Compilers Project

The aim of the project was to write a compiler for the Decaf language. Decaf is a simple imperative language similar to C++.

Tools Used:

- flex
- bison
- llvm

Flex:

Flex is a fast lexical analyser generator. It is a tool for generating programs that perform pattern-matching on text. Flex is a free (but non-GNU) implementation of the original Unix(*lex)*program.

Bison:

Bison, is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) which reads sequences of ttokens and decides whether the sequence cosnform to the syntax specified by the grammar. Bison by default generates LALR parser but can also create GLR parsers.

llvm:

The **LLVM compiler** infrastructure project (formerly **Low Level Virtual Machine**) is a "*collection of modular and reusable compilers and toolchain technologies*"used to develop compiler front end and back end.

Flow of work between flex and bison:

Bison fundamentally works by asking flex to get the next token, which it returns as an object of type "yystype". But tokens could be of any arbitrary data type. So to deal with that in Bison by defining a C union holding each of the types of tokens that Flex could return, and have Bison use that union instead of "int" or other types for the definition of "yystype".

Defining types and tokens:

We need to define the "terminal symbol" token types we are going to use (in CAPS), and associate each with a field of the union. Also we need to define every keyword that is returned by lex file for any matching regex so that when used in grammer defination bison knows that this is returned by flex.

Defining rules and grammer:

In flex file we define regex corresponding to language grammer and return token corrresponding to each, in bison we define actual grammer rules using tokens(terminals) returnedby lex and follow up non terminals in rules. In this we used left recursion in grammer rule.Processing of input file:
We assign input file pointer to yyin ie. yyin points to input file now by using yylex() we refer to flex and yyparse() parses the input file line by line and yyerror() to let us know about error during parsing incase it exist.

The compiler construction was done in 3 phases :

- **Phase 1** : Parser to parse Decaf production rules.
- **Phase 2** : AST for the parser built.
- **Phase 3** : Building LLVM for the AST built.

Phase 1

This involved building a lexical analyzer using LEX and a parser using Bison.

Challenges faced

- Biggest challenge faced is in writing the production rules without ambiguity in the grammar.
- Lex and Bison have a very specific way of input in every version, hence it was hard to identify the correct syntax in the specific installation of ours.
- There was also a problem of shift reduce errors and warnings in the grammar.

Phase 2

Phase 2 includes creation of AST from the grammar and lexer produced before. This includes creation of whole hierarchy of classes to create the AST and use a Visitor Design pattern to create an XML.

- A big challenge was to include Visitor Pattern into the existing AST class hierarchy.
- As the reductions happen in the way, the XML will have the number of variables in the class and the values.

Phase 3

This phase required generating the LLVM IR for the AST. For this we had to specify the Codegen() method in each class, and then call the codegen() method of the root node. This would result in code generation of the entire AST(by way of recursion).

Understanding the LLVM API took us a lot of time and effort, as it was completely new. This was the ***most challenging part*** of the project. However we progressed incrementally, from generating IR for constants to generating IR for for loops and so on.

The **process** we followed was

- First generate the IR assuming there are no mutable variables. In this case, the only variables in a method where its parameters.
- Assuming this, IR for Binary operations, method declarations, method calls, if else then expression, and forloop expression were added in the given order.
- After that, it was modified to allow the declaration of local mutable variables and also, assignment of already declared variables. Mutable variables were dealt with using location binding of the variables.
- Then, we tried to add the global variables support, which includes arrays also. It was pretty difficult as there was no documentation available for it. We were able to successfully generate the IR for the declaration of the global variables, but due to the time constraint, could not progress further in this.

Designing AST:

For designing AST from grammar we use visitor design pattern and the concept of double dispatch that make our work easy to understand and write AST.
In this we defined a visitor class which have pure virtual printing function which takes pointer to different class as argument and from each function the corresponding virtual functions are called from every class in order to print data for that class and call other class function from that class.

In this we also defined class that represent each non terminal in grammar and that class handles making of tree node of that particular type and also we defined list for accumulating all same type node under one node.

We traverse through the grammar and at each point accordingly create node and that node will call subsequent non terminal class in order to make node of that type in this way we build Abstract syntax tree.

Shift Reduce Conflicts nd their resolution:

we have face many shift reduce conflicts in this project. One of the main reason we got shift reduce conflicts was when a string was being generated from two different paths. While putting newline character and comments we got many shift reduce conflicts. To resolve that we made a specific grammar rule for endline and comments.

Shift reduce were happening when there was a point when on the same input token the grammar rule was reducing as well as shifting that stream.

Case:

When we are parsing a language which has if-then and if-then-else statements, with a pair of rules like this:

if Block: IF expr THEN stmt
| IF expr THEN stmt ELSE stmt
;
Here IF, THEN and ELSE are terminal symbols for specific keyword tokens.

When the ELSE token is read and becomes the lookahead token, the contents of the stack

(assuming the input is valid) are just right for reduction by the first rule. But it is also legitimate to shift the ELSE, because that would lead to eventual reduction by the second rule.

This situation, where either a shift or a reduction would be valid, is called a shift/reduce conflict.

AST Class Hierarchy:
root---- child
program Class

- fieldDecClass
    - idClass
    - idVarClass
    - idArrayClass
- methodDecClass
    - argClass
    - blockClass
        - varDecClass
            - idClass
            - idVarClass
            - ArrayClass
        - statementClass
            - assignmentStmtClass
                - assignment_class
                - exprClass
                - locationClass
            - method_callClass
                - exprClass
            - continueClass
            - breakClass
            - returnClass
            - forClass
                - assignment_class
                - exprClass
                - blockClass
                - loctionClass
                    - exprClass
            - ifElseClass
                - exprClass
                - blockClass

        - blockClass
            - varDecClass
            - statementClass
- exprClass
    - operatorClass
    - literalClass
        - int_literalClass
        - char_literalClass
        - bool_literalClass

exprClass—(operatorClass,literalClass)

literalClass—(int_literalClass,char_literalClass,bool_literalClass)

operatorClass--(exprClass)

visitor--(display)