Compilers Project

Decaf compiler

Mohit Sharma
201505508

Phase 1 and 2

Our objective for phase 1 & 2 is to implement the syntax analyzer and parser for the Decaf programming Language. Analyzer should be able to parse all valid programs and give an error for invalid program. Check Decaf manual to get an idea about the language.

Tools used:

1. **Flex:** Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers").

   In simple words we can consider it as a tool which provides an easy way to specify the regular expressions corresponding to lexemes expected in source file. It also provides us a way to specify the action to be taken when we encounter a particular lexeme. In action part we can do operations like printing the lexeme (text matched) or returning corresponding token to the parser program. Check the following example. Here we have defined the regex for “if” and returning corresponding token in action part.

```
%{
        /* Global declarations and control information*/
%}
IF      "if"
%%
        {IF}      { foutlex<<"IF"<<"\n";yylval.sval = strdup(yytext); return If; }
%%
```

   The token we are returning here are specified in the parser file (bison file) explained below. We pass this .l file to flex to generate corresponding C code for lexical analyzer which can scan any text for the lexemes specified in .l file. The output file from flex is named “lex.yy.c”.

2. **Bison:** It 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 tokens and decides whether the sequence conforms to the syntax specified by the grammar. Bison by default generates LALR parsers but can also create GLR parsers. We specify the grammar and tokens that are needed for that grammar in a .y file. We provide this .y file to bison as input and it generates two files “.tab.h” (to be included in flex code, it contains all the terminal token declarations that flex will return when it encounters the

lexeme corresponding to that token) and ".tab.c" (contains the parser code). To understand how to specify the grammar specifications, consider following dummy grammar which contains just one string "x y".

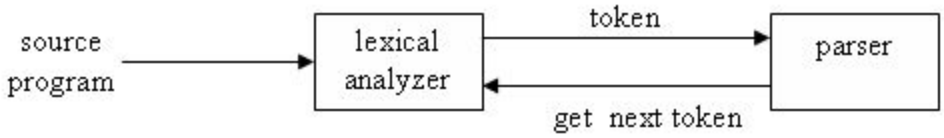
```
%{
        /* Global declarations and control information */
%}

// Union for data types of tokens that can be returned by flex
%union
{
        int ival;
        char *sval;
}

// Different tokens that can be returned by flex
%token <sval> X Y
%%
        // Grammar that'll be parsed by bison
        Program         :       X Y     {fout <<"PROGRAM ENCOUNTERED\n"; };
%%
```

In this grammar we are expecting just two lexemes or terminals: "x" and "y". Both are specified as tokens. The same are used in the only production rule. The parser generated will work in bottom up manner i.e. it'll first try to match X then Y, if it is able to complete the rule "X Y", reduction will happen and matched rule will be replaced by corresponding LHS i.e. "Program" in this case. The moment it is able to reduce to start symbol, parser stops successfully. Whenever it reduces, the corresponding statements specified in the action part will be executed. If any unexpected symbol is encountered it calls the error handler "yyerror()".

For complete codes for flex and bison check git repository link at the end.



Basic diagram to understand the interaction between lexical analyzer and parser.

Issues faced:

1. It took some time to understand how lex and bison interact with each other.
2. **Shift/reduce warnings:** These warnings can occur whenever grammar contains productions which can be reduced to a non-terminal and also one more symbol can be shifted instead of reducing it. If we don't specify the precedence and associativity of operators then there will be many shift/reduce warnings. The token specified earlier has lower precedence than that of specified later. For specifying associativity , we can use *%left, %right* and *%noassoc*. E.g. following three tokens are left associative.

   %left Multiplication Division Remainder

2. **Same lexeme to be used in two different contexts (binary/unary minus):** Both the minus are represented by '-' but we can't return two different tokens from flex corresponding to same string '-'. We need a way to use same token as per context. It can be done by specifying a dummy token for Unary minus just to specify that it has higher precedence than binary minus. Same is specified in the production using %prec. Check following example for better understanding:

   ...
   %left '+' '-'
   %left ''*
   %left UMINUS

   Now the precedence of UMINUS can be used in specific rules:

   exp:
   ...
   | *exp '-' exp*
   ...
   | *'-' exp %prec UMINUS*

Phase 3:

Our objective for phase 3 is to build the Abstract Syntax Tree (AST) for the parsed program. Then to traverse the tree to produce nested XML output which represents the structure of the program.

Input:

```
class Program
{
        boolean field;
        int x[10];
        int foo(int y)
        {
        }
}
```

Output:

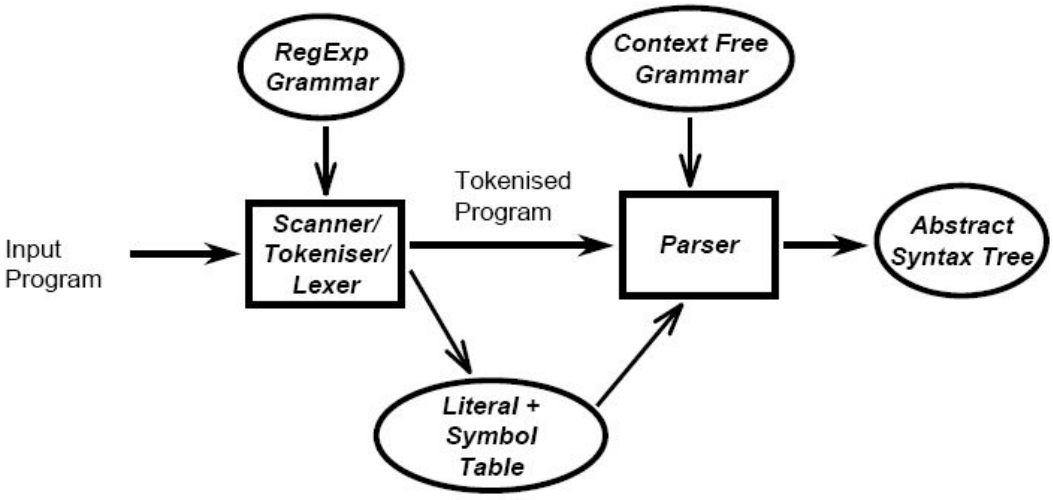
```
<program>
        <field_declarations count="2">
                Normal
                <declaration name="field" type="boolean"/>
                Array
                <declaration name="x" count="10" type="integer"/>
        </field_declarations>
        <methods count="1">
                <method name="foo" return_type="int">
                        <params>
                                <param name="y" type="int">
                        </params>
                        <locals>
                        </locals>
                        <body>
                        </body>
                </method>
        </methods>
</program>
```

Steps:

1. Create a base class “Node”.
2. Create a class corresponding to every non-terminal symbol in grammar. All these classes will inherit Node class. Every class contains a Node* for each of its children.
3. In bison file create a node corresponding to every production rule and assign it to $$ so that it can be passed to its parent rule. E.g. :

```
Identifier1
        :       Identifier
                { $$ = new Identifier1Node(new IdentifierNode($1), NULL); }
        |       Identifier OpenSquare Int_literal CloseSquare
                { $$ = new Identifier1Node(new IdentifierNode($1), $3); };
```

4. Now follow visitor pattern to display output corresponding to every node. It’ll help to avoid type castings to be done manually from Node* to any of its child classes. Type casting is required so that we can map correct display() method to each node. The task is can be done easily using visitor patterns. It utilizes run time binding using virtual functions and function overloading technique to avoid explicit type castings.
5. Implement the functionality to display appropriate output in corresponding visit methods.



Issues faced:

1. Deciding about class structure for every node was a challenging task. But solution was simple to implement if we just create a class for every symbol on left of production and making the symbols on right as its members /children.
2. Visiting pattern made the task of calling appropriate visit method depending on node value, simple but to generate the correct XML structure some effort was required. As

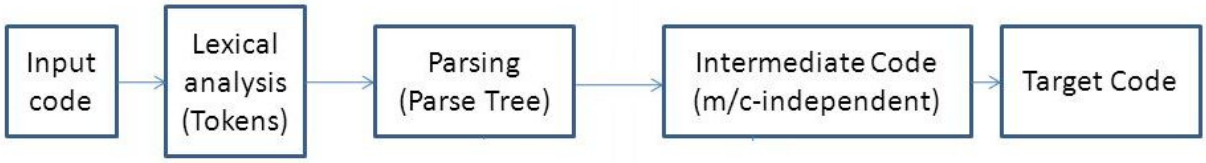
we want to display in such a way that output represent the program in easily understandable form.

Phase 4:

Our objective for phase 4 is to generate LLVM IR for the Abstract Syntax Tree (AST) generated in phase 3.

Tools used:

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



Issues faced:

1. It took the most amount of time to understand LLVM API and how to use them as not much examples are available for the same except LLVM documentation on llvm.org. The documentation is very difficult to understand as no examples are mentioned.
2. There are multiple versions available for LLVM. All versions vary a lot in terms of header file's locations. It was also difficult to find that on which version the any sample program will run. Finally I started implementation on 3.4. It took lot of time to make a sample program run.
3. How to store the variable/parameter information i.e. how to implement symbol table was another challenge. Finally used a map to store the information.
4. As we can have nested basic blocks so generating code for such a program required stack implementation.
5. Implementation for if-else-then, for loop was also time consuming as not many sample examples were available for them.

Limitations:

1. Not much error handling is done. It'll report errors like "variable/function not declared". But errors like "type mismatch, incorrect parameters passed to function" aren't checked. If any syntax error is there then it'll print "Syntax error".

References:

1. https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator)
2. https://en.wikipedia.org/wiki/GNU_bison
3. http://www.gnu.org/software/bison/manual/html_node/Contextual-Precedence.html
4. http://ashimg.tripod.com/Parser.html
5. http://aquamentus.com/flex_bison.html
6. https://en.wikipedia.org/wiki/Visitor_pattern
7. http://llvm.org/docs/LangRef.html
8. http://llvm.org/docs/tutorial/index.html

Code repository

https://github.com/thegame61916/DecafCompilerProject