway b; cacao a wordt met b vermenigvuldigd en in a

cacao opgeslagen

verkade(a); cacao a wordt geprint

*Stukje voorbeeld code*

# Problemen en oplossingen - Kim

Tijdens het maken van de programmeertaal zijn een aantal problemen tegengekomen. Voor de ANTLR ll(k) parser bleek het lastig om te zorgen dat k=1. Dit zorgde voor moeilijkheden bij bijvoorbeeld de specificatie een ‘assignment’:

assign

: IDENTIFIER ASSIGN^ (assignexpr)

;

assignexpr

: IDENTIFIER ASSIGN^ assignexpr

| single\_expr

| closed\_compound\_expr

;

Hier geldt dat meerdere toewijzingen in één regel mogelijk zijn en dat er daarom meerdere ‘identifiers’ mogelijk zijn met als afsluiting een ‘single\_expr’. Het probleem dat hierbij ontstond, was dat een ‘single\_expr’ ook naar een ‘identifier’ kan leiden en dat de parser daarom niet wist voor welke optie er gekozen moest worden als er een ‘identifier’ tegengekomen werd. Dit is opgelost door gebruik te maken van een syntactisch predikaat dat de parser in de tokenstream laat kijken.

De regel is toen veranderd naar:

assign

: IDENTIFIER ASSIGN^ (assignexpr)

;

assignexpr

: (IDENTIFIER ASSIGN^) => (IDENTIFIER ASSIGN^ assignexpr)

| single\_expr

| closed\_compound\_expr

;

Wat hier gebeurt, is dat de parser eerst controleert of ‘IDENTIFIER ASSIGN’ de eerstvolgende tokens zijn. Als dit het geval is, wordt er voor de eerste regel gekomen. Als er echter een ‘IDENTIFIER’ voorkomt met daarna geen ‘ASSIGN’, wordt deze regel niet gekozen en belandt de parser bij ‘single\_expr’.

Verder is bij het specificeren van de checker het probleem tegengekomen dat er op types gecontroleerd moest worden, omdat er bij deze taal drie types mogelijk zijn, namelijk integer, char en boolean. Dit is gedaan door het construeren van een AST node, namelijk ChocolateTree, die CommonTree extend met een ChocolateTreeAdaptor die CommonTreeAdaptor extend. Vervolgens zijn in de checker voor de regels methoden uitgevoerd die de typen controleren en toewijzen aan variabelen of expressies. ChocolateTree slaat deze informatie op in een Symbol Table.

Bovenstaande methode voor het controleren op types leverde het probleem op dat er regels bestaan zonder root. Om deze reden kon het type van de regel niet worden opgeslagen in een node. Om deze reden is er voor deze regels een return value gebruikt, waarin als String het type wordt opgeslagen. Op deze manier wordt het type dan niet meegegeven als ChocolateTree-node, maar als String. Voorbeeld:

closed\_compound\_expr

: ^(r=*LCURLY* {ca.openScope();}declarations\* ce=compound\_ext

{ ca.checkCompoundExpr(r, $ce.val); } {ca.closeScope();})

;

compound\_ext **returns**[String val = CheckerActions.NO\_TYPE;]

: ^(r=*RCURLY* se=single\_expr)

{ ca.checkCompoundExt(r,$se.tree); $val = $se.tree.getChocolateType();}

| statements declarations\* c=compound\_ext

{$val = $c.val; $c.tree.setChocolateType($val);}

;

In bovenstaand voorbeeld bestaat er geen node voor de regel ‘statements declarations\* c=compound\_ext’. Daarom wordt als value de waarde van compound\_ext meegegeven en wordt deze value gebruikt in checkCompoundExpr in plaats van de ChocolateTree. Uiteindelijk wordt deze value ingevuld wanneer de regel met single\_expr gekozen wordt, omdat de value dan de waarde van de node van de single\_expr is.

Daarnaast bleek het in de codegenerator een probleem te zijn om een read- of printstatement met meet dan één variabele uit te voeren. Daarom zijn

CodeGenerator read en print splitsen, om read of print met één variabele uit te kunnen voeren

# Syntax, context-beperkingen en semantiek - Tanja

van de taal met waar nodig nadere uitleg over de

betekenis. Geef de beschrijving bij voorkeur in dezelfde terminologie als die gebruikt is bij

de beschrijving van Triangle in Watt & Brown (hoofdstuk 1 en appendix B).

De syntax wordt gespecificeerd door middel van een CFG, die bestaat uit:

* Een eindige set terminals:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| : | ; | ( | ) | { | } | , | \ | “ | bounty |
| mars | nuts | milkyway | dove | maltesers | snickers | kinder | bueno | lion | kitkat |
| bros | rolo | twix | if | else | then | while | do | pos?? | neg?? |
| reep | bonbon | milka | droste | verkade | wit | melk | puur | . | 0 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A |
| B | C | D | E | F | G | H | I | J | K |
| L | M | N | O | P | R | S | T | U | V |
| W | X | Y | Z | a | b | c | d | e | f |
| g | h | i | j | k | l | m | n | o | p |
| q | r | s | t | u | v | w | x | y | z |

* Een eindige set nonterminals inclusief het startsymbool:

Program (startsymbool)

Declarations

Declaration

Type

Type\_op

Statements

Statement

Read

Assign

Assignexpr

Print

Closed\_compound\_expr

Compound\_ext

Single\_expr

Arithmetic

Ifthenelse

Whiledo

Operand

String

Graphic

Boolean\_operator

Char\_operator

Identifier

Number

Comment

Whitespace

Digit

Lower

Upper

Letter

* Een eindige set productieregels:

|  |  |  |
| --- | --- | --- |
| Program | ::= | (Declarations\* Statements)+ |
| Declarations | ::= | Declaration **;** |
| Declaration | ::=  | | **reep** Type Identifier (**,** Identifier)\* **milka** Type\_op  **bonbon** Type Identifier (**,** Identifier)\* (**milka** Type\_op)? |
| Type | ::=  |  | | **wit**  **melk**  **puur** |
| Type\_op | ::=  | | Single\_expr  Closed\_compound\_expr |
| Statements | ::= | Statement **;** |
| Statement | ::=  |  |  |  | | Read  Print  Assign  Ifthenelse  Whiledo |
| Read | ::= | **droste (** Identifier (**,** Identifier)\* **)** |
| Assign | ::= | Identifier **milka** assignexpr |
| Assignexpr | ::=  |  | | Identifier **milka** assignexpr  Single\_expr  Closed\_compound\_expr |
| Print | ::= | **verkade (** (Closed\_compound\_expr | Single\_expr | String) ( **,** (Closed\_compound\_expr | Single\_expr | String))\* **)** |
| Closed\_compound\_expr | ::= | **{** Declarations\* Compound\_ext |
| Compound\_ext | ::= | Single\_expr **}**  Statements Declarations\* compound\_ext |
| Single\_expr | ::= | Arithmetic |
| Arithmetic | ::= | Arith2 (**twix** Arith2)\* |
| Arith2 | ::= | Arith3 (**rolo** Arith3)\* |
| Arith3 | ::= | Arith4 (**snickers** | **kinder** | **bueno** | **lion** | **kitkat** | **bros**) Arith4)\* |
| Arith4 | ::= | Arith5 (( **bounty** | **mars** ) Arith5)\* |
| Arith5 | ::= | Arith6 (**milkyway** | **dove** | **maltesers**) Arith6)\* |
| Arith6 | ::= | ( **bounty** | **mars** | **nuts** )? Operand |
| Ifthenelse | ::= | **if** Single\_expr **then** Closed\_compound\_expr (**else** Closed\_compound\_expr)? |
| Whiledo | ::= | **while** Single\_expr **do** Closed\_compound\_expr |
| Operand | ::=  |  |  |  | | Identifier  Number  **(** Single\_expr **)**  Boolean\_operator  Char\_operator |
| String | ::= | **“** graphic\* **“** |
| Graphic | ::=  |  |  |  |  | | Letter  Digit  Ws  **:**  **,**  **.** |
| Boolean\_operator | ::=  | | **true**  **false** |
| Char\_operator | ::= | **‘** (Digit | Letter) **‘** |
| Identifier | ::= | Letter (Letter | Digit)\* |
| Number | ::= | Digit+ |
| Comment | ::= | **cacao** .\* |
| Whitespace | ::= | (‘ ‘ | ‘\t’ | ‘\f’ | ‘\r’ | ‘\n')+ |
| Digit | ::= | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| Lower | ::= | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** |
| Upper | ::= | **A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z** |
| Letter | ::= | Lower | Upper |

In deze taal kunnen variabelen enkel gebruikt worden als ze gedeclareerd zijn en kunnen variabelen meerdere keren gedeclareerd worden in verschillende levels.

De contextual constraints bestaan uit scope rules en type rules. Voor deze taal wordt static binding gebruikt, omdat de checker bepaalt bij welke binding occurence een applied occurence van een variable hoort voordat het programma uitgevoerd wordt. Daarnaast is de taal ook statically typed, omdat in de checker de types gecontroleerd worden, voordat het programma wordt uitgevoerd.

De semantiek van Chocolate wordt hieronder gespecificeerd:

Een *declaratie* D wordt uitgewerkt om bindings te produceren en constanten toe te wijzen en kan ook variabelen toewijzen.

De *constante declaratie* ‘reep T I (,I)\* milka T\_O’ wordt uitgewerkt door elke versie van *I* te binden aan de waarde van *T\_O*.

De *variabele declaratie* ‘bonbon T I (,I)\*’ wordt uitgewerkt door elke versie van *I* te binden aan een nieuw toegewezen variabele met onbekende initiële waarde. De waarde wordt gedealloceerd wanneer het blok dat de declaratie bevat verlaten wordt.

De *variabele declaratie* ‘bonbon T I (,I)\* milka T\_O’ wordt uitgewerkt door elke versie van *I* te binden aan een nieuw toegewezen variabele met als initiële waarde T\_O. De waarde wordt gedealloceerd wanneer het blok dat de declaratie bevat verlaten wordt.

Een *type* T levert het type op van een ‘Identifier’, die gedeclareerd wordt.

Een *type\_op* T\_O levert een expressie op van het type T in een declaratie.

Een *statement* S wordt uitgevoerd variabelen te updaten, om waarden te verkrijgen of om te printen.

Het *read statement* ‘droste (I (,I\*))’wordt op de volgende manier uitgevoerd. Een waarde w

wordt ingelezen via de command-line en toegewezen aan de Identifier I. Elke volgende waarde die wordt ingelezen, wordt toegewezen aan de volgende Identifier I voor het aantal Identifiers dat in het read statement gebruikt worden.

Het *print statement* ‘verkade ((C | E | S) (, (C | E | S))\*)’ wordt als volgt uitgevoerd. Eerst wordt een closed\_compound\_expression C, een single\_expr E of een string S geprint op de command-line. Vervolgens wordt er iteratief een komma geprint en daarna weer een closed\_compound\_expression, single\_expr of string, totdat er geen komma, maar een haakje volgt.

Het *assign statement* ‘I milka A’ wordt als volgt uitgevoerd. Aan de Identifier I wordt de waarde van A toegewezen.

Het *assign\_expr statement* ‘I milka A | S | C’ wordt als volgt uitgevoerd. Recursief wordt aan de Identifier I de waarde van A toegewezen, tot en met de keer dat de waarde van A een S of een C is.

Het *if statement* ‘if S then C1 (else C2)?’ wordt als volgt uitgevoerd. De expressie S wordt geëvalueerd naar een boolean-waarde b. Als b waar is, wordt C1 uitgevoerd. Als C1 niet waar is en er bestaat een else-clausule, wordt C2 uitgevoerd.

Het *while* *statement* ‘while S do C’ wordt als volgt uitgevoerd. De expressie S wordt geëvalueerd naar een boolean-waarde b. Als b waar is, wordt C uitgevoerd en wordt vervolgens de while-clausule opnieuw uitgevoerd. Als b niet waar is, is de uitvoering van het while statement voltooid.

Een *single\_expr* E wordt uitgevoerd om een waarde op te leveren.

De expressie ‘O’ levert de waarde op van de operand O.

De unary expressie ‘P O’ levert de waarde op die gegenereerd wordt door de unary operator P uit te voeren op de operator O.

De binary expressie ‘E1 P E2’ levert de waarde op die gegenereerd wordt door de binary operator P uit te voeren op de waarden van de expressies E1 en E2.

Een *Identifier* Ievert op de volgende manier een waarde op. Als I gebonden is aan een variabele of constante levert I de waarde op die daarin opgeslagen is.

# Vertaalregels - Tanja

voor de taal, d.w.z. de transformaties waaruit blijkt op welke wijze een opeenvolging van symbolen die voldoet aan een produktieregel wordt omgezet in een opeenvolging van TAM-instructies. Vertaalregels zijn de ‘code templates’ van hoofdstuk 7 van Watt

& Brown.

# Beschrijving van Java-programmatuur - Tanja

Voor de compiler zijn nog enkele extra Java-klassen gedefinieerd. Ten eerste is er een SymbolTable gedefinieerd die ‘Identifiers’ opslaat. Hierin bestaan methoden voor het openen en sluiten van een scope, zodat het mogelijk is om dezelfde naam voor een ‘Identifier’ in verschillende scopes te gebruiken, een methode om het huidige level op te vragen en methoden om ‘Identifiers’ toe te voegen en op te vragen. Deze SymbolTable maakt gebruik van de klasse IdEntry, die van elke ‘Identifier’ opslaat wat het type is, het level waarin het gedeclareerd is en de node waarin de ‘Identifier’ hoort. Daarnaast zijn er methoden om het level op te vragen en te veranderen, om het type op te vragen en te veranderen en om de node op te vragen en te veranderen. Er bestaat een klasse SymbolTableException, die gebruikt wordt voor error handling wanneer er geen correct scope level bestaat als een ‘Identifier’ gedeclareerd wordt of wanneer de ‘Identifier’ al gedeclareerd is in het huidige level.

Verder is ChocolateTree.java toegevoegd, die een subklasse is van CommonTree. Dit is een AST-klasse die het type van een node bijhoudt, het memoryaddress voor de codegeneratie, of de node die wordt bijgehouden een variabele of constante is en wat de declarerende node is. Hiervoor is ook ChocolateTreeAdaptor.java ontworpen, die een subklasse is van CommonTreeAdaptor en ChocolateTree-nodes fabriceert.

Verder bestaat de klasse CheckerActions, die gebruikt wordt door ChocolateChecker. Hierin wordt een instantie van een SymbolTable aangemaakt en worden er methoden gedefinieerd voor het controleren of een ‘Identifier’ al gedeclareerd is en voor het declareren van een ‘Identifier’. Verder worden alle bestaande types bijgehouden (namelijk integer, boolean en char) en worden er methoden gedefinieerd om het type van een ‘Identifier’ in de SymbolTable op te vragen, om het type van een Node van ChocolateTree te wijzigen, om te controleren of een type bestaat en om twee types met elkaar te vergelijken. Alle overige methoden in CheckerActions worden gebruikt in de checker door voor regels te controleren of de types overeenkomen met de verwachte types en types van nodes op te slaan.

De klasse ChocolateException wordt gebruikt voor error handling wanneer een gebruikte ‘Identifier’ niet gedeclareerd is, of niet bruikbaar in het huidige scopelevel, wanneer een type niet overeenkomt met het verwachte type of wanneer er geprobeerd wordt om een constante ‘Identifier’ te overschrijven. Ook vangt ChocolateException een SymbolTableException op, wanneer er geprobeerd wordt een ‘Identifier’ te declareren, die in het huidige level al gedeclareerd is.

# Testplan en –resultaten - Kim

. Bespreking van de ‘correctheids-tests’ aan de hand van de criteria

zoals deze zijn beschreven in het xA.5 van deze appendix. Aan de hand van deze criteria moet

een verzameling test-programma’s in het taal geschreven worden die de juiste werking van de

vertaler en interpreter controleren. Tot deze test-set behoren behalve correcte programma’s

die de verschillende taalconstructies testen, ook programma’s met syntactische, semantische

en run-time fouten. Alle uitgevoerde tests moeten op de CD aanwezig zijn; van e´en testprogramma moet de ´

uitvoer in de appendix opgenomen worden (zie onder).

# Conclusies – Tanja

Met behulp van ANTLR is een complete vertaler ontwikkeld voor de programmeertaal ‘Chocolate’, die deze vertaald naar Java ByteCode. Hier zijn een parser, lexer en checker voor gemaakt om de taal in te lezen en te controleren op fouten. In dit verslag is besproken hoe ‘Chocolate’ gedefinieerd is en hoe tegengekomen problemen zijn opgelost. De syntax, context-beschrijving en semantiek van de programmeertaal zijn gegeven en de vertaalregels voor de taal zijn uiteengezet. Uiteindelijk is de Java-programmatuur beschreven en is het testen besproken.

Tijdens het ontwikkelen van de de vertaler voor de taal ‘Chocolate’ zijn er een aantal moeilijkheden opgedoken. Deze hadden vooral te maken met de lookahead in de parser wanneer er in regels recursie voorkomt, met het controleren van types bij expressies en het codegenereren naar Jasmin in plaats van het bekende TAM. Er zijn verschillende aanpakken geprobeerd om deze problemen op te lossen. Uiteindelijk is gekozen voor een syntactisch predikaat voor het oplossen van lookahead-problemen, het gebruik van een zelf-gedefinieerde AST klasse met SymbolTable voor het controleren van expressies en het gebruik van Jasmin voor het codegenereren naar Java ByteCode.