Raumaan Ahmed 2034972

CS325 Report

Grammar

```
program -> extern_list decl_list
| decl_list

extern_list -> extern extern_list_prime

extern_list_prime -> extern extern_list_prime | epsilon

extern -> "extern" type_spec IDENT "(" params ")" ";"

decl_list -> decl decl_list_prime

decl_list_prime -> decl decl_list_prime | epsilon

decl -> var_decl
| fun_decl

var_decl -> var_type IDENT ";"

type_spec -> "void"
| var_type

var_type -> "int" | "float" | "bool"

fun_decl -> type_spec IDENT "(" params ")" block

params -> param_list
| "void" | epsilon

param_list -> param param_list_prime
param_list_prime -> "," param param_list_prime | epsilon

param -> var_type IDENT

block -> "{" local_decls stmt_list "}"

local_decls -> local_decls_prime

local_decls_prime -> local_decl local_decls_prime | epsilon

local_decl -> var_type IDENT ";"

stmt_list -> stmt_list_prime
```

```
stmt_list_prime -> stmt stmt_list_prime | epsilon

stmt -> expr_stmt
| block
| if_stmt
| while_stmt
| return_stmt

expr_stmt -> expr ";"
| ";"

while_stmt -> "while" "(" expr ")" stmt

if_stmt -> "if" "(" expr ")" block else_stmt

else_stmt -> "else" block
| epsilon

return_stmt -> "return" return_stmt_B

return_stmt_B -> ";"
| expr ";"

expr -> IDENT "=" expr
| or_val

or_val -> and_val or_val_prime

or_val_prime -> "||" and_val or_val_prime
| epsilon

and_val -> eq_val and_val_prime

and_val_prime -> "&&" eq_val and_val_prime
| epsilon

eq_val -> comp_val eq_val_prime

eq_val_prime -> "==" comp_val eq_val_prime
| "!=" comp_val eq_val_prime
| epsilon

comp_val -> add_val comp_val_prime

comp_val_prime -> "<=" add_val comp_val_prime
| "<" add_val comp_val_prime
| ">=" add_val comp_val_prime
```

```
| ">" add_val comp_val_prime
| epsilon

add_val -> mul_val add_val_prime

add_val_prime -> "+" mul_val add_val_prime
| "-" mul_val add_val_prime
| epsilon

mul_val -> unary mul_val_prime

mul_val_prime ->  "*" unary mul_val_prime
| "/" unary mul_val_prime
| "%" unary mul_val_prime
| epsilon

unary ->  "-" unary
| "!" unary
| identifiers

identifiers -> "(" expr ")"
| IDENT identifiers_B
| INT_LIT
| FLOAT_LIT
| BOOL_LIT

identifiers_B -> "(" args ")"
| epsilon

args -> arg_list
| epsilon

arg_list -> expr arg_list_prime

arg_list_prime -> "," expr arg_list_prime | epsilon
```

First and Follow Sets

Production	Nullable	First	Follow
add_val	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "&&" ")" "," ";" "<" "<=" "==" ">" ">=" "||"
add_val_prime	true	"+" "-" epsilon	"!=" "&&" ")" "," ";" "<" "<=" "==" ">" ">=" "||"

Production	Nullable	First	Follow
and_val	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")" "," ";" "||"
and_val_prime	true	"&&" epsilon	")" "," ";" "||"
arg_list	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")"
arg_list_prime	true	"," epsilon	")"
args	true	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT epsilon	")"
block	false	"{"	"!" "(" "-" ";" "bool" "else" "float" "if" "int" "return" "void" "while" "{" "}" $ BOOL_LIT FLOAT_LIT IDENT INT_LIT
comp_val	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "&&" ")" "," ";" "==" "||"
comp_val_prime	true	"<" "<=" ">" ">=" epsilon	"!=" "&&" ")" "," ";" "==" "||"
decl	false	"bool" "float" "int" "void"	"bool" "float" "int" "void" $
decl_list	false	"bool" "float" "int" "void"	$
decl_list_prime	true	"bool" "float" "int" "void" epsilon	$
else_stmt	true	"else" epsilon	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
eq_val	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"&&" ")" "," ";" "||"
eq_val_prime	true	"!=" "==" epsilon	"&&" ")" "," ";" "||"
expr	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")" "," ";"
expr_stmt	false	"!" "(" "-" ";" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT

Production	Nullable	First	Follow
extern	false	"extern"	"bool" "extern" "float" "int" "void"
extern_list	false	"extern"	"bool" "float" "int" "void"
extern_list_prime	true	"extern" epsilon	"bool" "float" "int" "void"
fun_decl	false	"bool" "float" "int" "void"	"bool" "float" "int" "void" $
identifiers	false	"(" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "%" "&&" ")" "*" "+" "," "-" "/" ";" "<" "<=" "==" ">" ">=" "||"
identifiers_B	true	"(" epsilon	"!=" "%" "&&" ")" "*" "+" "," "-" "/" ";" "<" "<=" "==" ">" ">=" "||"
if_stmt	false	"if"	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
local_decl	false	"bool" "float" "int"	"!" "(" "-" ";" "bool" "float" "if" "int" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
local_decls	true	"bool" "float" "int" epsilon	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
local_decls_prime	true	"bool" "float" "int" epsilon	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
mul_val	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "&&" ")" "+" "," "-" ";" "<" "<=" "==" ">" ">=" "||"
mul_val_prime	true	"%" "*" "/" epsilon	"!=" "&&" ")" "+" "," "-" ";" "<" "<=" "==" ">" ">=" "||"
or_val	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")" "," ";"
or_val_prime	true	"||" epsilon	")" "," ";"
param	false	"bool" "float" "int"	")" ","
param_list	false	"bool" "float" "int"	")"
param_list_prime	true	"," epsilon	")"

Production	Nullable	First	Follow
params	true	"bool" "float" "int" "void" epsilon	")"
program	false	"bool" "extern" "float" "int" "void"	$
return_stmt	false	"return"	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
return_stmt_B	false	"!" "(" "-" ";" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
stmt	false	"!" "(" "-" ";" "if" "return" "while" "{" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
stmt_list	true	"!" "(" "-" ";" "if" "return" "while" "{" BOOL_LIT FLOAT_LIT IDENT INT_LIT epsilon	"}"
stmt_list_prime	true	"!" "(" "-" ";" "if" "return" "while" "{" BOOL_LIT FLOAT_LIT IDENT INT_LIT epsilon	"}"
type_spec	false	"bool" "float" "int" "void"	IDENT
unary	false	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "%" "&&" ")" "*" "+" "," "-" "/" ";" "<" "<=" "==" ">" ">=" "||"
var_decl	false	"bool" "float" "int"	"bool" "float" "int" "void" $
var_type	false	"bool" "float" "int"	IDENT
while_stmt	false	"while"	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT

Parser and AST Nodes

I first constructed the parser without AST nodes to get the general structure of the parser which is achieved by mapping each production to a function. In each function, if there are multiple choices that can be taken, a switch statement with the first set of of that production would decide which next function to take. A current token variable was used to take a token from the token buffer and this variable is used to make decisions within switch statements in functions. For most productions, the first set of each production that can be chosen were disjoint(LL(1)) so picking the production was as simple as choosing the correct case in the

switch statement. However with the case of `decl` and `expr`, these were not LL(1), so more than 1 lookaheads were needed to determine which production to take and this was achieved through the use of peek functions that got the relevant token needed to make a decision. Ambiguity was present in the grammar, especially in the binary operator productions. A solution was to enforce precedence by splitting the productions into seperate productions with highest precendence at bottom and lowest at top.For some productions, they were nullable, which means if the current token matches the follow set of a production, it should going further with that production (the equivalent of returning nothing). The follow set was also included inside switch statements alongside first sets.

For functions that did have a first and follow set check, if the current token did not match these sets, a syntax error is thrown and the program exits. Furthermore, a match function was used to match the current token based off of the grammar rules, and if a match occurs, we get the next token otherwise a syntax error is thrown. Any syntax error encountered, results in an error message printed, the offending line of code, and then termination of the program. We exit as we already know there is an error so there is no need to check further, as the programmer should direct their attention to this error first. Some productions were left recursive(which can lead to infinite loops) so this had to be eliminated and the result was prime productions being made. Left factoring was used on some productions to eliminate any further backtracking.

AST node classes were created to serve as the nodes for the parse tree and only a select number of AST nodes are created rather than creating a node for each production, as nodes can be simply be passed along functions. Nodes may contain multiple children so a vector of nodes were used in some classes. A base AST node is created first which contains the `to_string` and `codegen` definitions which would need to be implemented in the nodes that derive this base class. Unique pointers were used throughout, where these pointers can be moved across functions using `std::move` and is disposed off as soon as the pointer is out of scope.

With the AST classes implemented, the nodes can be integrated into the parser functions. There was more difficulty with binary operations as there were prime functions for each operator and there was the problem with right associativity in operator prime functions. This was solved in operator (non prime) functions obtaining all expressions with operaters of higher precedence first before continuing with operators of lower precendence. Productions could also build vectors which would be the children for various nodes. We can stop the further building of the vector by making use of the follow set.

The AST tree was printed horizontally, making use of `to_string()` methods in each class and calling the root node which does a pre-order traversal to print the tree.

Codegen, Error checking and Lazy Evaluation

In each AST node, there is a codegen function, used to generate IR.

Various semantic checks were considered such as if the variable or functions exists, or if there was an implicit type conversion. This was done by checking every scope, which was implemented as a vector of maps which can be iterated. There is also a map of global variables that can be checked as well. On any semantic error, just like syntax errors, an error message is printed, as well as the offending line and the program exits. Some semantic issues such as implicit type casting, result in a warning. Unlike errors, warnings are placed on a queue and at the end of the code generation, the queue is printed with each message and

each offending line. Warnings will not exit the program as they are not pressing issues that violate syntax and semantics but they can still lead to undefined consequences so the programmer should be alerted. Another warnable fault would a return statement not being in non void function which the program checks by using a `return_found` variable which is `1` if a return is found, `0` otherwise. Various semantic errors were implemented due to time constraints and the complexity of the language such as division by 0 or checking if all control paths in a program have a return in a non void function. The printing of the errors and warnings was inspired the errors and warnings printed by the GCC compiler.

```
./tests/addition/addition.c:6:3 warning: implicit conversion from float
 to integer
  result = n + 4.0;
  ^
```

Fig.1 - Example warning

Lazy evaluation, was implemented as it cuts out evaluating the entire condition. The design was insipired by the if-then statements codegen implementation. The left hand side is evaluated first and if true, for or, set an alloca that stores the truth as true and return, and for and, evaluate the right hand side. On the right hand side, if true, if false for "or", then we jump past the block that sets the truth alloca true, and return the false alloca, the same follows for "and".

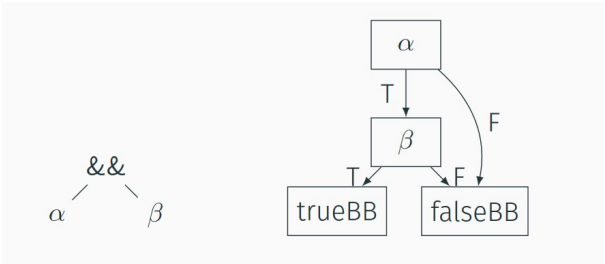

Fig.2 - lazy evalaution diagram for "&&"

Testing and miscellaneous

Testing was conducted using the provided C file test cases, which my parser passes on all of them, and the C file files were modified to test error checking. I also used more tests to test more niche aspects such as complicated unary expressions, boolean formulas and lazy evaluation. The bash file provided helped automate tests. LLVM IR errors helped with debugging errors from malformed IR.

The codebase was split into multiple files and header files to help greatly with readibility and made working with the codebase much easier rather than navigating a large single file.

References and Sources of information

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html (Tutorial on the lexer)(Last Acessed: 21/11/2022)

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html (Tutorial on Parser and AST nodes)(Last Acessed: 21/11/2022)

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html (Tutorial on LLVM functions, expressions, prototypes)(Last Acessed: 21/11/2022)

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl05.html (Tutorial on LLVM control flow)(Last Acessed: 21/11/2022)

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl07.html (Tutorial on LLVM mutable variables)(Last Acessed: 21/11/2022)

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/090 Top-Down Parsing.pdf (Used as inspiration for the Parser structure)(Last Acessed: 21/11/2022)(Maggie Johnson and Julie Zelenski)

https://warwick.ac.uk/fac/sci/dcs/people/u1607856/fighting_dragons_how_to_use_llvm.pdf (General tutorial on LLVM)(Last Acessed: 21/11/2022)(Finnbar Keating)

https://www.compilers.cs.uni-saarland.de/teaching/cc/2017/slides/llvm_intro.pdf (Used for lazy "and" diagram)(Last Acessed: 21/11/2022)(Fabian Ritter)

https://cyberzhg.github.io/toolbox/first_follow (Used to verify validity of first and follow sets for grammar)(Last Acessed: 21/11/2022)

https://llvm.org/doxygen/classllvm_1_1IRBuilder.html (Documentation for IR Builder functions)(Last Acessed: 21/11/2022)

https://llvm.org/doxygen/classllvm_1_1ConstantInt.html (Documentation for IR ConstantInt)(Last Acessed: 21/11/2022)

https://llvm.org/doxygen/classllvm_1_1GlobalVariable.html (Documentation for IR GlobalVariable)(Last Acessed: 21/11/2022)

https://llvm.org/doxygen/classllvm_1_1AllocaInst.html (Documentation for IR AllocaInst)(Last Acessed: 21/11/2022)

https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs325/cs325-3-syntaxanalysis.pdf (lecture on first and follow sets, left factoring, left recursion, precedence, grammars)(Last Acessed: 21/11/2022)(Gihan Mudalige)