VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

VYPA – compiler

Vypcode Compiler

EXTENSIONS

IFONLY

MINUS

8. prosince 2024

Ladislav Dokoupil (`xdokou14`)

1 Technologies

The project was created using Python3.13 and can be run using the Python interpreter via this command:

```
python3 vypcode.py <input_file> [<output_file>]
```

where the output file is optional and defaults to `out.vc`. The grammar is defined using ANTLR4 [0] and can be found in `Vypcode.g4` file. `antlr4` lexer and parser are generated using this command:

```
antlr4 Vyp.g4 -o src/antlr_src
```

Furthermore, antlr4 creates listener class, which generates listener for entry and exit of each rule in grammar. This can be used to traverse the parse tree and create AST (Abstract Syntax Tree) or in our case, to generate code directly.

2 Implementations

In our approach we utilize syntax-directed translation, which means that translation is driven by the parser and the grammar rules. We use ANTLR4 to generate lexer and parser, which are then used to create a parse tree. We then traverse the parse tree generated by ANTLR4 using listener and generate code in the process. For most of the constructs we can directly generate upon entering the rule (top-down approach), for expressions, we need to wait until we know the result of the expression (bottom-up approach). Thus, exiting the rule is more suitable as we can forward information upward in the tree.

Because object/function usage can precede their declaration, we need to utilize a 2-pass approach. In the first pass, we gather all the information about objects and functions, and in the second pass we generate code. Vypcode is generated directly from the parse tree traversal (no intermediate code) and is written to the output file specified in the command line.

2.1 Main

Entry point of the program, where we parse command-line arguments and run lexer and parser. Translation errors are caught here and translated into appropriate error codes according to the project specification.

2.2 Syntactic analysis

Syntactic analysis is handled entirely by ANTLR4, and error handling is carried out in the main function.

2.3 Symbol table

On the 1st pass of the tree traversal, we gather all the information about objects and functions. The function symbols are stored in the global scope, and the object symbols are stored in separate objects. Upon 2nd pass, scopes are pushed/popped as we traverse the tree and enter/leave code blocks. These scopes are used to store local variables and to semantically check the program.

2.4 Semantic analysis

As mentioned above, semantic analysis is performed in the second pass of the tree traversal. Semantic analysis takes care of:

- type checking

- redeclaration
- usage before declaration
- function argument count and type
- object member access
- expression type checking
- type casting check
- *and more*

For type checking, we utilize class variable that stores the expression results (we can afford this because we use exit rules). For declarations, we check all symbol table scopes from the current scope to the global scope.

2.5 Code generation

Code is generated directly from the parse tree traversal, where for each rule, we generate appropriate code. Because the tree is in the process of being traversed and error may yet occur, we store the generated code in an inner class variable. Upon entering `exitProgram` rule, we can be sure that no errors occurred, and we write the code to the output file. With this in mind, let us go through some of the design decisions made for the generated code.

2.5.1 Function calls

After entering function we store a base pointer (BP) of a previous function. The new base pointer separates the function arguments from the local variables, where:

- BP[1] is 1st variable of function
- BP[0] is Base pointer for parent function
- BP[-1] is return address
- BP[-2] is last argument
- ...
- BP[-n] is first argument

Upon return to parent function, BP is restored and return to known return address is made. Callee also cleans up the stack. If return value is needed, it overwrites the first argument from the caller function, thus after return it will be on the top of the stack. The top of the stack is tracked by the SP (Stack Pointer) register.

2.5.2 Expressions

The expression results are stored on top of the stack. Because of this decision, we can treat unary and binary expressions as stateless (the first operand is on top of the stack, and the second operand is below it). Thus, if we encounter binary operations, we decrease SP and overwrite the second operand with the result of the operation. Similarly, if we call function (which pushed arguments on the stack), we can treat the function call as any other operand.

2.5.3 Variable declaration

When we encounter the declaration of the variable, we store the variable as a local variable on stack and its reference to BP. Each time expression with this variable is encountered, we copy its value from BP to the top of the stack. Upon assignment, we store the result in the original BP reference.

2.5.4 Virtual Method Table (VMT)

Each class has a virtual method table, which is a list of function pointers defined as strings in format `class:function`. Due to inheritance, the class can be a superclass of this class if it is not overridden. The function is dynamically called by loading the object pointer to the top of the stack and then loading the function name as a pointer of the appropriate VMT record.

2.5.5 Object member access and method calls

When an object is created, it creates memory on the heap as follows:

- 0: reference to self VMT
- 1: reference to parent VMT
- 2: string representation of the object (used for `getClass` method)
- 3: field 1
- ...
- n: field n - 2

When an object is accessed, we load the object pointer to the top of the stack and then load the field index from the object. When object method is called, we also load the object pointer on stack before calling the function, thus it becomes the first argument of the function, and this/super can be used in the function body.

2.5.6 Built-in functions

Built-in functions are hardcoded in the compiler and always added to the output code.

- `readString` - reads string from stdin
- `readInt` - reads integer from stdin
- `length` - returns length of the string
- `Object:toString` - returns string representation of the object
- `Object:getClass` - returns heap address of the class
- `__str_concat__` - hidden function for string concatenation (used by + operator)

3 bibliography

Terence Parr *The Definitive ANTLR 4 Reference* `https://www.fit.vutbr.cz/study/courses/VYPa/private/materials/parr-antlrv4_reference_2012`

Alexander Meduna, Roman Lukáš, Zbyněk Křivka. *VYPa presentations*.