KIDEB
Onze eigen programmeertaal

H.M. Bastiaan
s1204254
Doctor van Damstraat 198, Enschede

V.J. Smit
s1206257
Kremersmaten 168, Enschede

8 juli 2014

Inhoudsopgave

Hoofdstuk 1

Inleiding

Hoofdstuk 2

Taalbeschrijving

KIDEB is een kleine, imperatieve programmeertaal met beperkte mogelijkheden, ontwikkeld als project voor het vak vertalerbouw als onderdeel van de Technische Informatica bachelor van de Universiteit Twente. Ondanks de beperkte mogelijkheden bevat de taal toch enkele leuke onderdelen.

Uiteraard is de basis van een programmeertaal ook aanwezig in KIDEB. Variabelen kunnen worden gedeclareerd met primitieve typen: integer, boolean en character. Door toegevoegde type inferentie hoeft het primitieve type zels niet expliciet te worden vermeld.

Op deze variabelen zijn een aantal bewerkingen mogelijk. Zo zijn de standaard boolean operaties beschikbaar, om vershil of gelijkheid te bepalen. Daarnaast bevat de taal de basis rekenkundige operaties en zelfs bestaat de mogelijkheid tot machtsverheffen. Belangrijke statements zijn ook geïmplementeerd. If-else is onderdeel van de taal, evenals een while-statement.

De eerste belangrijke uitbreiding is de mogelijkheid gebruik te maken van subroutines. Naast het hoofdprogramma zijn namelijk ook functies te definiëren en uit te voeren. Functies geven altijd een waarde terug. De tweede belangrijke uitbreiding was al even genoemd. Dit is namelijk het gebruik van arrays. Door het gebruik van arrays wordt de taal een stuk complexer, maar ook een stuk sterker. Daarnaast kan reeds eerder geschreven broncode geïmporteerd worden. Grote code reduplicatie is dus overbodig. Als laatste bestaat ook de mogelijheid met geheugen allocatie bezig te zijn. Geheugen kan gealloceerd en vrijgemaakt worden en er kan naar geheugenlocaties verwezen worden met behulp van pointers.

De volledige grammatica van de taal is te vinden in het hoofdstuk 4.

Hoofdstuk 3

Problemen en oplossingen

In dit hoofdstuk worden enkele belangrijke problemen besproken die bij de ontwikkeling van de taal naar voren kwamen. Allereerst komt scoping aan bod. Daarna wordt kort de AST besproken, gevolgd door type inferentie.

3.1 Scoping

Een belangrijk probleem bij het programmeren in het definiëren en gebruik van variabelen in verschillende scopes. Variabelen die bijvoorbeeld binnen een lus worden gedefinieerd, mogen daarbuiten niet gebruikt worden. Ook moet de variabele worden gebruikt die gedeclareerd is onder of in die scope.

Voor het definiëren van scopes is de volgende oplossing gekozen. Een scope binnen KIDEB bestaat tussen twee accolades. De scope wordt geopend door een '{'en gesloten door een '}'. Verdere uitleg over scoping is the vinden in subsectie 4.2.1.

Om bij te houden waar een variabele gedeclareerd is en gebruikt wordt, wordt een symbol table bijgehouden.

3.2 Opbouwen AST

Voor het opbouwen van onze AST voldeed de standaard node niet. Hiertoe is een eigen node-hiërarchie gemaakt. De specificatie van deze nodes is te vinden in hoofdstuk 6.

Een ander probleem met de AST is het moeten gebruiken van CommonNode voor de belangrijke node dublicatie. Om dit te voorkomen heeft de klasse AbstractNode, als subklasse van CommonNode, een functie met generiek type. Hier kunnen subklassen van CommonNode toch de methode *getDuplicate()* gebruiken.

3.3 Type inferentie

3.4 Import statement

Een belangrijke functie in onze taal is het importeren van code. Dit is een lastig probleem, met een vrij simpele oplossing. Op de plek van het import-statement wordt in de AST van het hoofdprogramma de AST van de geïmporteerde code geplaatst. Hierdoor is de code ook in het hoofdprogramma te gebruiken.

3.5 Arrays als argument

Het is niet mogelijk om in een functiedeclaratie een array mee te geven zonder een expressie binnen de blokhaken. Het is echter absoluut niet wenselijk te eisen dat arrays als argument voor een functie een vaste lengte hebben. Om dit probleem te verhelpen moeten functies die een array nodig heeft een pointer naar deze array mee krijgen, in plaats van de array zelf.

Hoofdstuk 4

Syntax, context en semantiek

Dit hoofdstuk bespreekt de specificatie van de taal aan de hand van de syntax, de context regels en de semantiek.

4.1 Syntax

Deze sectie beschrijft de symbolen en productieregels van KIDEB. Samen vormen deze de totale grammatica van de taal.

4.1.1 Terminale symbolen

De terminale symbolen:

:	;	(	)	[
]	{	}	,	\
!	+	-	/	<
^	=	<	>	>=
<=	==	||	&&	*
&	%	print	import	call
swap	if	else	then	do
while	from	break	continue	return
returns	func	true	false	
char	var	of	int	bool

4.1.2 Non-terminale symbolen

De non-terminale symbolen:

program (startsymbool)

command

declaration IDENTIFIER
- var_declaration
- scope_declaration
- func_declaration
- assign_statement

assignment
- var_assignment

argument
- arguments

statement
- while_statement
- if_statement
- if_part
- else_part
- for_statement
- return_statement
- assign_statement
- print_statement
- import_statement

expression
- expressionAO
- expressionLO
- expressionPM
- expressionMD
- expressionPW
- expression_list
- call_expression
- raw_expression
- get_expression
- operand
- array_literal
- array_value_list

type
- primitive_type
- compositie_type

identifier

number

4.1.3 Productieregels

program :=
    command+;

command :=
    assign_statement SEMICOLON |
    declaration |
    statement |
    expression |
    SEMICOLON;

commands :=
    command commands?;

declaration :=
    var_declaration |
    scope_declaration;

var_declaration :=
    type IDENTIFIER (var_assignment) SEMICOLON;

scope_declaration :=
    func_declaration;

func_declaration :=
    FUNC IDENTIFIER LPAREN arguments? RPAREN RETURNS
    type LCURLY commands? RCURLY;

assignment :=
    ASSIGN expression;

var_assignment :=
    ASSIGN expression;

argument :=
    type IDENTIFIER;

arguments :=
    argument (COMMA arguments)?;

statement :=
    if_statement |
    while_statement |

return_statement |
import_statement |
BREAK SEMICOLON |
CONTINUE SEMICOLON;

if_statement :=
if_part else_part?;

if_part :=
IF LPAREN expression RPAREN LCURLY command* RCURLY;

else_part :=
ELSE LCURLY command* RCURLY;

while_statement :=
WHILE LPAREN expression RPAREN LCURLY commands? RCURLY;

for_statement :=
FOR LPAREN expression RPAREN LCURLY commands? RCURLY;

return_statement :=
RETURN expression SEMICOLON;

print_statement :=
PRINT LPAREN expression RPAREN;

import_statement :=
IMPORT STRING_VALUE;

expression :=
raw_expression |
expressionAO |
array_literal;

expressionAO :=
expressionLO (AND expressionLO |OR expressionLO)*;

expressionLO :=
expressionPM ((LT |GT |LTE |GTE |EQ |NEQ) expressionPM)*;

expressionPM :=
expressionMD ((PLUS |MINUS) expressionMD)*;

expressionMD :=
expressionPW ((MULTIPLE |DIVIDE) expressionPW);

expressionPW :=
operand (POWER operand)*;

expression_list :=
    expression (COMMA expression_list)?;

call_expression :=
    IDENTIFIER LPAREN expression_list? RPAREN;

get_expression :=
    IDENTIFIER LBLOCK expression RBLOCK;

operand :=
    get_expression |
    call_expression |
    ASTERIX operand |
    AMPERSAND IDENTIFIER |
    ASTERIX operand |
    LPAREN expression RPAREN |
    IDENTIFIER |
    NUMBER |
    STRING_VALUE |
    bool;

bool :=
    TRUE |
    FALSE;

array_literal :=
    LBLOCK array_value_list RBLOCK;

array_value_list :=
    expression (COMMA array_value_list)?;

type
    primitive_type
    compositie_type

primitive_type :=
    INTEGER |
    BOOLEAN |
    CHARACTER |
    AUTO |
    VAR;

composite_type :=
    primitive_type LBLOCK expression RBLOCK

IDENTIFIER :=
    (LETTER |UNDERSCORE) (LETTER |DIGIT |UNDERSCORE);

NUMBER :=
    DIGIT+;

STRING_VALUE :=
    ‘(\\‘? |˜(\\|‘))*‘;

COMMENT :=
    // .*\n;

WS :=
    ␣|\t |\f |\r |\n

DIGIT :=
    0..9;

LETTER :=
    LOWER |UPPER;

LOWER :=
    a..z;

UPPER :=
    A..Z;

UNDERSCORE :=
    _;

4.2 Context

De context van de taal wordt opgedeeld in twee delen, namelijke scope regels en type regels. De eerste bespreekt declaratie en het gebruik van variabelen. De tweede bespreekt de typering van de taal.

4.2.1 Scope regels

Om de scoperegels uit te leggen, gebruiken we de volgende voorbeeld code.

```
1 int x;
2 x = 5;
3
4 func som(int x) returns int {
5     int y = 7;
6     return x + y;
7 }
8
9 print(som(x));
```

Op regel 1 wordt variabele x gedeclareerd, dit is de *binding occurence* voor x. De eerste *applied occurence* komt meteen op regel 2, waar x de waarde 5 kijgt.

De functie *som* wordt op regel 4 gedefinieerd en telt de waarde van variabele y hierbij op. De variabele y wordt gedefinieerd binnen de functie en is dus ook alleen binnen de functie te gebruiken.

4.2.2 Type regels

Voor de rekenkundige operatoren gelden de volgende type regels.

prioriteit	**operatoren**	**operand types**	**resultaat type**
1	^	int	int
2	*,/	int	int
3	+,-	int	int
4	<, <=, >=, >	int	bool
	==, !=	int, char, bool	bool
5	&&,||	bool	bool

Voor de statements gelden de volgende regels.

`if Expression then Command else Command`
Expression must be of type *boolean*.
`while Expression do Command`
Expression must be of type *boolean*.
`Identifier = Expression`
Identifier and Expression must be of the same type.

4.3 Semantiek

Deze sectie bespreekt de semantiek, ofwel de betekenis van de geschreven code.

Een *statement* S wordt uitgevoerd om de variabelen te updaten. Dit is inclusief input en output.

- Assign-statment(I = E): Expressie E wordt gevalueerd en levert de waarde v op. Deze waarde wordt gekoppeld aan I.
- Import-statment(import SW): Van stringwaarde SV wordt uitgezocht of het bestaat als .kib broncode bestand. Zo ja, wordt de AST van het porgramma uitgebreid met de AST van dit bestand en is de code uit dit bestand beschikbaar voor gebruik in de eigen brongcode.
- If-statement(if E then C_1 else C_2): Expressie E wordt geëvalueerd en levert een booleanwaarde op. Als de boolean waarde `true` is, worden commando's C_1 uitgevoerd, anders commando's C_2.

- While-statement(while E do C): Expressie E wordt gevalueerd en levert een booleanwaarde op. Als de booleanwaarde `true` is, wordt commando C uitgevoerd. Daarna wordt E opnieuw gevalueerd. Is de booleanwaarde `false`, dan eindigt de loop.

- Print-statement(print(E)): De expressie E wordt gevalueerd en levert waarde v op. Deze waarde v wordt op de standaard output getoond.

Een *expressie E* levert een waarde op na evaluatie.

- expressionAO(E_1 operator E_2): Evalueert expressie E_1 en E_2, welke beiden een boolean opleveren. De waarde die de expressie AO oplevert is ook een boolean, met als operator de binaire AND(&&) of OR(||).

- expressionLO(E_1 operator E_2): Evalueert expressie E_1 en E_2, welke beiden een integer opleveren. De waarde die de expressie LO oplevert is een boolean, met als operator een waarde vergelijker.

- expressionPM(E_1 operator E_2): Evalueert expressie E_1 en E_2, welke beiden een integer opleveren. De waarde die de expressie AO oplevert is de opgetelde of afgetrokken waarde van beide expressies.

- expressionMD(E_1 operator E_2): Evalueert expressie E_1 en E_2, welke beiden een integer opleveren. De waarde die de expressie AO oplevert is de vermenigvuldigde of gedeelde waarde van beide expressies.

- expressionPW(E_1 operator E_2): Evalueert expressie E_1 en E_2, welke beiden een integer opleveren. De waarde die de expressie AO oplevert is E_1 tot de macht E_2.

- raw-expressie(__tam__ (type, SW)): JA, wat doet dit eigenlijk.

- call-expressie(I(E_1,... E_x)): Evalueert expressie E_1 tot en met E_x en leveren waarden v_1 tot en met v_x op. Functie I wordt vervolgens aangeroepen met argumenten v_1 tot en met v_x.

- get-expressie(get I E): JA, wat doet dit eigenlijk.

Een *delclaratie D* wordt uitgevoerd om bindingen te maken.

- Variabele declaratie(T I): Identifier I wordt gebonden aan een waarde, die op dit moment nog onbekend is. De waarde moet gelijk zijn aan type T. De variabele wordt buiten de scope waarin deze wordt gebruikt, gedealloceerd.

- Functie declaratie(func I(ARGS) returns T{C}): Functie met identifier I wordt aangemaakt. De argumenten ARGS stellen de waarden voor die de functie in de aanroep ervan mee moet krijgen. Type T is het

type van de return-waarde van functie I. Commando C vormt de body van de functie en dit zijn de commando's die uitgevoerd worden na aanroep van deze functie.

Hoofdstuk 5

Vertaalregels

Om de vertaalregels van de broncode naar TAM-code duidelijk te maken, worden code templates gebruikt. KIDEB kent de volgende acties.

Klasse	Code functie	Effect van gegenereerde code
Program	*run P*	Draai programma P en daarna stoppen. Beginnen en eindigen met een lege stack.
Commando	*do C*	Is de volledige lijst met instructies, bestaande uit statements, expressies en declaraties.
Statement	*execute S*	Voer het statement S uit met mogelijk aanpassen van variabelen, maar zonder effect op de stack.
Expressie	*evaluate E*	Evalueer de expressie E en push het reultaat naar de stack. Geen verder effect op de stack.
Identifier	*fetch I*	Push de waarde van identifier I naar de stack.
Identifier	*assign I*	Pop een waarde van de stack en sla deze op in variabele I.
Declaratie	*declare D*	Verwerk declaratie D, breidt de stack uit om ruimte te maken voor variabelen die hierin gedeclareerd worden.

Een programma in KIDEB is een serie commands. Elk los command kan een statement, een expressie of een declaratie zijn. Een programma kan er dus als volgt uitzien.

run [[**C**]] =
    do C
    HALT

Dit leidt tot de volgende commando's.

do [[S]] =
execute S

do [[E]] =
evaluate E

do [[D]] =
declare D

De statements gaan als volgt.

execute [[I = E;]]
evaluate E
assign I

execute [[if E then C_1 else C_2]] =
evaluate E
JUMPIF(0) *g*
execute C_1
JUMP *h*
g: gexecute C_2
h:

execute [[while E then C]] =
JUMP *h*
g: execute C
h: evaluate E
JUMPIF(1) *g*

execute [[return E]] =
evaluate E
RETURNS(1)

execute [[import SW]] =
Is wel een statement, maar levert geen code op. Importeert nl een bestand met naam SW, en voegt de AST van die code toe aan de AST van het hoofdbestand.

execute [[print E]] =
evaluate E
CALL put
LOADL 10 newline
CALL putint

De expressie worden als volgt verwerkt.

evaluate [[E_1 O E_2]] =
evaluate E_1

evaluate E_2
CALL p p is het adres van de primitieve routine die hoort bij O

evaluate [[**I**]] =
fetchV

evaluate [[ ]]

Laden en opslaan van waarden in variabelen

fetch [[**I**]]

Declaraties worden hiermee afgehandeld.

declare [[ ]]

Hoofdstuk 6

Java programmatuur

Voor het correct laten werken van de KIDEB-compiler zijn een aantal extra Java-klassen gedefinieerd. Deze betreffen het opbouwen van de *symbol table*, code voor het checken van types en de extra nodes voor de AST. Ook valt hieronder een stukje foutafhandeling.

6.1 Symbol table

De zogenaamde *symbol table* zoekt uit welke variabele waar gedefinieerd is, waar deze gebruikt wordt en koppelt dit aan elkaar. De reeds gemaakte *symbol table* uit het practicum van vertalerbouw is hiervoor gebruikt.

6.2 Type checking

Voor het verwerken van types is het de klasse Type aangemaakt. Deze klasse bevat een enumerator van alle types die onze taal kent. Deze klasse wordt gebruikt voor het zetten van de types van de identifiers.

6.3 AST klassen

De AST is uitgebreid met extra nodes, om extra benodigde informatie bij te houden.

De abstracte klasse *AbstractNode* is een subklasse van CommonTree, de normale AST-klasse. Deze vormt de superklasse voor de klasse CommonNode. Deze laatste is de superklasse voor alle zelf-gedefinieerde AST-nodes. De node van het import-statement is een instantie van deze klasse, want, JA¡ WAAROM??

De klasse ControlNode representeert nodes die de uitvoervolgorde van het programma veranderen. Hieronder vallen onder andere het return-, continue- en break-statement. Deze klasse houdt de scope bij waar het statement bij hoort. Deze klasse is een subklasse van CommonNode.

Een andere subklasse van CommonNode is de klasse TypedNode. Deze klasse vormt de superklasse van alle nodes in de AST die van een bepaald type zijn. Daarom heeft deze klasse als eigenschap een type. Tevens is het geheugenadres van deze (VAN DEZE WAT??) een eigenschap.

Onder de TypedNode komt de IdentifierNode. Deze node wordt gebruikt om de scope van een identifier te bepalen, evenals zijn type. Als eigenschap heeft deze klasse een zogenaamde *realNode*. Deze *realNode* is de node waar de declaratie van deze identifier plaatsvindt en er wordt bij gebruik van de identifier naar verwezen.

Het laagst in de hiërarchie zit de FunctionNode. Deze node wordt vanzelfsprekend gebruikt voor functies en houdt een lijst van identifiers en bijbehorende types bij. Deze lijst is dus de lijst met argument. Tevens heeft deze node een naam en returnType als eigenschap.

6.4 Foutafhandeling

Om compilatie af te breken bij een typefout, is de InvalidTypeException-klasse geïmplementeerd. Deze exceptie wordt gegooid bij een declaratie, als het gedeclareerde type niet bestaat. Tevens wordt deze exceptie gegooid als er verkeerde types in expressies gebruikt worden.

Hoofdstuk 7

Testplan en resultaten

Om de juistheid van de compiler te garanderen, is testen essentieel. Hiertoe zijn dan ook twee typen tests geschreven. De eerste is een in *python* geschreven programma, welke de syntax en context test. Het tweede type bestaat uit grotere voorbeeldprogramma's, welke syntax, context en semantiek testen.

De testprogramma's geschreven in *python* zijn te vinden in het bijgeleverde zip-bestand in de map *tests*. De testprogramma's uit onze eigen taal in de map *examples*.

7.1 Python programma's

De python programma's testen zeer snel en efficiënt de volledige syntax en context. Hieronder vallen zowel correcte als foute code. Als de code bewust fout gaat, wordt dit afgevangen op de correcte foutmelding.

Als al deze tests slagen is er geen noemenswaardige output. Mocht er een test wel falen, dan wordt dit getoond op de standaard output.

7.2 Testprogramma's

Verder wordt het volgende getest in deze correcte testprogramma's. Al deze programma's werken en leveren dus geen noemenswaardige resultaten op.

array.kib Het declareren van, toewijzen van waarden aan, en uitlezen van een array.

fibonacci_recursive.kib, power.kib Het declareren en gebruik van een functie.

heap.kib Het alloceren en vrijgeven van geheugen wordt hier getest.

pointers.kib Test de declaratie en het gebruik van pointers.

raw.kib Ja, uh, wat??

Het volgende wordt gestest op fouten, met incorrecte testprogramma's. Deze program"ma's leveren ook de bijgeschreven output.

Hoofdstuk 8

Conclusies

Hoofdstuk 9

Appendix

9.1 Lexer specificatie

Voor de lexer zijn de verschillende tokens van belang. Deze tokens staan hieronder allen gedefinieerd.

Tekens

Token	teken
COLON	:
SEMICOLON	;
LPAREN	(
RPAREN	)
LBLOCK	[
RBLOCK	]
LCURLY	{
RCURLY	}
COMMA	,
DOUBLE_QUOTE	”
SINGLE_QUOTE	\
BODY	body
EXPR	assignment_expression
GET	get_expression

Operators

Token	teken
PLUS	+
MINUS	-
DIVIDES	/
MULTIPL	*
POWER	^
LT	<
GT	>
GTE	>=
LTE	<=
EQ	=
NEQ	!
ASSIGN	==
OR	||
AND	&&

Pointers

Token	teken
AMPERSAND	&
ASTERIX	%

Keywords van KIDEB

Token	keyword
PROGRAM	program
SWAP	swap
IF	if
THEN	then
ELSE	else
DO	do
WHILE	while
FROM	from
IMPORT	import
BREAK	break
CONTINUE	continue
RETURN	return
FOR	for
IN	in
RETURNS	returns
FUNC	func
ARRAY	array
ARGS	args
CALL	call
VAR	var
OF	of
PRINT	print
TAM	__tam__

Standaard types

Token	keyword
INTEGER	int
CHARACTER	char
BOOLEAN	bool
AUTO	auto

9.2 Parser specificatie

9.3 Treeparser specificaitie

9.4 Testverslag