Lecture: Compiler and Language-Processing Tools
Winter 2020/2021
Dr. Annette Bieniusa
FB Informatik, TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Portfolio Exam

January 11 – February 22, 2021

BHALACHANDRA GAJANANA BHAT

Eigenständigkeitserklärung

Mit Einreichen des Portfolios versichere ich, dass ich das von mir vorgelegte Portfolio selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Mir ist bekannt, dass Plagiate einen Täuschungsversuch darstellen, der dem Prüfungsausschuss gemeldet wird und im wiederholten Fall zum Ausschluss von dieser und anderen Prüfungen führen kann.

Declaration of Academic Honesty

By submitting the portfolio, I confirm that the submitted portfolio is my own work, that I have fully indicated the sources and tools used, and that I have identified and referenced the passages in the work - including tables and figures - that are taken from other works or the Internet in terms of wording or meaning. I am aware of the fact that plagiarism is an attempt to deceit which will be reported to the examination board and, if repeated, can result in exclusion from this and other examinations.

Documentation

Problem Statement.

NotQuiteJava language is a subset of Java and it's features/functionalities are listed in the portfolio. The portfolio refers to the implementation of compiler which includes parsing, type checking, translation etc. This project mainly focuses on the implementation of Name and Type Analysis along with Translation for the language NotQuiteJava.

Implementation Method

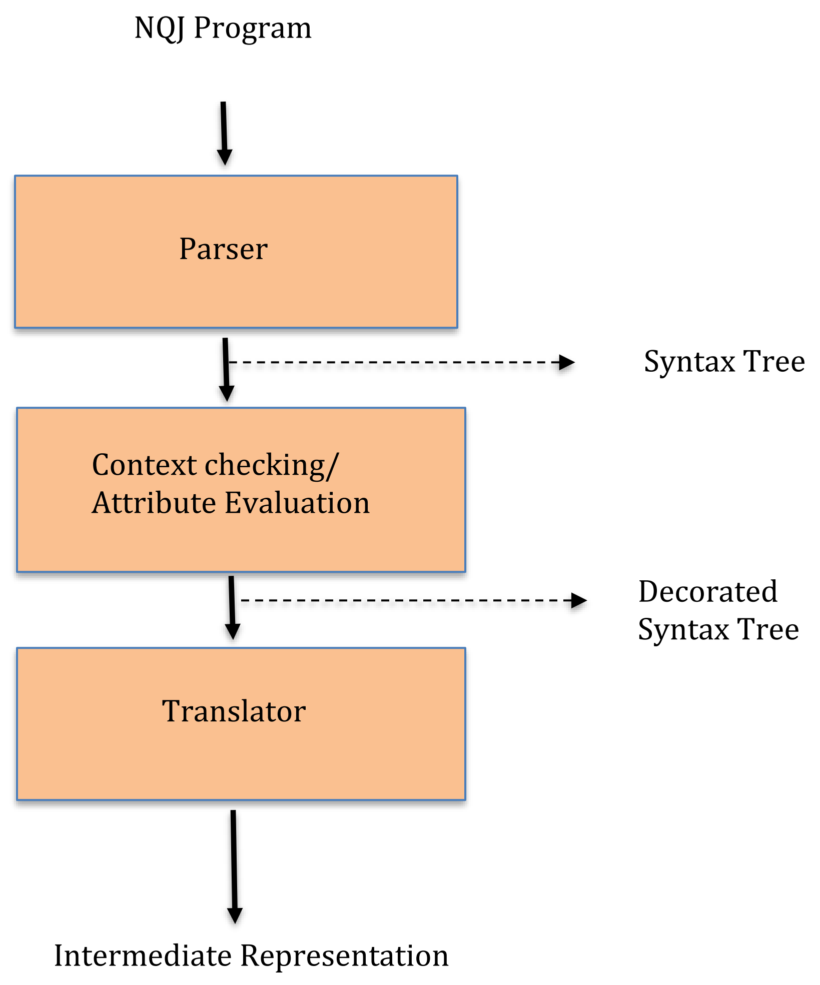
NQJ Program
Parser
Syntax Tree
Context checking/ Attribute Evaluation
Decorated Syntax Tree
Translator
Intermediate Representation

Fig 1. Abstract design of the portfolio

As depicted in the Fig (1), Parser takes NotQuiteJava program as input and perform lexical analysis and syntax analysis which produce syntax tree. The output from this stage is passed on to the next stage for further processing. In the Name and Type Analysis phase which is one of the main focus of this document, the obtained syntax tree is type checked and is further processed to produce a decorated syntax tree where the relation between the symbols are linked. This phase is the last phase of front end. Later translator converts the obtained decorated syntax tree into intermediate representation form. Overall, once NQJ program is passed to the system, the system will convert it to LLVM code which is a machine native code.

1. NAME AND TYPE ANALYSIS

This phase catches the errors which are missed in the previous phase. It is also known as semantic analysis. It helps in interpreting the structures and symbols obtained from the parser. This document discusses on the task performed in this phase along with the techniques and data structures used for corresponding tasks.

The data structures used to achieve the tasks are defined below:

1. **ArrayList<TypeError> typeError:**

   This is used to store the errors caught in this phase. At the later point of time this buffer is used to display the errors in console.

2. **LinkedList<TypeContext> ctxt:**

   This data structure is used to check the validity of the variable in current scope.

   "TypeContext" has a Hash Map which is used to store the variable names and its declarations. In each scope new object is initialized and pushed to LinkedList.

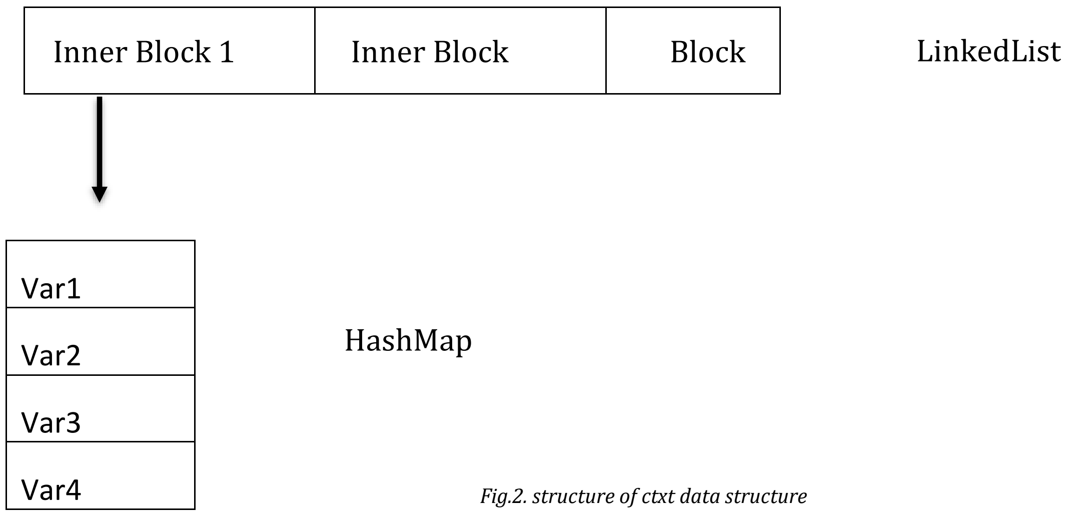

Fig.2. structure of ctxt data structure

This operation also could be achieved using stack data structure. It follows LIFO for operations but iterations through stack doesn't serve the purpose as it traverses from the bottom of the stack and follows FIFO. Copying the entire data structure and popping out the elements would be expensive. So linked list is the suitable data structure as it maintains the insertion order. Thus, FIFO can be achieved.

Hash Map is used to store variables as the keys are unique and also the search operations in the Hash Maps are not expensive.

3. **NameTable(class):**

   This class contains set of data structures which are used to store global static functions, classes and array types. The data structure used inside this class is explained below.

   1. HashMap<Type,ArrayType> arrayType:

      This data structure is used to store different array types declared in the program. It stores the base type like int as key and stores the array type as value for future references.

   2. HashMap<String,NQJFunctionDecl> globalFunctions:

      It is used to store global static functions with function name as key and function declaration obtained from syntax tree as value. This is stored in Hash Map as it doesn't allow duplication of keys which ensures the function names to be unique.

   3. HashMap<String,ClassObjects> classList:

      In this Hash Map the class names are stored as key and *ClassObjects* class is stored as the value. *ClassObjects* stores the method and field of the classes in Hash Maps. The data structures that are used inside the class is defined below.

      1. HashMap<String,NQJFunctionDecl> classMethods:

         This is used to store the methods of the classes. It takes method name as the key and method declaration as the value. The use of Hash Map prevents the duplication of the method name. The NQJ does not allow method overloading. Hence Hash Map is best suit for this type of operation. Later these method declarations can be accessed to interpret symbols and tokens obtained in syntax tree.

      2. HashMap<String,NQJVarDecl> classFields:

         The class variables are stored in this data structure which takes variable name as key and its declaration as the value. Usage of Hash Map helps in keeping all the class variables unique. The class fields can be later accessed by lookup operations defined.

4. **Stack<NQJClassDecl> curClassList:**

   This contains the present class along with its parent classes when system is processing the particular class. This data structure is defined to obtain the current class (this expression) along with the methods and fields from its super classes.

5. **HashMap<String,String> inhMap:**

   This contains (childclass,ParentClass) in pair. This data structure is mainly designed to check for the cyclic dependency of classes which could. Hash Map is suitable as this can be reused to throw error when a class tries to inherit from multiple classes which is not allowed.

The important tasks performed in this Name and Type Analysis phase using the above data structures are mentioned below.

1. **Check for the declaration of identifiers.**

   The declaration of all the variables is stored in the data structure *ctxt* with respect to its scope. Availability of the variable is checked using lookupVar() operation while processing it.

2. **Type Checking.**

   The expression is evaluated for each statement, Later the type of the end expression is searched in respective data structure. Error will be added to type Error if there is a mismatch between types. This is made possible by overriding isSubTypeOf() function from the Type Class. The type of the class is represented using ClassType which inherits Type class. This class also contains a field for super class type which is formed from the data structure Set. In case of simple inheritance or multilevel inheritance, this field helps in type checking.

3. **Scope resolution.**

   This feature is implemented with the help of *ctxt* (LinkedList<TypeContext>). TypeContext class contains a Hash Map where the variable declaration is stored. Every time new TypeContext is created when program enters a new block. If the variable declaration is not found in the current scope, the variable will be searched till the iterator reaches the end of the LinkedList *ctxt.* Then variable will be linked to its corresponding declaration.

4. **Simple Inheritance**

   The child class inherits the methods and fields from parent class. The parent class methods can be called using the object of child class. Hence child class should load all the properties of super class. This is achieved with the help of HashMap *classList*. In this project this is achieved through a phenomenon similar to *two pass check.* In the first pass, the declaration of all classes and methods are loaded into the buffer *classList*. Later this buffer is used to assign the methods and fields of super class to child class except *overridden method*. The implementation has taken care of overridden method by ignoring the overridden method while copying the data from the map. Check for *Multilevel Inheritance* is also implemented in this phase.

5. **Check for cycles in inheritance.**

   The hash map *inhMap* is designed mainly to implement this feature. This map contains is updated in the first pass where class declarations were updated. This buffer is used to compute the existence of cycles by traversing through the map till the end. The program will be having cycles when passed argument equals the parent class name. It is achieved in the function isCyclicDependecy(). The check with this function also helps in fetching all the super classes in case of multilevel inheritance. Otherwise, the program will run for eternity.

6. **Uniqueness of class, method and field names**

   Hash map is used to store class, method and variables with their names as key. As the hash map doesn't allow duplication, uniqueness is preserved.

7. **Method Overriding (signature matching)**

   Method overriding is a run time phenomenon but the signature of a method in derived class should match with its parent class method signature. This makes use of already available buffer *classList* and *inhMap.* The list of child classes is obtained from inhMap. Later with this information overridden methods are fetched from classList and stored in set using java *retainAll* functionality of set. Function signatures are checked with the available inputs from previous step along with classList hash map.

8. **Check for return statements**

   This feature is not dependent on any of the data structure mentioned above. In this case the last element of the function is passed to a recursive function which checks all the possible paths of return statement.

This phase will produce the decorated syntax tree, once all of the above operations are completed. The declaration of the variable, classes and the methods are assigned to respective object. Later this output is passed to translator phase for generating the intermediate representation of the input given.

2. Translation of Object-Oriented Constructs

The program obtained from the previous phase is converted to intermediate representation form. Later this can be executed. The implementation part for this phase is not fully functional. The idea behind the working of Translator is explained below.

The data structures used to achieve translation of object-oriented constructs is explained below.

1. Map<Type,TypeStruct> classList:

   The data structure stores the class type as key and the structure as value. The structure defined will have reference with the virtual method table using which the methods can be loaded. Structure type also contains its fields along with the fields of super class.

2. Map<Type,TypeStruct> vTable:

   vTable keeps record of pointers to all the methods of class which are initiated and translated. It stores class type as key and its structure as value. The structure contains pointer to all the procedures. The methods of super classes are not stored in virtual method unlike the case of name analysis where the methods are duplicated in the implementation part.

3. Map<TypeStruct,Map<NQJFunctionDecl,Proc>> methodList:

   methodList keeps track of the declaration of all the procedures with respect to virtual method table. The virtual method table is considered as key and the list of all the procedures are stored as values.

4. Map<type,Map<NQJVarDecl,temporaryVar> classVariables:

   In this data structures the field of current class and its declaration is stored. This will be used for future references.

Conclusion

The project implemented all the functionalities discussed in Name and Type Analysis phase. The project implemented name table using Linked List and Hash Map data structures. It processed the tokens obtained from parser and sends the updated syntax tree to the next phase. The translation phase is not fully functional in the implementation part. The idea behind this is discussed.

References

- https://stackoverflow.com/
- https://www.geeksforgeeks.org/
- Appel: Modern compiler implementation in Java.
- https://softwareengineering.stackexchange.com/