Compiler Construction
Compiler Implementation for VYPlanguage Programming Language

Michal Horký (xhorky23):50
Matúš Múška (xmucka03):50

Extensions:
MINUS

December 2019

1 Front-end of the compiler

In this section, we will describe the front-end of the compiler, i.e. tools that were used, lexical analyzer information and grammar used for a syntactic analyzer.

1.1 Tools

The compiler is implemented in Python language using PLY, which is an implementation of lex and yacc tools for Python[1].

1.2 lex

A Lexical Analyzer Generator (lex) was used for the lexical analyzer. This section contains information about how the lex is configured.

1.2.1 Tokens

To configure lex the rules must be defined. In Table 1 are shown all tokens and corresponding regular expressions.

1.2.2 Unicode

In the lexical analyzer we check with regex `\x[A-Fa-f0-9]{6}` for escaped Unicode symbols and convert it to UTF characters. If Unicode character with specified value does not exist, a lexical error is returned.

[1] `http://www.dabeaz.com/ply/`

Token	Regex
PLUS	+
MINUS	-
TIMES	*
DIVIDE	/
ASSIGMENT	=
LPAREN	(
RPAREN	)
NAME	(?!(new|false|true|if|else|return|class|void|int|string|super|this— |while)b)[a-zA-Z_][a-zA-Z0-9_]*
CLASS	class
ELSE	else
IF	if
INT	int
NEW	new
RETURN	return
STRING	string
SUPER	super
THIS	this
VOID	void
WHILE	while
LBRACKET	{
RBRACKET	}
NEGATION	!
LESS	<
LESSEQUAL	<=
GREATER	>
GREATEREQUAL	>=
EQUAL	==
NOTEQUAL	!=
AND	&&
OR	||
COMMA	,
SEMICOLON	;
COLON	:
DOT	.
NUMBER	[0-9]+

Table 1: Tokens and corresponding regexes

1.3 Grammar

The following grammar is defined by comments, that are at the beginning of functions, and is generated by ply.

S' → *program*
function → *function_head statements_block*
statements_block → *LBRACKET statements RBRACKET*
function_head → *type NAME LPAREN functions_params RPAREN*
function_head → *type NAME LPAREN functions_params_empty RPAREN*
class → *CLASS NAME COLON NAME class_body*
statements → *statement SEMICOLON statements*
statements → <*empty*>
expression → *expression PLUS expression*
functions_params_empty → *VOID*
expression → *expression MINUS expression*
program → *init program_body*
functions_params → *type NAME*
functions_params → *type NAME COMMA functions_params*
statements → *if_statement statements*
expression → *expression TIMES expression*
class_body → *LBRACKET class_statements RBRACKET*
expression → *expression DIVIDE expression*
class_statements → <*empty*>
function_call → *NAME LPAREN function_params RPAREN*
init → <*empty*>
expression → *expression AND expression*
statements → *while_loop statements*
class_statements → *function class_statements*
statement → *RETURN expression*
statement → *RETURN*
expression → *expression OR expression*
function_params → <*empty*>
function_params → *expression*
function_params → *expression COMMA function_params*
class_statements → *type variables_declarationSEMICOLONclass_statements*
expression → *expression EQUAL expression*
program_body → <*empty*>
if_statement → *IF LPAREN expression RPARENstatements_block ELSE statements_block*
expression → *expression NOTEQUAL expression*
program_body → *class program_body*
while_loop → *WHILE LPAREN expression RPARENstatements_block*
expression → *NEW NAME*
expression → *expression GREATER expression*
program_body → *function program_body*
statement → *NAME ASSIGMENT expression*
expression → *class_function_call*

$expression \rightarrow expression\ GREATEREQUAL\ expression$
$statement \rightarrow type\ variables\_declaration$
$type \rightarrow STRING$
$type \rightarrow INT$
$type \rightarrow VOID$
$type \rightarrow NAME$
$statement \rightarrow class\_function\_call$
$variables\_declaration \rightarrow NAME\ COMMA\ variables\_declaration$
$variables\_declaration \rightarrow NAME$
$expression \rightarrow expression\ LESS\ expression$
$class\_function\_call \rightarrow NAME\ DOT\ NAME\ LPAREN function\_params\ RPAREN$
$class\_function\_call \rightarrow THIS\ DOT\ NAME\ LPAREN function\_params\ RPAREN$
$class\_function\_call \rightarrow SUPER\ DOT\ NAME\ LPAREN function\_params\ RPAREN$
$expression \rightarrow expression\ LESSEQUAL\ expression$
$expression \rightarrow class\_variable\_call$
$statement \rightarrow function\_call$
$expression \rightarrow NEGATION\ expression$
$class\_variable\_call \rightarrow NAME\ DOT\ NAME$
$class\_variable\_call \rightarrow THIS\ DOT\ NAME$
$class\_variable\_call \rightarrow SUPER\ DOT\ NAME$
$expression \rightarrow LPAREN\ expression\ RPAREN$
$statement \rightarrow class\_variable\_call\ ASSIGMENT\ expression$
$expression \rightarrow WORD$
$expression \rightarrow NUMBER$
$expression \rightarrow NAME$
$expression \rightarrow function\_call$
$expression \rightarrow LPAREN\ type\ RPAREN\ expression$

2 Back-end

2.1 Abstract syntax tree

When the whole program is parsed, the abstract syntax tree is created. Each block has function `get_instructions()` which recursively calls its children. When `get_instructions()` is called on the root of the abstract syntax tree instructions for the whole program is returned. In figure 1 there is a program represented in simplified abstract syntax tree that will print `"more than five"` if user write a bigger number as a 5 in stdin.

2.2 Build in Object and functions

Every compiled program has some built-in methods. Namely `subStr`, `*stringConcat`, `length`, `*readString`, `*readInt`, `*printString`, `*printInt`, and class `Object` with functions `toString` and `getClass`. Some functions with star (`*`) prefix are

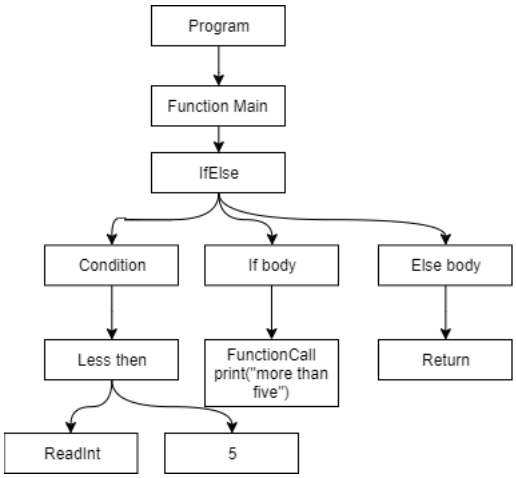

Figure 1: Example of program represented in AST

only for internal use. Thanks to our abstract syntax tree approach we could simply "program" these functions with building subtrees with our syntax blocks.

2.3 Expressions evaluating

Every operation (PLUS, MINUS, TIMES, DIVIDE...) is calculated on register with alias `Accumulator` and then pushed on the stack.

2.4 IF-ELSE condition

Block which represents if-else condition in abstract syntax tree have three parts. Condition, if body and else body. There are several labels in each condition which are used for jumping. Example of condition represented in VYPaCode, AST and instructions can be seen in figure 2.

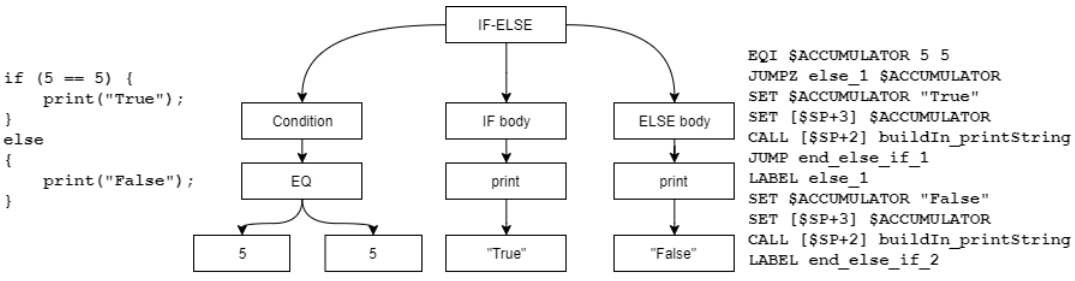

Figure 2: Example of the condition

2.5 While loop

While loops is represents which block that contains condition and body blocks. If the conditions is `False` jump to the end of the loop will be executed. Example of while loop that will print `"10987654321"` represented in VYPaCode, AST and instructions can be seen in figure 3.

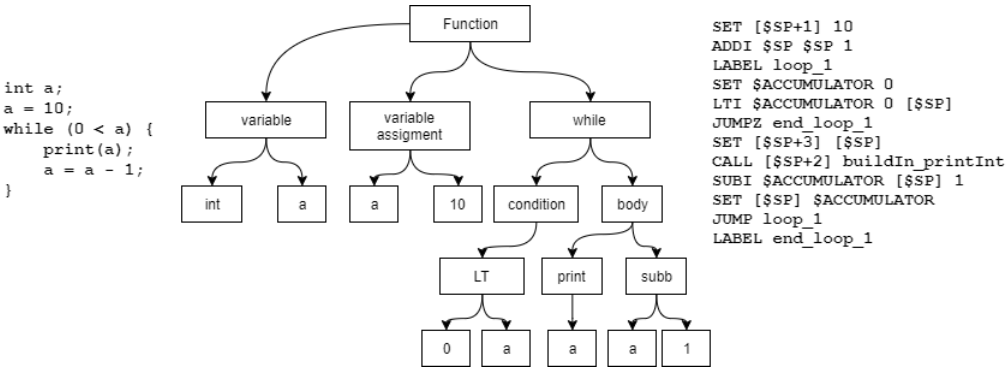

Figure 3: Example of the while loop

2.6 Function calling

Before function call, two blank variables and all its parameters are pushed on the stack. A return value of the function is pushed to the first blank variable and return address to the second. After the function call, all its parameters and variables are popped from the stack, and on the top of the stack is the return value of the function.

2.7 Classes

Every class instance has a special variable with the name `this`. After creating a new class instance, the position of the stack pointer is stored in this variable. Class variables are stored after `this`. An example of the stack after creating a new instance can be seen in figure 4. Object class have special variable `**runtime_name**` in which is stored type of the class instance and this variable is used only for inner purposes in functions `toString` and `getClass`.

2.8 Class function calling

When a class function is called, `this` variable is pushed as a zero parameter. Other class variable addresses are calculated as an offset of this variable (for example address of the first class variable is `[this + 1]`). An example of the stack when a class function is called can be seen in figure 5.

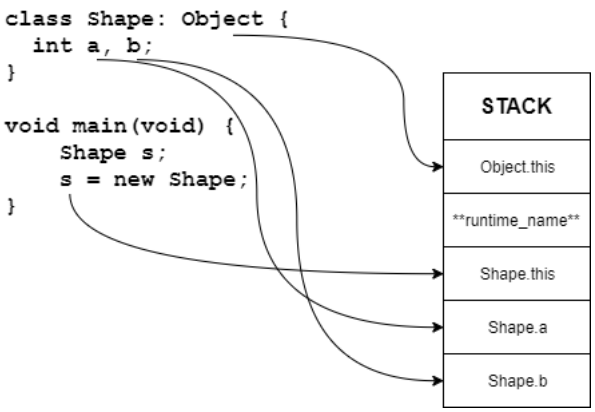

Figure 4: Creating new class instance

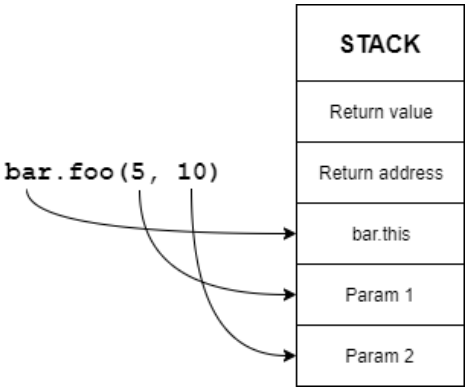

Figure 5: Class function calling

3 MINUS extension

Mathematically, the unary minus is normally given very high precedence - being evaluated before the multiply. However, in our precedence specifier, MINUS has lower precedence than TIMES. To deal with this, PLY library for python supports so-called "fictitious tokens". We used it for overriding default precedence of MINUS when it is before expression and no expression is on the left.

4 Run

Makefile is present because it is required by project specifications but does nothing. The compiler itself can be run by command `python3.6 app.py inputFile outputFile (optional)`. But, as the project specifications require, the compiler can be run with a `vypcomp` executable file (command `vypcomp inputFile`

`outputFile (optional)`).

5 Tests

A set of tests (41 tests) is included in the project. They are in the tests folder together with the `vypint` interpreter that is used to evaluate tests too. The tests can be run by `python -m unittest` command.

6 Work division

This project was made together in close cooperation. Michal Horký primarily made tests for the compiler, lexical analyzer and expressions evaluating. Matúš Múčka was responsible for abstract syntax tree and function calling. Class system was made together.

References

[1] D. M. Beazley. Ply (python lex-yacc). Accessed: 2019-12-12.