Yet Another Programming Language

Vertalerbouw

N. ten Veen
s1223631
Leijdsweg 15

Wybren Kortstra
s1209531
Leijdsweg 15

June 24, 2014

Contents

Chapter 1

Introduction

Yet Another Programming Language, later on referred to as YAPL, is created with the purpose to gain a better understanding of translators and programming languages.

In this report we will explain in detail how YAPL is built and works. We will dive into the syntax, semantic and contextual constraints of the language.

We will dive into the way the compiler was written for YAPL. This is done with the help of ANTLR4. From a grammar, ANTLR generates a parser that can build and walk parse trees. The tree is then walked and YAPL code is then translated to the Java Virtual Machine (JVM).

Last but not least, there are some tests. The tests use all the functionality of YAPL and shows correct and incorrect statements. The tests contains syntactical errors as well as semantically run-time errors.

Chapter 2

Short Description

YAPL is a simple imperative programming language. The language is quite practical for smaller programs, but has also something to offer for the more skilled programmer.

YAPL is the abbreviation for Yet Another Programming Language.

The syntax of YAPL is based on Scala and Java, there are also other programming language that have constructions like YAPL, but these constructions are either in Scala or Java.

YAPL has the following programming constructions.

- declaration: We can declare either variables or constants.
  - constants: defined with a default value that does not change
  - variables: declaration of variable name with its type
- assignment: this is an expression which assigns a value to a variable
- expressions: examples of expressions are if then else, while

Chapter 3

Problems and Solutions

During the development of this language we ran into some problems. This problems were not always trivial and we will describe here the problem and the solution. Hopefully this can help you to understand certain decisions and help you when you develop your own language or extend our language.

3.1 ANTRL4 and ll(1)

In ANTLR3 it was very simple to check if your language was ll(1), in the options you set k to 1. If the language would need to have a look-ahead of more then 1 ANTLR3 would give an error. ANTLR4 however does not have this option anymore.
A solution could be to check if the function *adaptivePredict* exists in the java source. Another option is to copy paste the ANTLR4 grammar into ANTLR3 and set the option $k = 1$. The ANTLR4 grammar is almost equal to ANTLR3 except for some minor things like the options and trimming the whitespace.

Chapter 4

Syntax

4.1 Example program

A YAPL program could look something like this.

```
var i: int;
const c = 4;

i = 7;

print(if i==c then c else i);
```

This is a trivial program and does nothing interesting.

4.2 Terminal symbols

The terminal symbols of YAPL include:

var	const	return	if	then	else	while	do	end
;	:	=	{	}	(	)	@	,
—		&	==	!=	!		=	
=	+	-	*	/	%			

4.3 Nonterminal symbols

The nonterminal symbols of YAPL include:

yapl (start symbol)	statement		
declaration			
expression	exprconstruct	orExpr	andExpr
compareExpr	plusMinusExpr	multDivModExpr	primaryExpr
typeDenoter			
operand	id		
letter	digit		

4.4 Production Rules

yapl	::=	statement*	(1.1)
statement	::=	**(**declaration | expression**);**	(1.2)
declaration	::=	***var*** id **:** typeDenoter	(1.3a)
	|	***const*** id **=** expression	(1.3b)
expression	::=	exprconstruct (**=** expression)?	(1.4)
exprconstruct	::=	orExpr	(1.5a)
	|	**{** statement* ***return*** expression **}**	(1.5b)
orExpr	::=	andExpr (**||** andExpr)*	(1.6)
andExpr	::=	compareExpr (**&&** compareExpr)*	(1.7)
compareExpr	::=	plusMinusExpr ((**>** | **>=** | **<** | **<=** | **==** | **!=**) plusMinusExpr)*	(1.8)
plusMinusExpr	::=	multDivModExpr ((**+** | **−**) multDivModExpr)*	(1.9)
multDivModExpr	::=	primaryExpr ((***** | **/** | **%**) primaryExpr)*	(1.10)
primaryExpr	::=	(**+** | **−** | **!**)? operand	(1.11)
operand	::=	id (**(** expression (**,** expression)***)**?	(1.12a)
	|	number	(1.12b)
	|	**(** expression **)**	(1.13)
typeDenoter	::=	id	(1.14)
include	::=	letter (letter | digit)*	(1.15)
letter	::=	[***a***-***z***]	(1.16a)
	|	[***A***-***Z***]	(1.16b)
digit	::=	[**0**-**9**]	(1.17)

Chapter 5

Contextual constraints

Chapter 6

Semantics

Chapter 7

Translation rules

Chapter 8

Code description

Chapter 9

Testing

Chapter 10

Conclusion

Appendix A

jemoeder