Decaf Compiler

(CSE419 Compilers – Project Report)

Spandan Veggalam
201307674

Abstract

Decaf is simple imperative language similar to C or Pascal. We design and implement a compiler which translate Decaf language source to LLVM intermediate representation and is executed using LLVM interpreter.
Our compiler uses flex, byacc-j for implementing parser, Jllvm3.2 (with LLVM3.2) for constructing LLVM intermediate representation (IR).
With the help of visitor design pattern we have separated, AST to LLVM IR translation logic operated on the AST classes.

Introduction

As part of Compilers course (CSE 419) we develop compiler for Decaf language, an imperative language which is similar to C or Pascal.

Objectives:

1. We formally describe the properties of Decaf language, and consider this formal description of Decaf language to develop a compiler in Java.
   We choose Java because it is a popular programming language in wide spread use and active development. It is also a fairly open standard.

2. We will also brief our implementation and framework.

3. We use clang as alternate compiler which in turn uses LLVM, to compare if our implementation is correct.

Implementation

1. Scanner and Parser

Scanning:

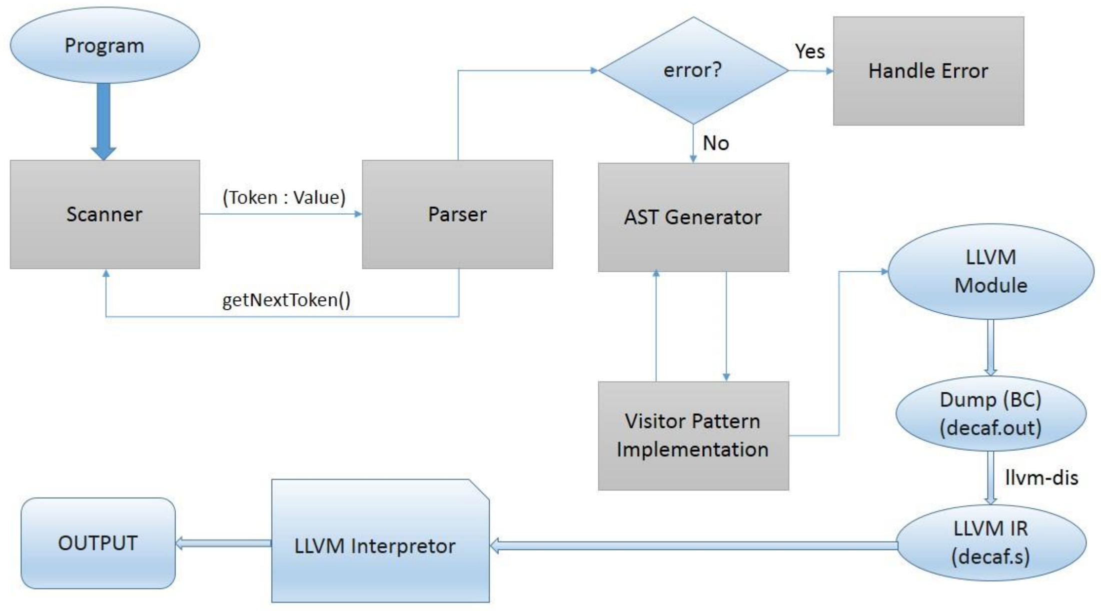
We have used jFlex (Java version of lex) tool to generate scanner, which reads the input tokens and passes it to Parser which constructs parse tree. Once the input token is read, action defined to corresponding regular expression definition gets executed. In our case on successful match of the token, token type and its value is returned to Parser.
Regular expressions for different tokens are defined in definitions section of lex input file. Rules are defined in declarations section along with their corresponding action.
The flex input file consists of three sections,

i. Rules

ii. Definitions
iii. User code

Parsing:

We have used byaccj (Java version of Yacc) to generate parser. Parser specification file is defined based on the language grammar.
All the productions are defined in this file along with the actions that are to be taken before reducing the productions. On successful reduction of a production, action statements are executed.



Decaf Compiler Architecture

2. AST Generation

We used Visitor design pattern which separate the logical part (i.e., functionality that has to be applied on the AST hierarchal elements) from AST elements. (Note: Refer Appendix for AST class hierarchy)
The implementation proceeds as follows.

a. virtual accept() method is added to the every base class in AST hierarchy.
b. accept() is defined, receives a single argument which is a reference to the abstract base class of the Visitor interface hierarchy.
c. Each concrete derived class of the Element hierarchy implements the accept() method by calling the visit() method on the concrete derived instance of the Visitor hierarchy that it was passed, passing its "this" pointer as argument.
d. For each derived class in AST hierarchy, visit() method is implemented as part of visitor patter implementation. During which we perform semantic checks and generate LLVM Intermediate representation code.

e. As the semantic checking is separated from the AST definition to Visitor implementations, any changes to semantics can be easily applied without disturbing the AST definitions.
f. Similarly any implementation changes that are to be performed, can be handled in visitor pattern implementation without disturbing parser implementations. This modularizes the code and improves the code reusability.

3. LLVM Generation

The implementation proceeds as follows.

a. After semantic check the LLVM IR is generated to each virtual function in the visitor design pattern implementation.
b. Apart from visitor implementation functions for each derived AST class, static variables for LLVMModule, LLVMContext, LLVMInstructionBuilder, List and Maps to store local variable names and named values respectively, name.
c. A global context w.r.t to present context is created with LLVMContext.

   public static final LLVMContext context = LLVMContext.getGlobalContext()
d. LLVMModule is the base object created in the global context, contains all the functions and global variables constructs.

   At the end LLVM IR code is retrieved from the module.

   public static final LLVMModule module=new LLVMModule("Compilers Proj",context);
e. LLVMInstructionBuilder appends all the instructions created as part of AST to LLVMIR translations.

   public static final LLVMInstructionBuilder builder = new LLVMInstructionBuilder();

   All these generated instructions in the scope of builder constructs module.
f. LLVMGlobalVariable creates register values.
g. LLVMFunction creates a function, LLVMFunctionType defines function properties of declared function.
   i. Return type
   ii. Argument types
   iii. Variable Arguments
h. Blocks are constructed with LLVMBasicBlock object are positioned in the builder. Each block is appended to the parent function.

   LLVMBasicBlock fblock = f.appendBasicBlock("entry");

   builder.positionBuilderAtEnd(fblock);
i. LLVMIntegerType, LLVMArrayType, LLVMVoidType, LLVMPointerType, LLVMConstantString, LLVMConstantInteger, LLVMConstantArray, used to create integer, array, void, pointer type variables, string, integer and array constants respectively.
j. LLVMType and LLVMConstant are the parents of LLVM types and constants. LLVMConstant is a derived type of LLVMValue.
k. LLVMAddInstruction, LLVMDivideInstruction, LLVMSubstractInstruction, LLVMMultiplyInstrunction, LLVMRemainderInstruction generates arithmetic instructions.

LLVMUnaryInstruction returns unary instruction. LLVMIntegerComparison creates an instruction to compare integer type operands.

l. LLVMLoadInstruction loads value from a location.
m. LLVMStackAllocation creates a memory location in stack taking type and name as parameters, returns pointer to location.
n. LLVMStoreInstrunction stores operand value at particular location taking value and pointer as argument.
o. LLVMGetElementPointerInstruction returns pointer which points to address of particular element of an array.
p. Other branching instructions like LLVMReturnInstruction is used in building function, creates a return statement, LLVMCallInstruction invokes a function either from within the module or external libraries.

4. Handling LLVM IR

a. LLVM IR can be dumped to a file or assigned to a string from LLVMModule. LLVMModule dumps bytecode.
b. LLVM disassembler (llvm-dis) converts byte code to LLVM IR source
```
llvm-dis -f decaf.out -o decaf.s
```
c. LLVM interpreter is used to compile and executes the LLVM IR
```
lli decaf.s
```
d. Clang can be used to generate LLVM IR for a C language source file, and can be used to compare with LLVM IR generated by our implementation.
```
clang –S –emit-llvm s.c
```

5. Issues faced

a. Understanding SSA value representation of elements in LLVM IR and LLVM API took considerable time
b. Applying semantics need more effort than implementing project as whole.

6. Future Work

a. Apply more semantics and thereby make compiler robust.
b. Error handling (both semantics and syntax) can be implemented in efficient way
c. Optimization of LLVM IR generation logic to generate optimized code.

Appendix

1. Syntax

```
program             :       CLASS ID '{' declarations '}'
declarations        :       type  fields ';' declarations
                    |       method_decl
fields              :       field
                    |       field ',' fields
field               :       ID
                    |       ID  '[' INT_LITERAL  ']'
method_decl         :       type ID  '(' args_decl ')' block method_decl
                    |       VOID ID  '(' args_decl ')' block method_decl
                    |
args_decl           :       arg ',' args_decl
                    |       arg
                    |
arg                 :       type ID
vars                :       ID ';'
                    |       ID ',' vars
var_decl            :       type vars
var_decls           :       var_decl var_decls
                    |       var_decl
block               :       '{' block_body '}'
statements          :       statement
                    |       statement statements
block_body          :       var_decls statements
                    |       var_decls
                    |       statements
                    |
type                :       INT
                    |       BOOLEAN

Statement           :       location ASSGN_OP expr ';'
                    |       method_call ';'
                    |       IF '(' expr  ')' block  ELSE block
                    |       IF '(' expr  ')' block
                    |       FOR ID  E_ASSIGN_OP expr ',' expr  block
                    |       RETURN ';'
                    |       RETURN expr ';'
                    |       BREAK ';'
                    |       CONTINUE ';'
                    |       block
```

```
ASSGN_OP            :       ASSIGN_OP
                    |        E_ASSIGN_OP
exprs               :       expr
                    |       expr ',' exprs
callout_args        :       callout_arg
                    |       callout_arg ',' callout_args ;

method_call         :       method_name '('   ')'
                    |        method_name '(' exprs ')'
                    |       CALLOUT '(' STRING_LITERAL ',' callout_args ')'
                    |       CALLOUT '(' STRING_LITERAL  ')'
method_name         :       ID
location            :       ID
                    |       ID  '[' expr  ']'
ARTH_OP             :       ARITH_OP
                    |       MINUS
expr                :       expr ARTH_OP term1
                    |       term1
term1               :       term1 REL_OP term2
                    |       term2
term2               :       term2 EQ_OP term3
                    |       term3
term3               :       term3 COND_OP term4
                    |       term4
term4               :       location
                    |       method_call
                    |       literal
                    |       MINUS term4
                    |       '!' term4
                    |       '(' expr  ')'
callout_arg         :       expr
                    |       STRING_LITERAL


bool_literal        :       TRUE
                    |       FALSE
literal             :       INT_LITERAL
                    |       bool_literal
```

2. Semantic Rules

- No identifier is declared twice in the same scope. This includes callout identifiers, which exist in the global scope.

- No identifier is used before it is declared.
- The program contains a definition for a method called main that has no parameters (note that since execution starts at method main, any methods defined after main will never be executed).
- The <int-literal> in an array declaration must be greater than 0.
- The number and types of arguments in a method call (non-callout) must be the same as the number and types of the formals, i.e., the signatures must be identical.
- If a method call is used as an expression, the method must return a result.
- String literals and array variables may not be used as arguments to non-callout methods. Note: a[5] is not an array variable, it is an array location
- A return statement must not have a return value unless it appears in the body of a method that is declared to return a value.
- The expression in a return statement must have the same type as the declared result type of the enclosing method definition.
- An id used as a location must name a declared local/global variable or formal parameter
- For all locations of the form <id> [<expr>]
    - <id> must be an array variable, and
    - the type of expr must be int
- The <expr> in an if or a while statement must have type boolean.
- The operands of <arith-op> is and <rel-op> must have type int.
- The operands of <eq-op> must have the same type, either int or boolean.
- The operands of <cond-op> and the operand of logical not (!) must have type boolean.
- The <location> and the <expr> in an assignment, <location> = <expr> and <location> = <expr>, must have the same type.
- The <location> and the <expr> in an incrementing/decrementing assignment, <location> += <expr> and <location> -= <expr>, must be of type int
- The initial <expr> and the ending <expr> of for must have type int.
- All break and continue statements must be contained within the body of a for or a while.

3. Class Hierarchy

```
DecafIntf(A)
        Decaf_Class
        DeclarationsIntf(A)
                FieldDeclIntf(A)
                        FieldDecl
                MethodDeclarations(A)
                        MethodDeclarations1
                        MethodDeclarations2

Var_Decl
```

- Fields(A)
  - Field
  - Fields1
- Arguement(A)
  - Arguement1
  - Arguement2
- Type
  - IntegerType
  - BooleanType
  - Void
- ExpressionIntf(A)
  - StatementIntf(A)
    - Statement1
    - ReturnStatemen1
    - ReturnStatement2
    - BreakStatement
    - ContinueStatement
    - IfStatement
    - ElseStatement
    - ForStatement
    - Block(A)
      - Block1
      - BLock2
      - Block3
      - Block4
    - MethodStatement1
    - MethodStatement2
    - MethodStatementCallOut1
    - MethodStatementCallOut2
  - callout_args(A)
    - callout_arg
    - callout_arg1
  - Location(A)
    - Location1
    - Location2
  - Expression1
  - Expression2
  - Expression3
  - Literal
    - IntegerLiteral
    - BooleanLiteral
    - StringLiteral

Other Supporting Classes:

- Class
- Identifier
- IF
- Else
- Return
- Break
- Continue
- For
- Callout

Note: (A) represents Abstract Class