KIDEB

Onze eigen programmeertaal

H.M. Bastiaan
s1204254
Doctor van Damstraat 198

V.J. Smit
s1206257
Kremersmaten 168

3 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-statment. Voor het doorlopen van een complete array is het for-statement bijgevoegd.

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 hebben de mogelijkheid tot het opleveren van een waarde, maar dat is niet verplicht.

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.

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 standaar node niet. Hiertoe is een eigen node-hiërarchie gemaakt. De specificatie van deze nodes is te vinden in hoofdstuk 6.

3.3 Type inferentie

Type inferentie is een krachtige toevoeging voor een taal, maar brengt ook problemen met zich mee. Op compiletijd moet bepaald worden wat het type is, zonder het expliciet te vermelden.

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:

:	;	(	)	[
]	{	}	,	\
”	+	-	/	<
ˆ	=	<	>	>=
!	<=	==	||	&&
swap	if	else	then	do
while	from	break	continue	return
for	in	returns	func	array
args	var	of	int	bool
char	call			

4.1.2 Non-terminale symbolen

De non-terminale symbolen:

program (startsymbool)

command

declaration
- var_declaration
- scope_declaration
- func_declaration

assignment
- var_assignment

argument
- arguments

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

expression
- expressionAO
- expressionLO
- expressionPM
- expressionMD
- expressionPW
- expression_list
- call_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)? {commands?};

assignment :=
    ASSIGN expression;

var_assignment :=
    ASSIGN expression;

argument :=
    type IDENTIFIER;

arguments :=
    argument (COMMA arguments)?;

statement :=
    if_statement |
    while_statement |
    for_statement |
    return_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;

expression :=
    call_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;

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;

composite_type :=
primitive_type LBLOCK expression RBLOCK

IDENTIFIER :=
LETTER (LETTER |DIGIT);

NUMBER :=
DIGIT+;

STRING_VALUE :=
‘(\\‘? |˜(\\|‘))∗‘;

COMMENT :=
// .∗\n;

WS :=
␣|\t |\f |\r |\n

DIGIT :=
0..9;

LETTER :=
LOWER |UPPER;

LOWER :=
a..z;

UPPER :=
A..Z;

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
6	||	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*.

```
for Identifier in Expression Command
```
Identifier must be of type integer. Expression must be of type array.
```
Identifier = Expression
```
Identifier and Expression must be of the same type.

4.3 Semantiek

Hoofdstuk 5

Vertaalregels

Hoofdstuk 6

Java programmatuur

Hoofdstuk 7

Testplan en resultaten

7.1 Testen

7.2 Resultaten

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

Operators

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

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
VAR	var
OF	of

Standaard types

Token	keyword
INTEGER	int
CHARACTER	char
BOOLEAN	bool
CALL	call

9.2 Parser specificatie

9.3 Treeparser specificaitie

9.4 Testverslag