VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Compiler Construction

Project – Compiler

20th December 2020

Tomáš Kožár (xkozar02) - 50%
Maroš Holko (xholko01) - 50%

1 Chosen technologies

Before implementation itself, we needed to choose programming language and optionally, lexer and/or parser generator.

We chose `Python` programming language version `3+`. It provides good abstractions and has good-sized "standard library". Syntax of language is also quite easy to learn so it was good compromise for us in terms of languages we knew.

For lexer and parser generator we chose `ANTLR4`. In our use-case there were no advantages to implementing these parts ourselves. `ANTLR4` is able to generate both lexer and parser with one grammar definition. It's also able to generate those parts to multiple "target" languages. One of which is `Python`. Parser that is generated is LR-parser.

2 Grammar definition

Both lexer and parser are generated from same grammar definition file. Our grammar is defined in `VYP.g4`.

In this definition files, parser rules are defined with *lowercase* rule names and tokens (lexemes) are defined with *uppercase* rules. In addition, tokens can be composed from *fragments* that are not fully-fledged tokens themselves.

Parser rules can be renamed which allows us to differentiate between different states while using same rules.

3 Lexical analysis

There is not much to describe for generated lexer. You only need to give it input program and it generates tokens in form that parser can work with.

4 Symbol table

Symbol table consists of currently most nested scope of variables. Each scope contains it's symbols that represent variables and parent scope. During search of symbol, most nested scope is queried. If it doesn't contain said symbol, it asks it's parent. Root scope has special parent that throws exception when it is queried since it means that symbol is currently defined in any scope.

Symbol table contains methods that allow to create or delete most nested scope.

Scope of variable is represented by class `PartialSymbolTable`. It uses built-in data type `dict` to store symbols. This data structure is known as *hash table*. Each symbol stored by it's name that is used as key in `dict`.

4.1 Symbol representation

Each symbol has stored it's `name` and `dataType`. Variables are represented by class `GeneralSymbol`, functions by `FunctionSymbol` and classes by `ClassSymbol` classes. For functions, `dataType` is it's return type. They also have list of variable symbols that represent their parameters. Classes additionally contain separate symbol tables for fields and methods and lastly their parent symbol.

Data types

Data types stored in symbols are either *string* for primitive data types or class symbol reference for non-primitive data types.

Function and class symbols have defined equivalence to other symbols. If they are equal, it means they are compatible. For example subclasses are equivalent to instances of same class and to instances of their direct and indirect parents.

5 Syntax analysis

Generated parser creates parse tree that you can traverse in order to perform syntax analysis itself. For this purpose, `ANTLR4` provides two types of tree traversal. First one is called *Visitor* and you are in control of visitation of tree nodes yourself. Second one is called *Listener* and `ANTLR4` traverses (walk) tree itself. You can implement methods that are called on either enter or exit of each parse tree node. We chose second approach since we thought it suits our needs better.

5.1 Two tree traversals

Since `VYPlanguage` allows usage of functions and classes that are defined lexicologicaly after usage, we decided to traverse tree first time to get all symbol definitions.

At end of first tree traversal, we perform check for existence of non-primitive data types and functions. At this point, non-primitive data types are represented just by *string*. However, during actual syntax analysis it's more convenient to have this data type represented by object. So all of the non-primitive types are checked for existence of said type and it is replaced by proper symbol.

At this point, variables are not checked at all. Since `VYPlanguage` supports scoping of variables and variable shadowing, it is more convenient to process them during second tree traversal.

5.2 Proper syntax analysis

During second tree traversal, proper syntax analysis is done and all function and class definitions are already available.

5.3 Implementation classes

As mentioned earlier, parsing is done `Listeners`. For custom behavior, we inherit generated class `VYPListener` in `CustomParseTreeListener`. This class implements general parsing behavior. Class `ExpressionListener` inherits our class `CustomParseTreeListener` and is used as actual class that is used to listen to tree traversal. This way we can divide implementation into individual classes. There two classed override mutually exclusive sets of methods for `VYPListener`.

Class `DefinitionsTreeListener` is used during first traversal.

`Symbol table usage` Symbol table itself was already described. During parsing, multiple symbol tables are used. For variables, `localSymbolTable` is used. It contains symbols for current function or method. Before entering each function definition, this symbol table is reinitialized with empty symbol table so symbols from last function are deleted. Functions are stored in `functionTable` symbol table. This table is filled in during first tree traversal and is only accessed during second one. Classed are similarly stored in `classTable` that is also filled during first traversal. Symbols that represent classes also contain two additional symbol tables for methods and fields.

All symbol tables, except for `localSymbolTable` are degraded since they don't need to work with scopes. Because of this, `PartialSymbolTable` class is used for their representation.

Functions and methods also contain their parameters. They are added to symbol table right at the beginning of function definition so they can be used right away.

Variable symbols

As mentioned earlier, it is more convenient to process variables in the second traversal. We don't have to reprocess non-trivial data types for all of the variables. That would also be quite difficult since we don't keep all of the variables. Each time we enter code block (scope of variable) we add new scope to symbol table. Equally, each time we exit code block (scope of variable) we delete most nested scope. Because of this, we cannot simply keep track of all the variables.

Shadowing variables

This feature meant little obstacle to us since we realized it at end of implementation. Getting right symbol for symbol table is simple since we start symbol searching from most nested scope. So right variable will be found. Problem is in target code since it get offset of variable by it's name in function (described properly later). For shadowed variables this means trouble.

ANTLR4 provides us position (line and column) of currently processed node in source file. For each variable, combination of line and column are unique when delimited by some character, in our case ':'. This is then used as prefix for variable name, but only for target code generation. Resulting name is in format {line}:{column}{name}.

5.4 Expressions

Expressions represent huge part of parsing itself. Since we are using Listener approach for parsing, we have no trivial way of knowing our position in tree relative to other expression nodes. For example we know that we are currently doing addition of two expression, but we don't know anything about those expressions.

This is solved using expressionStack stack for storing expressions. We are listening to exit of individual expression nodes. This corresponds to *postorder* tree traversal.

Expression object representation

There exist class for each type of expression category, namely BinaryExpression, CastExpression, UnaryExpression, LiteralExpression, ObjectExpression, VariableExpression, FunctionExpressions. Each one of them contain their dataType and other information needed for that expression type.

First expression nodes that will be exited are leaf nodes. In our case those are literal and variable values. For these nodes, expression object is created and added to expressionStack. Other types of expression pop required number of expressions (operands) from stack. For example, negation works with single operator so it pops single expression. Addition works with two operands, so it pops two expressions.

Since we already know that program is syntactically correct (AST was created), we don't need to worry about popping from empty stack.

Function expression

Processing of function expression is a little more complicated. Not only we have to listen for exiting of function call, we also have to process function call parameters and check their types. Also, function calls can be nested, meaning that you can use result of function call as parameter for another function call.

We solved this by using stack of lists for function call parameters functionCallStack. Each list contains expressions that are used for call. When new function call node is entered, empty list is pushed to stack. When function call node is exited, one list is popped.

Theoretically, expressionStack could be used for storing parameters. But we couldn't store number of parameters that was really used in call. That's why we chose our solution.

Object expression chaining

Since objects can be used as variable data types and function/method return types, it means that we can access object properties right through these expressions. We are storing this sequence in list. Storing whole sequence is probably not needed but we don't consider it a problem.

During chaining of objects, same problem as with function calls nesting emerges. We can use object chain in method call parameter. We solved this same way as earlier. Stack is used for lists of object chains.

During this chaining, data types are checked.

6 Semantical analysis

Our compiler doesn't have separate semantical analysis part, as is usually the case. Semantics are checked during second tree traversal as part of syntax analysis.

There exists class SemanticsChecker that was intended to do all of the semantic checks. It does most of them. Mainly expression operands types check, function call checks and method overriding checks (we did not implement overloading). However, during implementation this idea was followed and there are some ad-hoc semantic checks since they were easier to implement.

Class compability

Polymorphism is one of fundamental OOP features. Since VYPlanguage is statically types, we have to check compatibility of non-trivial (object) data types. Class `ClassSymbol` that is used to represent them has method that is used for checking of it's compatibility against another `ClassSymbol`. String values that contain class names are compared. If they are not equal, same check is performed for name of first symbol and parent of second symbol. When we get to *Object* parent, we no longer compare it's parent but comparison failed.

Function calls

There are two semantic checks that need to be done before function call. First one is checking number of parameters that are used for call against number of defined function parameter.

Second one is checking type compatibility of individual parameters. This check is trivial.

7 Code generation

We chose direct code generation approach for our compiler. We decided not to implement any optimizations so we even skipped immediate code generated and jumped straight to generating target code. Another reason is that it is faster.

Class that acts as interface for code generator is `CodeGenerator`. In VYPlangage all code is structured inside of functions/methods and in classes for member variables. Because of that, our generated code is mostly in functions. Except for virtual method tables.

Each function has corresponding instance of class `FunctionCodeGenerator` that stores it's generated code and contains methods for it's generation. `CodeGenerator` is responsible for proper addressing of individual instances.

Important register that is used during generation is `$FP - function pointer` that contains index of stack where function data begins. It is set right after `LABEL` of function is defined by copying value in register `$SP`.

Variables

Class `FunctionCodeGenerator` contains list of defined variables. When variable is defined, it is appended into this list and it is set with default value into stack. Position of stack is evaluated by adding `$FP` and index of that variable in list. Similarly, when we want to access value of variable, we get index in stack same way.

Function calls

Stack that is available is VYPcode is shared among all of the function calls. For accessing variables, we need proper `$FP` value. However, that is problem since called function will override it so it can work with it's variables. Because of this, it's current value is saved on stack right before calling another function and is restored right before returning from that function.

Function parameters

Before function is called, values that are used as parameters are pushed onto stack. Problem is that we cannot access parameters same way as variables. Class `FunctionCodeGenerator` contains another list of variables for parameters. When these parameters are being accessed, their index in stack is computed similarly as variables index, but instead of adding their position in list to `$FP`, it is subtracted.

Embedded functions

These functions are written as string. Methods for generation of user defined functions are not used.

print

Function `print` was quite tricky to implement. That is because it can accept both *string* and *int* and number of these parameters is not predefined. We solved this by not implementing it as normal function. For each parameter, instructions `WRITEI` or `WRITES` are generated, depending on their type.

7.1 String representation

In our implementation, each *string* is represented by chunk and word values on indexes represent characters of said string.

7.2 Object representation

Objects are represented as chunks as well. First word in chunk (0th index) contains ID of chunk with name of it's class. Second word contains ID of chunk with it's VMT and third word is ID of parents VMT. After that, words represent field members of class in same order as they were defined. Their indexing is nearly same as for function variables.

7.3 Virtual method table

We need unique names for individual methods since they can have same name in VYPlanguage thanks to overriding. In VYPcode, name that is used is generated by using class name as it's prefix with delimiter symbol ':'.

VMTs are generated using `VirtualMethodTableGenerator`. There exists only one instance of this class. It contains mapping for classes and their VMT and list of it's methods.

Individual VMTs are represented using chunks as well. Each word contains ID of chunk with string of that method used for function calling.

VMTs are stored at most bottom of stack. Their index is gotten from `VirtualMehtodTableGenerator`.

Indexing of methods

It is crucial to be able to properly index in VMT so we don't call wrong method.

When class is defined, it's method symbol table is prefilled with contents of parent's method table. Python `dict()` data structure has ordering based on order of inserting of values. Thanks to this, subclasses have same order or inherited methods as their parents.

Expressions

For leaf node expressions, value is set to stack on index of `$SP` and `$SP` is incremented. Expressions that use operands read their values from stack and set result to index of most bottom operand. `$SP` is set to one above of that index.

Function return values

Before returning from function, it sets return value to it's most bottom parameter on stack and sets `$SP` to one above it.

Object instances

Default value of variable of non-trivial data type is 0 which corresponds to kind of NULL value.

When new instance is explicitly created, new chunk is created for said instance. Values are set as they were described sooner. After that, chain of constructors is invoked, if any are available.

8 Testing

We wrote testing script for our compiler together with tests. Tester is implemented in file `tester.py` and tests are located in directory `tests`. Tests are defined by two files. First one contains source code (`.test` postfix) and second one contains expected return codes and value on stdin (`.result` postfix).

9 Work division

I, Tomáš Kožár (xkozat02) worked mostly on front-end of our compiler. Meaning definition of grammar and syntax analysis together with semantic checks.

Maroš Holko (xholko01) worked mostly on code generation and testing of our compiler.

10 How to run

Makefile does nothing since nothing needs to be done. Main entry point of our compiler is in file `src/main.py`. We created script `vypcomp.sh` that should be used to run our program (as per assignment). This script also sets `PYTHONPATH` for `ANTLR4` library since it is not set by default. It also cannot be set by Makefile since it is environment variable.