**Compiler Design Coursework Report**

The current paper holds information about the decisions taken during the creation of the compiler for the Mini C programming language.

To keep things simple, I will discuss those using 3 mini chapters which represent the 3 steps used for implementing the compiler. I will not discuss how I have created the first and follow sets as they are already explained in the lectures. Those and the grammar are listed at the end of this documentation.

1. The creation of ASTnodes

The basic class for any node, as recommended by the task, is called *ASTnode* which contains a basic ASTnode constructor, the *codegen()* function and the *to\_string()* function. Other nodes extend this class by adding functionality and actual code.

class ASTnode

{

public:

virtual ~ASTnode() {}

virtual Value \*codegen() = 0;

virtual std::string to\_string() const {};

};

*ASTnode class definition*

Usually, all the nodes follow the structure of the grammar, with some exceptions where some productions have been restructured or not considered at all.

As a general rule, if in the grammar existed recursive productions, I have used an array, represented by the std::vector interface, to store all the information. This type of behavior can be seen at DeclarationList node, ExternList node, LocalDeclarations node, FunctionArgs node, Params node and others. It was easier to parse all the multiple individual nonterminal that create that production and structure them under an array.

class Params : public ASTnode

{

public:

std::vector<std::unique\_ptr<Param>> \_ParameterList\_;

std::string VoidTok;

*Declaration of an array that holds all the parameters in a function argument list*

To keep things organized, I preferred sometimes to create multiple redundant nodes, such as STMT node which holds only one value, the rest being nullptr. For me, it was easier to debug and write code. This node acts like an *if* that it checks if there is an expression, a block, or an if/while/return.

To add more info, the BinOp class has been implemented for the operations. It uses 2 pointers, one for the left part of the expression, one for the right part of the expressions, and an int which holds the token for an operation (this is because CurTok.lexeme was at some point bugged and it did not have an empty string in it, causing everything to break on code gen part).

1. The Top-Down parser

The parser is implemented in a simple manner, by following the follow and first sets. My grammar has an LL(0), LL(1), LL(2) mainly productions and only one with LL(3).

To make things easier, you can see a list with the prototypes of the functions after the node declaration.

In every function, there is a check if the current token is in the first set. If yes, then the parser consumes the token and does the same thing. When reaching at the end of the production, it is returned the corresponding node with instantiated values. If there is a parsing error, then it is displayed the error and the function returns nullptr. This nullptr is propagated towards the main function which stops from execution.

1. The Code Generation

In this part of the coursework, I have taken the approach of the C99 compiler, with some tweaks. One of the major decisions was the conversion. If a number is float and it is used into an int context, then the context is converted to the float. I followed the C99 conversions: float > int > bool. This means that if a bool is used in an int context, then the bool is converted to int type. Same operation is done for the int -> float or bool -> float value and to the operands within a certain context or BinOp.

For the if part I have not done any PHI nodes, I have just created the blocks and labels and then inserted the code withing the code. But, for the AND and OR operators I have used PHI noted to bind the values from multiple branches to simulate the lazy evaluators (if LHS is true and the operator is AND, then skip RHS eval. Swap conditions for OR). Although, I have decided not to make conversions at the return statement. For instance, if it is returned bool, but the function is float, an error will be displayed.

The variable scope is held by the *VariableScopes* variable. This is build using the *LinkedTable* class, which is inspired form the lecture notes. All that I had to do is to convert it from Java to C++. When a new block is created, then I create a new table and linked it to the previous one. When the current block ends, then the current table is destroyed and replaced by the previous one. I do not permit variable redeclaration, but I allow function redefinition.

For semantics, I check if the variable name exists, if the name function exists when calling, if the function was called with the proper no and type of params, different return type and impossible implicit conversions.

**Grammar with FIRST and FOLLOW sets:**

**First**(programprime) = [extern, int, float, bool, void, EOF]

programprime ::= program EOF

| EOF

**First**(program) = [extern, int, float, bool, void]

program ::= extern\_list decl\_list

| decl\_list

**First**(extern\_list) = [extern, EOF, int, float, bool, void]

**Follow**(extern\_list) = [EOF, int, float, bool, void]

extern\_list ::= extern extern\_list

| epsilon

**First**(extern) = [extern]

extern ::= "extern" type\_spec IDENT "(" params ")" ";"

**First**(decl\_list) = [void, int, float, bool, EOF]

Follow(decl\_list) = [EOF]

decl\_list ::= decl decl\_list

| epsilon

**First**(decl) = [void, int, float, bool]

**LL(3)**

decl ::= var\_decl => [int, float, bool]

| fun\_decl => [void, int, float, bool]

**First**(fun\_decl) = [void, int, float, bool]

fun\_decl ::= type\_spec IDENT "(" params ")" block

**First**(var\_decl) = [int, float, bool]

var\_decl ::= var\_type IDENT ";"

**First**(var\_type) = [int, float, bool]

var\_type ::= "int"

| "float"

| "bool"

**First**(type\_spec) = [void, int, float, bool]

type\_spec ::= "void"

| var\_type

**First**(params) = [int, float, bool, void, epsilon]

**Follow**(params) = [")"]

params ::= param\_list

| "void"

| epsilon

**First**(param\_list) = [int, float, bool]

param\_list ::= param param\_list\_prime

**First**(param\_list\_prime) = [",", epsilon]

**Follow**(param\_list\_prime) = [")"]

param\_list\_prime ::= "," param param\_list\_prime

| epsilon

**First**(param) = [int, float, bool]

param ::= var\_type IDENT

**First**(block) = ["{"]

block ::= "{" local\_decls stmt\_list "}"

**First**(local\_decls) = [int, float, bool, epsilon]

**Follow**(local\_decls) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, "{", "}", "while", "if", "return"]

local\_decls ::= local\_decl local\_decls

| epsilon

**First**(local\_decl) = [int, float, bool]

local\_decl ::= var\_type IDENT ";"

**First**(stmt\_list) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, ";", "{", "while", "if", "return"]

**Follow**(stmt\_list) = ["{"]

stmt\_list ::= stmt stmt\_list => [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, ";", "{", "while", "if", "return"]

| epsilon => ["{"]

**First**(stmt) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, ";", "{", if, while, return]

stmt ::= expr\_stmt => [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, ";"]

| block => ["{"]

| if\_stmt => [if]

| while\_stmt => [while]

| return\_stmt => [return]

**First**(expr\_stmt) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, ";"]

expr\_stmt ::= expr ";" => [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

| ";" => [";"]

**First**(while\_stmt) = ["while"]

while\_stmt ::= "while" "(" expr ")" stmt

**First**(if\_stmt) = ["if"]

if\_stmt ::= "if" "(" expr ")" block else\_stmt

**First**(else\_stmt) = ["else", -, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, "{", "}", "while", "if", "return"]

**Follow**(else\_stmt) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT, "{", "}", "while", "if", "return"]

else\_stmt ::= "else" block

| epsilon

**First**(return\_stmt) = ["return"]

**LL(2)**

return\_stmt ::= "return" ";"

| "return" expr ";"

**First**(expr) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

expr ::= IDENT "=" expr => To check: IDENT =

| orterm => To check: IDENT ( ! - INT\_LIT FLOAT\_LIT BOOL\_LIT

**First**(orterm) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

orterm ::= andterm orterm\_prime

**First**(orterm\_prime) = [||, epsilon]

**Follow**(orterm\_prime) = [")" ";" ","]

orterm\_prime ::= "||" andterm orterm\_prime

| epsilon

**First**(andterm) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

andterm ::= equiv andterm\_prime

**First**(term\_prime) =[&&, ")" ";" "," "||"]

**Follow**(andterm\_prime) = [")" ";" "," "||"]

andterm\_prime ::= "&&" equiv term\_prime

| epsilon

**First**(equiv) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

equiv ::= rel equiv\_prime

**First**(equiv\_prime) = [==, !=, ")" ";" "," "||" "&&"]

**Follow**(equiv\_prime) = [")" ";" "," "||" "&&"]

equiv\_prime ::= "==" rel equiv\_prime

| "!=" rel equiv\_prime

| epsilon

**First**(rel) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

rel ::= subexpr rel\_prime

**First**(rel\_prime) = [<=, <, >=, >, ")" ";" "," "||" "&&" "==" "!="]

**Follow**(rel\_prime) = [")" ";" "," "||" "&&" "==" "!="]

rel\_prime ::= "<=" subexpr rel\_prime

| "<" subexpr rel\_prime

| ">=" subexpr rel\_prime

| ">" subexpr rel\_prime

| epsilon

**First**(subexpr) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

subexpr ::= factor subexpr\_prime

**First**(subexpr\_prime) = [+, -, ")" ";" "," "||" "&&" "==" "!=" "<=" "<" ">=" ">"]

**Follow**(subexpr\_prime) = [")" ";" "," "||" "&&" "==" "!=" "<=" "<" ">=" ">"]

subexpr\_prime ::= "+" factor subexpr\_prime

| "-" factor subexpr\_prime

| epsilon

**First**(factor) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

factor ::= unary factor\_prime

**First**(factor\_prime) = [\*, /, %, ")" ";" "," "||" "&&" "==" "!=" "<=" "<" ">=" ">", "+", "-"]

**Follow**(factor\_prime) = [")" ";" "," "||" "&&" "==" "!=" "<=" "<" ">=" ">", "+", "-"]

factor\_prime ::= "\*" unary factor\_prime

| "/" unary factor\_prime

| "%" unary factor\_prime

| epsilon

**First**(unary) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

unary ::= "-" unary

| "!" unary

| elem

**First**(elem) = [(, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

**LL(2)**

elem ::= "(" expr ")"

| IDENT

| IDENT "(" args ")"

| INT\_LIT

| FLOAT\_LIT

| BOOL\_LIT

**First**(args) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT , ")"]

**Follow**(args) = [")"]

args ::= arg\_list

| epsilon

**First**(arg\_list) = [-, !, (, IDENT, INT\_LIT, FLOAT\_LIT, BOOL\_LIT]

arg\_list ::= expr arg\_list\_prime

**First**(arg\_list\_prime) = ["," , ")"]

**Follow**(arg\_list\_prime) = [")"]

arg\_list\_prime ::= "," expr arglist\_prime

| epsilon