CS325 Compiler Design Coursework Report

Part 1: Parser and AST

Grammar

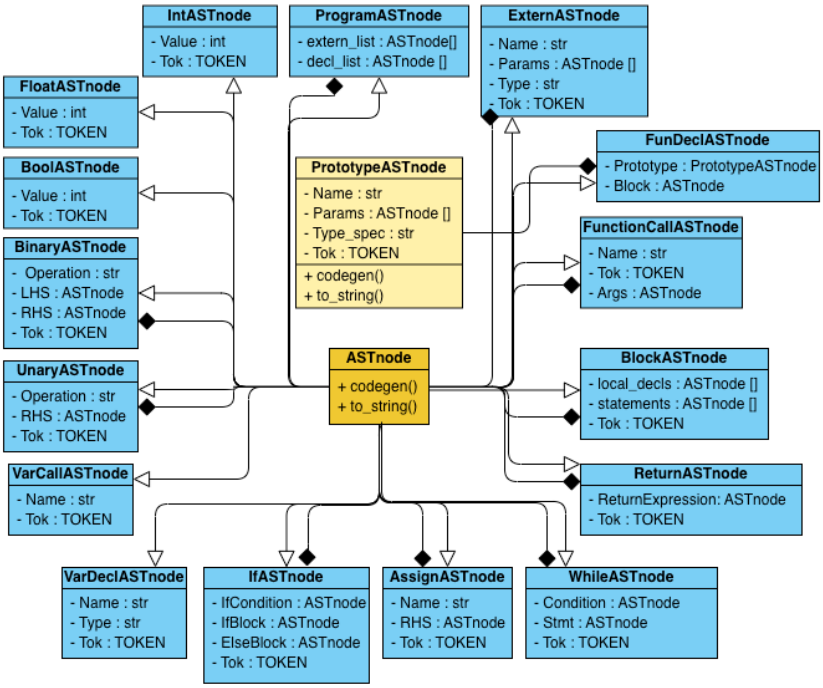
Overall, the grammar is LL(2). A lookahead of two is used in the rule: *decl ::= var_decl | fun_decl* to resolve the conflict between *var_decl* and *fun_decl,* which can start with *var_type IDENT ";"* and *var_type IDENT "("* respectively. Additionally, the uses a lookahead of one in the rule: *expr::= IDENT "=" expr | rval7* because *IDENT* is in FIRST(*rval7*). Otherwise, the grammar is LL(0). To get the grammar into this form, the left recursion was eliminated using the FIRST and FOLLOW sets at the end of this document. The initial *rval* rule was split into seven rules with increasing operator precedence.

Abstract Syntax Tree Nodes

There are 17 nodes in total with six main groups:

- Literals – integers, floating point numbers and bools
- Operations – unary and binary expressions
- Statements – if, while, return and assignment statements and blocks of statements
- Variable calls and variable declarations/parameters
- Function calls, function declarations (Prototype and block of code) and extern statements
- Root/ Program node

The function declaration node was chosen to be a combination of a prototype and a block of code to split the responsibilities of the node during code generation.



Syntax Errors and Printing the Parse Tree

Syntax errors are limited to throwing errors when missing or unexpected tokens are encountered in the parser. An error token variable is kept preserving the last token popped off the stack for error recovery and to give the line and column where the syntax error occurred. Similarly, the current token is stored in the AST nodes for error recovery during the semantic analysis. To print the parse tree, the to_string() method performs an in-order traversal of the generated tree and prints the attributes of each node.

Part 2: Types, Scope and Code Generation

Scope

A map of maps called VariableStack is used to store the variables and handle scope; the internal map storing the name of the variable and the Alloca and the external map stores the internal map at each level of the scope. The scope level is incremented when a new block of code, if statement, while statement or function declaration is encountered in the code generation and decremented before the value or function is returned.

Code Generation

- Literals (integer, float, bool) return the value with the correct type.
- Operations (unary and binary) first generate the code for the expressions and then apply the operation. For the binary Boolean expressions (|| and &&), lazy evaluation has been implemented.
- Variable calls check each level of the local and global variable stack and return the Alloca instance. Similarly, the assignment nodes find the variable and update the variable stack with the new value.
- Variable declarations check the local variable stack for variable redeclaration but allow global variables to be redeclared. Then the Alloca is created and added to the stack for local declarations, or the global variable is created.
- Function calls check the module for a function with the name, check that the number of arguments matches, generate the code for the arguments then finally create a call to the function.
- Function declarations checks if the module already has a function prototype for the given function and creates one. If not, it stores the arguments in the variable stack and generates the code for the block of code.
- Return statements create a return instruction for either a value or a void value.
- The control flow statements (if and while) evaluate the condition and branch conditionally or unconditionally to evaluate different basic blocks.

The code generation is also done in an in-order traversal of the abstract syntax tree from the program node.

Semantic Errors

Type checking is done in 3 places:

- When returning a value, it must match the return type of the function declaration
- When assigning a value to a variable, the types must be compatible
- In binary operations, both sides of the operation must have compatible types

Implicit and explicit type casting between floats and integers has been implemented in all 3 cases, and a warning is also given.

Limitations

There are two known limitations of the solution, it is possible for a function without a return statement to be parsed, but the code generation will fail because it is waiting for one and due to the structure of the nodes. Additionally, type checking when assigning global variables has not been implemented because LLVM global variable types are pointers, but the value type will not be a pointer, so the condition would always fail.

FIRST and FOLLOW Sets

Non-terminal	FIRST	FOLLOW
program	“extern”, “int”, “float”, “bool”, ”void”	eof
extern_list	“extern”	
externlist'	“extern”, epsilon	“int”, “float”, “bool”, “void”
extern	“extern”	
decl_list	“int”, “float”, “bool”, “void”	
decl_list'	“int”, “float”, “bool”	eof
decl	“int”, “float”, “bool”, “void”	
var_decl	“int”, “float”, “bool”	
type_spec	“int”, “float”, “bool”, “void”	
var_type	“int”, “float”, “bool”, “void”	
fun_decl	“int”, “float”, “bool”, “void”	
params	“int”, “float”, “bool”, “void”, epsilon	“)”
param_list	“int”, “float”, “bool”	
param_list'	“int”, “float”, “bool”, epsilon	“)”
param	“int”, “float”, “bool”	
block	“{“	
local_decls	“int”, “float”, “bool”, epsilon	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT,”-“, “!”, “(“, “;”,”{“, “if”, “while”, “else”, “return”, “}”
local_decl	“int”, “float”, “bool”	
stmt_list	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT,”-“, “!”, “(“, “;”,”{“, “if”, “while”, “else”, “return”, epsilon	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT,”-“, “!”, “(“, “;”,”{“, “if”, “while”, “else”, “return”, “}”
stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT,”-“, “!”, “(“, “;”,”{“, “if”, “while”, “else”, “return”	
expr_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT,”-“, “!”, “(“, “;”	
while_stmt	“while”	
if_stmt	“if”	
else_stmt	“else”, epsilon	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT,”-“, “!”, “(“, “;”,”{“, “if”, “while”, “else”, “return”, “}”
return_stmt	“return”	
expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
rval7	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
rval7'	“||”, epsilon	“;”, “)”

rval6	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
rval6'	“&&”, epsilon	“;”, “)”
rval5	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
rval5'	“==”, “!=”, epsilon	“;”, “)”
rval4	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
rval4'	“<=”, “<”, “>=”, “>”, epsilon	“;”, “)”
rval3	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
rval3'	“+”, “-“, epsilon	“;”, “)”
rval2	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
rval2'	“*”,”/”, “%”, epsilon	“;”, “)”
rval1	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
args	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“, epsilon	“)”
arg_list	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“	
arg_list'	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ”-“, “!”, “(“, epsilon	“)”

Grammar

start ::= program eof

program ::= extern_list decl_list
| decl_list

extern_list ::= extern extern_list'

extern_list' ::= extern extern_List'
| epsilon

extern ::= "extern" type_spec IDENT "(" params ")" ";"

decl_list ::= decl decl_list'

decl_list' ::= decl decl_list' | 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'
| param

param_list' ::= param "," param_list' | epsilon

param ::= var_type IDENT

block ::= "{" local_decls stmt_list "}"

local_decls ::= local_decl local_decls
| epsilon

local_decl ::= var_type IDENT ";"

stmt_list ::= stmt stmt_list
| 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" expr ";"

# operators in order of increasing precedence
expr ::= IDENT "=" expr
| rval7

rval7::= rval6 rval7'

rval7' ::= "||" rval6 rval7'
| epsilon

rval6 ::= rval5 rval6'

rval6' ::= "&&" rval5 rval6'
| epsilon

rval5 ::= rval4 rval5'

rval5' ::= "==" rval4 rval5'
| "!=" rval4 rval5'
| epsilon

rval4 ::= rval3 rval4'

rval4' ::= "<=" rval3 rval4'
| "<" rval3 rval4'
| ">=" rval3 rval4'
| ">" rval3 rval4'
| epsilon

rval3 ::= rval2 rval3'

rval3' ::= "+" rval2 rval3'
| "-" rval2 rval3'
| epsilon

rval2 ::= rval1 rval2'

rval2' ::= "*" rval1 rval2'
| "/" rval1 rval2'
| "%" rval1 rval2'
| epsilon
```

```
rval1 ::=  "-" rval1 | "!" rval1
| "(" expr ")"
| IDENT | IDENT "(" args ")"
| INT_LIT | FLOAT_LIT | BOOL_LIT

args ::= arg_list
|  epsilon

arg_list ::= expr "," arg_list' | expr

arg_list' ::= expr "," arg_list' | epsilon
```