Yet Another Programming Language

Vertalerbouw

N. ten Veen	Wybren Kortstra
s1223631	s1209531
Leijdsweg 15	Leijdsweg 15

July 8, 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. For this reason we included the ANTLR3 specification of our language. It can be found in Appendix **??**.

3.2 Optional return statement for compound expression

The while statement is of type void. Because usually the body of a while expression contains a compound expression, we have chosen to force this type of expression for the body of a while statement. This gives the added benefit that the while statement does not need a closing token to keep it LL(1) since the curly braces of a compound expression already captures this. Because the last expression of a compound expression should be a expression, this could give some problems with certain statements unable to be executed. Therefore we chose to lift this limitation and instead have a return type of Void for a compound expression if the last statement is not an expression.

3.3 Greedy Else

A rule `'if' expression 'then' expression ('else' expression)?` can have ambiguity whenever this rule can be an expression since the second expression could also creative an if statement and then it is uncertain if the else part belongs to the first or the second if expression. To solve this, the parser could be greedy in that it always matches an if statement as close as possible. This was quite easy to do in ANTLR3, but we were not able to reproducing this behaviour in ANTLR4. For this reason we added the 'end' token at the end of a conditional statement to make sure the grammar is LL(1).

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	orExpr	andExpr	
compareExpr	plusMinusExpr	multDivModExpr	primaryExpr
opCompare	opMultDivMod	opPlusMinus	
exprBlock			
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	::=	orExpr (**=** expression)?	(1.4)
orExpr	::=	andExpr (**||** andExpr)*	(1.5)
andExpr	::=	compareExpr (**&&** compareExpr)*	(1.6)
opCompare	::=	**>** | **>=** | **<** | **<=** | **==** | **! =**	(1.7)
compareExpr	::=	plusMinusExpr (opCompare plusMinusExpr)*	(1.8)
opPlusMinus	::=	**+** | **−**	(1.9)
plusMinusExpr	::=	multDivModExpr (opPlusMinus multDivModExpr)*	(1.9)
opMultDivMod	::=	***** | **/** | **%**	(1.10)
multDivModExpr	::=	primaryExpr (opMultDivMod primaryExpr)*	(1.11)
primaryExpr	::=	(**+** | **−** | **!**)? operand	(1.12)
operand	::=	id (**(** expression (**,** expression)*)?**)**)?	(1.13a)
	|	number	(1.13b)
	|	(letter | digit)*	(1.13c)
	|	**(** expression **)**	(1.13d)
	|	***true***	(1.13e)
	|	***false***	(1.13f)
	|	***if*** expression ***then*** expression (***else*** expression)? ***end***	(1.13g)
	|	exprBlock	(1.13h)
	|	***while*** expression exprBlock	(1.13i)
exprBlock	::=	**{** statement* (***return*** expression **;**)? **}**	(1.13j)
typeDenoter	::=	id	(1.14)
id	::=	letter (letter | digit)*	(1.15)
letter	::=	[***a***-***z***]	(1.16a)
	|	[***A***-***Z***]	(1.16b)
digit	::=	[**0**-**9**]	(1.17)

Chapter 5

Contextual constraints

5.1 Statement

- When closing a scope the context checker checks if every declared variable is used in the scope. If a variable is unused the checker will give a warning.

5.2 Declaration

- In `var I : T`, I cannot be declared twice on the same scope level.
- In `const I = E`, I cannot be declared twice on the same scope level.
- Before a variable can be used it must first be declared.

5.3 Expression

- In `OE (= E)?`, if = exists then expression E must be of the same type as the expression OE and OE must be an identifier.
- In `if E then EI (else EE)?`, E must be of type Boolean. If there is an else clause EI and EE must be of the same type.
- In `while E EB`, E must be of type Boolean.
- In E_1 `(O` E_2`)*`, if O is of type + | - | / | * | % then E1 and E2 must be of type Int.
- In `OP O`, O must be of type Boolean if OP is !. If OP is + | - then O must be of type Int.
- In E_1 `O` E_2, if O is of type ¿ | ¿= | ¡ | ¡= | == | != then E_1 and E_2 must both be of the same type.
- In E_1 `AND` E_2 and E_1 `OR` E_2 E_1 and E_2 must both be of type Boolean.
- In `F(`E_1`,` E_2`,` E_N`)`, if F must be `read` or `write`. The number of arguments of both these functions must be 1 or more. The type of the arguments may not be Void or Error.

Chapter 6

Semantics

6.1 Statement

- A declaration statement `D` is elaborated.
- A expression statement `E` is evaluated and the result is discarded.

6.2 Declaration

- A declaration `I : T` is elaborated by binding a new variable `I` to type `T`. The initial value of `I` is the zero value of type `T`.
- A declaration `I = E` is elaborated by binding expression `E` to `I`. If the expression is known at compile time, an occurrence of `I` will be substituted with the result of `E`. If the result of `E` is not known at compile time, `E` is evaluated and bound to `I`.

6.3 Expression

- A number literal `N` yields the integer value of `N`.
- A character literal `C` yields the character value of `C`.
- A boolean literal `B` yields the boolean value of `B`
- A parentheses expression `(E)` evaluates `E`.
- A conditional **if** `B` **then** E_1 **else** E_2 is evaluated by evaluating `B`. If this expression yields true, then E_1 is evaluated. If `B` yields false, E_2 is evaluated. If the types of E_1 and E_2 do not match, the evaluated expression is discarded.
- A conditional **if** `B` **then** E_1 is evaluated by evaluating `B`. If this expression yields true, then E_1 is evaluated and the result is discarded.
- an expression block `SL E` is evaluated by executing all statements `SL` and if `E` is given, `E` is evaluated.

- a while expression `while B do EB` is evaluated by evaluating `B`. if this yields true, `EB` is evaluated. Whenever `EB` yields a result, this result is discarded. This is repeated until `B` yields false.
- An assignment `I = E` is evaluated by evaluating `E` and binding it to `E`. The result of the assignment is `E`.
- A unary expression `O E` is evaluated by evaluating `E` and applying `O(E)`.
- A binary expression E_1 `O` E_2 is evaluated by applying `O(`E_1, E_2`)`
- A variable expression `V` yields the value identified by `V`.

Chapter 7

Translation rules

The codegenerator uses translation rules to generate machine code from a decorated AST. These rules can be defined in StringTemplate, a templating engine used by ANTLR. In our implementation, these rules are rendered using StringTemplate. The StringTemplate group files are located under `src/st`.

7.1 Program

```
run[S*] =
  class_header
  foreach S_i in S:
    execute S_i
  class_footer
```

A program is a sequence of statements. First, a header is emitted that is required by the class format, then the statements are executed and finally a class footer is emitted.

```
class_header[name] =
  .class public [name]
  .super java/lang/Object

  .method public static main([Ljava/lang/String;)V
    .limit stack        512
    .limit locals       512

class_footer[] =
  return
  .end method
  readFunctions[]
```

The code function read_functions is rather verbose. The read functions are generated by writing these code functions in Java and compiling them to JVM bytecode. this file is then disassembled and the assembly instruction are included as the code function `readFunctions`. The following Java functions were written for the read functions:

```java
private static int readInt(){
        try{
                return Integer.parseInt(readLine());
        } catch(NumberFormatException e){
                return 0;
        }
}

private static String readLine(){
        BufferedReader reader = new BufferedReader(new
            InputStreamReader(System.in));
        try{
                return reader.readLine();
        } catch(IOException e){
                return "";
        }
}

private static char readChar(){
        String s = readLine();
        return s.length() > 0 ? s.charAt(0) : '\0';
}

private static boolean readBoolean(){
        String s = readLine().toLowerCase();
        return s.equals("true");
}
```

7.2 Statement

```
execute[D] =
  elaborate D
```

A declaration statement simply elaborates the declaration.

```
execute[E] =
  evaluate E
  pop                       if type(E) not Void
```

An expression statement first evaluates the expression. When the type of the expression is not Void, the result is popped off the stack prevent unused values from staying on the stack.

7.3 Declaration

```
elaborate[var I : T] =
  iconst_0
  istore i                  where i = variable offset of I
```

A variable declaration initializes the variable with the zero value of that type. The variable offset is determined in the context checking phase.

```
elaborate[const I = E] =  if E cannot be computed at compile time
  evaluate E
  istore i                  where i = variable offset of I
```

A constant declaration only generates code when a constant cannot be computed at compile time. Whenever such a constant is declared, the constant expression is evaluated and assigned to variable I at the given offset.

7.4 Expression

```
evaluate[N]
  ldc i                    where i = integer value of N
```

An integer constant pushes its integer value on the stack

```
evaluate[C]
  ldc c                    where c = integer code of C
```

A Character constant pushes the charcode of the Character on the stack

```
evaluate[B]
  ldc b                    where b = 1 if b == true, else 0
```

A Boolean constant pushes a 1 on the stack when it is true and a 0 when it is false

```
evaluate[(E)] =
  evaluate E
```

A parentheses expression simply evaluates the expression within parentheses.

```
evaluate[if B then E1 else E2]
  iconst_1
  evaluate B
  if_icmpne      g
  evaluate E1
  goto h
g:
  evaluate E2
h:
```

The conditional is compared to 1. When it is not equal, it jumps to the false expression. The other case evaluates the true expression and jumps over the false expression

```
evaluate[if B then E]
  iconst_1
  evaluate B
  if_icmpne g
  evaluate E
  pop                      if type(E) not Void
g:
```

The conditional is compared to 1. When it is not equal, it jumps over the true expression. Whenever the type of the expression is not void, the value needs to be popped off the stack since it is not used anymore.

```
evaluate[S* E] =
  foreach Si in S:
    execute Si
  evaluate E     if E is not null
```

An expression block executes all statements and finally the return expression, if it is given.

```
evaluate[I = E] =
  evaluate E
  dup
  istore i                    where i = variable offset of I
```

An assignment evaluates the expression, then duplicates it to allow it to store it to `I` and return it as a result

```
evaluate[-E] =
  evaluate E
  ineg
```

The unary negate operator first evaluates E, then negates it

```
evaluate[!E] =
  evaluate E
  iconst_1
  ixor
```

The unary not operator xors the expression with 1 to invert it.

```
evaluate[E1 O E2) =
  evaluate E1
  evaluate E2
  call O
```

All binary expressions except && and || are evaluated by first evaluating the left and right expressions and then call the specific operator function

```
evaluate[E1 && E2]
  evaluate E1
  iconst_1
  if_icmpeq g
  iconst_0
  goto i
g:
  evaluate E2
  iconst_1
  if_icmpeq h
  iconst_0
  goto i
h:
  iconst_1
i:
```

The and operator is a short circuiting and operator, that is whenever the first expression evaluates to false, the second expression is not evaluated at all. Whenever the first expression matches false it immediately jumps over the second expression. If the first expression is true, then the second expression is also evaluated. Whenever both are true, a 1 is placed on the stack. Whenever at least one expression is false, a 0 is placed on the stack.

```
evaluate[E1 || E2] =
  evaluate E1
  iconst_1
  if_icmpne g
  iconst_1
  goto i
g:
  evaluate E2
  iconst_1
  if_icmpne h
  iconst_1
  goto i
h:
  iconst_0
i:
```

The or operator is also short circuiting. Whenever the first expression evaluates to true, the second expression is not evaluated at all. If the first expression evaluates to true it immediately jumps over the second expression. If the first expression is false then it jumps to the second expression and that one is also evaluated. When at least one expression evaluates to true, a 1 is placed on the stack. When neither expressions are true, a 0 is placed on the stack.

```
evaluate[I] = with I a variable or unknown constants
iload i                    where i = variable offset of I
```

A variable expression or constant that is unknown at compile time loads the current value of the variable with the offset of I.

```
evaluate[I] = with I a known constant
ldc i                      where i is the value of the known constant
```

Whenever the value of a variable is already known at compile time, the value is loaded on the stack directly.

```
evaluate[while E1 do E2] =
  goto h
g:
  evaluate E2
  pop              if type(E2) not Void
h:
  iconst_1
  evaluate E1
  if_icmpeq g
```

A while expression first jumps over the body expression. The conditional is evaluated and whenever it is true, it jumps back to the body expression.

Chapter 8

Code description

Chapter 9

Testing

Chapter 10

Conclusion

Appendix A

jemoeder