CS325 A Compiler for *MiniC*
Report

u1835495

Department of Computer Science
University of Warwick
2020-21

Contents

1 Parser and AST

For the first part of the coursework a Recursive Descent Parser was created for the *Mini C* language with functional Lexer provided. In order to have a recursive descent parser, some modifications in the given grammar had to be made. First of all the precedence rules were added for the arithmetic expressions, having operators *,/,% in level one, operators +,- in level two, operators ¡=, ¡ in level three, etc. Then in order to get a recursive descent parser, left recursion was eliminated from the grammar. See Figure 1. After that FIRST and FOLLOW sets were computed, see Figure 2 and Figure 3.

For terminals and non-terminals nodes for Abstract Syntax Tree were created. Nodes - classes, extending **ASTnode** base class. Each node class contains one or more variables of another type node except for literal nodes **IntASTnode**, **FloatASTnode**, **BoolASTnode** and **IdentASTnode** which do not reference any other nodes. Such implementation lets us build an Abstract Syntax Tree where the root node is the entry of the program (**ProgramASTnode**) which holds vector of **external** function declarations, vector of global variables and vector of function nodes. Parsing begins when **parse()** method is called which returns an Abstract Syntax Tree of type ASTnode.

With every token, its value is being compared to the values of the FIRST set of the possible non-terminal. If the value is equal to the ones in the set, then that parsing 'direction' is chosen. For example, if **ParseStmt())** method is being called, as we can see from the grammar, possible non-terminals are: **stmt::=expr_stmt** | **block** | **if_stmt** | **while_stmt** | **return_stmt**. We then compare first token with the FIRST sets of each of the non-terminal. If current token is equal to **IF** token, then we will advance with **ParseIfStmt()**, if current token is equal to **LBR** token, we will then continue parsing block.

If, at some step, current token is not equal to any of the expected values, we return error. For example if we are parsing local variable declaration and current token is equal to **FLOAT_TOK** we look into next token and, if it is not an **IDENT** token, we return error saying that variable name is missing.

Parsing continues until it hits the bottom, when it reaches the literals, the nodes are being created and stored in the Abstract Syntax Tree and it is then returned. Once it is returned, it can be printed. Each node has its **to_string** method, where recursively at each node this method is called for its children nodes and the print is being constructed. In these methods the information of each node is stored in the string and returned.

2 Types, Scope and Code Generation

Second part of the coursework was intended for code generation using the created AST as well as type and scope checking of the compiling program. It was done using **llvm** compiler infrastructure.

Every **ASTnode** class and **FunctionAST** with **PrototypeAST** class have **codegen()** method which was used for generating an Intermediate Representation of each node. The code generation begins when the method **codegen()** is called for the **extern** function node if one is created, multiple calls for multiple definitions. Then if global variables are defined, the same call is being performed for their created AST nodes. The **codegen()** method is then being called for every defined function with a body.

In the function body code is generated for locally defined variables to whom memory access is created using llvm **alloca**. After that, the code for the rest of the body of a function is being generated. It contains any number of statements.

For If/Else statement **IEBuilder** blocks are created as well as conditional branch instruction, which selects the block to continue depending on the condition outcome. For While statement conditional branch before the while block is created which either continue after the block or enter the while block created by **IRBuilder**.

Arithmetic expressions are performed using in the **codegen()** method for **ExpressionASTnode**, where recursively code is generating for left and right hand side of the arithmetic operation. Then if both **Values** are returned we perform the arithmetic operation depending on the operator value of the node. **CreateBinOp** of **IRBuilder** class is used for arithmetic expressions (+,-,*,/, etc.) and unary negation. Other methods are used for operands comparison.

Types are being checked in the same **codegen()** method. If one value of two values is Integer value and another one is a Float, then the Integer is being converted to Float and operations on Floats is performed. If one value is Integer and another is Boolean, the error is returned. If Values are Boolean and operator is (+,-,*,/ etc.) then error is returned. Similarly other type checking is performed.

Scope checking is performed when code is being generated for local declarations in the function. Variable that is being declared is being checked. If it appears in the **NamedValues**[] or **GlobalNamedValues**[] maps that means it was either given to function as an argument or was already declared globally. Thus, the error is returned.

If no errors detected, the generated code is printed out into the **output.ll** file which can be built and executed using **clang++**.

```
program           -> extern_list globals_list decl_list
                     | globals_list decl_list
globals_list      -> global globals_list | eps
global            -> param ";"
extern_list       -> extern extern_list'
extern_list'      -> extern extern_list' | eps
extern            -> "extern" prototype ";"
prototype         -> type_spec IDENT "(" params ")"
decl_list         -> decl decl_list'
decl_list'        -> decl decl_list' | eps
decl              -> prototype block
type_spec         -> "void" | var_type
params            -> param_list | "void" | eps
param_list        -> param param_list'
param_list'       -> "," param param_list' | eps
param             -> var_type IDENT
var_type          -> "int" | "float" | "bool"
stmt              -> expr_stmt | block | if_stmt
                     | while_stmt | return_stmt
return_stmt       -> "return" return_stmt_tail
return_stmt_tail  -> ";" | expr ";"
if_stmt           -> "if" "(" expr ")" block else_stmt
else_stmt         -> "else" block | eps
while_stmt        -> "while" "(" expr ")" stmt
block             -> "{" local_decls stmt_list "}"
local_decls       -> local_decl local_decls | eps
local_decl        -> param ";"
stmt_list         -> stmt stmt_list | eps
expr_stmt         -> expr ";" | ";"
expr              -> IDENT "=" expr | rval
rval              -> rval1
rval1             -> rval2 rval1'
rval1'            -> "||" rval2 rval1' | eps
rval2             -> rval3 rval2'
rval2'            -> "&&" rval3 rval2' | eps
rval3             -> rval4 rval3'
rval3'            -> "==" rval4 rval3' | "!=" rval4 rval3' | eps
rval4             -> rval5 rval4'
rval4'            -> "<=" rval5 rval4' | "<" rval5 rval4'
                     | ">=" rval5 rval4' | ">" rval5 rval4' | eps
rval5             -> rval6 rval5'
rval5'            -> "+" rval6 rval5' | "-" rval6 rval5 | eps
rval6             -> rval7 rval6'
rval6'            -> "*" rval7 rval6' | "/" rval7 rval6'
                     | "%" rval7 rval6' | eps
rval7             -> "-" rval7 | "!" rval7 | "(" expr ")"
                     | IDENT | IDENT "(" args ")" | INT_LIT
                     | FLOAT_LIT | BOOL_LIT
args              -> arg_list | eps
arg_list          -> expr arg_list'
arg_list'         -> "," expr arg_list' | eps
```

Figure 1: Grammar

```
globals_list            eps, "int", "float", "bool"
extern_list'            eps, "extern"
extern                 "extern"
decl_list'                 eps, "void", "int", "float", "bool"
type_spec                  "void", "int", "float", "bool"
params                 "void", eps, "int", "float", "bool"
param_list'             ",", eps
var_type                   "int", "float", "bool"
return_stmt             "return"
return_stmt_tail           ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LI
T
if_stmt                    "if"
else_stmt                "else", eps
while_stmt               "while"
block                  "{"
local_decls             eps, "int", "float", "bool"
stmt_list                 eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                       BOOL_LIT, "if", "return", "{", "while"
expr_stmt                  ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT
expr                    IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT
rval1'                 "||", eps
rval2'                 "&&", eps
rval3'                 "==", "!=", eps
rval4'                 "<=", "<", ">=", ">", eps
rval5'                 "+", "-", eps
rval6'                 "*", "/", "%", eps
rval7                  "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
args                    eps, IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT
arg_list'                 ",", eps
extern_list             "extern"
param                  "int", "float", "bool"
rval6                  "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
program                    "extern", eps, "int", "float", "bool"
global                 "int", "float", "bool"
prototype                  "void", "int", "float", "bool"
param_list                 "int", "float", "bool"
local_decl                 "int", "float", "bool"
rval5                  "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
decl                    "void", "int", "float", "bool"
rval4                  "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
decl_list                  "void", "int", "float", "bool"
rval3                  "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
rval2                  "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
rval1                  "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
rval                    "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
arg_list                   IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT
stmt                    ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT,
                       "if", "return", "{", "while"
```

Figure 2: FIRST sets

```
program                   $
globals_list              "void", "int", "float", "bool"
global                  eps, "int", "float", "bool"
extern_list               eps, "int", "float", "bool"
extern_list'              eps, "int", "float", "bool"
extern                  eps, "extern"
prototype                   "(", ";"
decl_list                   $
decl_list'                  $
decl                      eps, "void", "int", "float", "bool"
type_spec                   IDENT
params                  ")"
param_list                  ")"
param_list'               ")"
param                   ";", ",", eps
var_type                    IDENT
stmt                      eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                        BOOL_LIT, "if", "return", "{", "while"
return_stmt               eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                        BOOL_LIT, "if", "return", "{", "while"
return_stmt_tail            eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                        BOOL_LIT, "if", "return", "{", "while"
if_stmt                     eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                        BOOL_LIT, "if", "return", "{", "while"
else_stmt                   eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                        BOOL_LIT, "if", "return", "{", "while"
while_stmt                  eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                        BOOL_LIT, "if", "return", "{", "while"
block                   "else", eps, ";", IDENT, "-", "!", "(", INT_LIT,
                        FLOAT_LIT, BOOL_LIT, "if", "return", "{", "while",
                        "void", "int", "float", "bool"
local_decls               eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
                        BOOL_LIT, "if", "return", "{", "while"
local_decl                  eps, "int", "float", "bool"
stmt_list                   "}"
expr_stmt                   eps, ";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL
_LIT, "if", "return", "{", "while"
expr                        ",", eps, ")", ";"
rval                        ",", eps, ")", ";"
rval1                     ",", eps, ")", ";"
rval1'                    ",", eps, ")", ";"
rval2                     "||", eps
rval2'                    "||", eps
rval3                     "&&", eps
rval3'                    "&&", eps
rval4                     "==", "!=", eps
rval4'                    "==", "!=", eps
rval5                     "<=", "<", ">=", ">", eps
rval5'                    "<=", "<", ">=", ">", eps
rval6                     "+", "-", eps, "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
rval6'                    "+", "-", eps, "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
rval7                     "*", "/", "%", eps
args                        ")"
arg_list                    ")"
arg_list'                 ")"
```

Figure 3: FOLLOW sets