CS325 Report - 1811083

1 The Parser

The parser was implemented as a recursive descent parser as per the specification. A lookahead of 2 was utilised to differentiate between a variable declaration and a function declaration as the first two non terminals match. In addition a lookahead of 1 was utilised when expanding an expression. As IDENT is in the first set of rval this required a lookahead. However an assign symbol could not be the second symbol in an rval as it is one of the productions of element. Therefore the lookahead of 1 searched for an assign. For error checking for the parser any unambiguous error such as a missing right parenthesis was output however for the ambiguous ones like for example an error in a binary expresion. a op b where op is an binary operation expects any of a dozen operands however this would make for a lengthy and hard to understand error code if all possible operands were output. Instead I compiled the erroneous c file with clang which instead says expected ';' assuming it is just a lone variable. In addition for a block in the language it must be local declaration(s) followed by statements therefore if a local declaration appears after a statement in the language, it will output an error saying expected statement.

Note grammar for binary expressions was taken from the lectures and then left-recursion was removed. [6]

2 Code generation

Implementation of global variables in the compiler was adapted from [1].

Generating IR for binary expressions requires type checking. This is because you need to ensure the operands are of the correct type for the instruction. To solve this problem I implemented the solution from the dragon book [2] where max(l,r) returns the maximum type in a hierarchy between l and r. And widen(v,t,w) generates type conversions if needed to widen the contents of v of type t into a value of type w, if t=w simple return v. Then all that is left is to get the maximum type of the two sides of the binary expression and then widen them both to that maximum type. To be noted is that division and modulus by 0 as stated in the language reference [3] is undefined behaviour therefore a semantic check was implemented which extracts the constant from the Value and checks if it is 0.

The code generation for unary expressions was handled recursively. If statements were handled almost exactly like the LLVM tutorial [4], except that it is possible for the if statement in miniC to not contain an else therefore implementing this edge case was required.

Return statements have two specific cases to handle, the first is an empty return and the second is return x; where x is some value. The first is trivial to implement however you can run into semantic errors in the second case. If

the return type differs from the function type in C an implicit cast will occur, however I thought this was unsafe behaviour as it is possible to derive incorrect outputs as a narrowing cast can result in lost information like a cast from a float to an int. As a result I have handled this like an error. A limitation of this approach is that LLVM provides some functionality to determine if a cast will be lossless, so it would be possible to implement implicit casts with more work.

The while statement is similar to the for statement defined in the LLVM tutorial [4] except instead of checking the value of the iterator a conditional branch is used.

3 AST

The Abstract Syntax Tree (AST) somewhat resembles the clang AST but I have taken liberties. Any terminals that are syntactic sugar such as parenthesis are removed, in addition any direct transitions from a non-terminal to another non-terminal in the grammar did not result in the first non-terminal being a parent node to the latter non-terminal as this would cause needless AST bloat. A special case was implemented for the if statement to highlight the differences between the then block and the else block.

4 Scope checking

Implemented scopes as a hierarchy of symbol tables[5] The symbol table was implemented as a doubly linked list of hash tables as it is very efficient provided the memory space is available. implementations like a unordered list are only suitable for small amounts of variables and binary search trees take log n time for n variables. 4 functions were required to be defined.

1. Enter scope - this adds a new symbol table to the head of the linked list this is the current scope
2. Exit scope - this removes the symbol table at the head of the linked list, this will kill any local variables defined in that scope
3. Call variable - this determines if a variable exists in any scope if it does return it. Note that I implemented global variables as a separate map outside of the symbol table hierarchy for simplicity,
4. Declare variable - this simply adds a variable to the current scope.

References

[1]Marco, A 2014, *LLVM, Initialize an integer global variable with value 0*,Available at: https://stackoverflow.com/questions/23328832/llvm-initialize-an-integer-global-variable-with-value-0 (Accessed 5 November 2020).

[2]Aho, A., Lam, M., Sethi, R. and Ullman, J., 2015. *Compilers*. India: Pearson India Education Services.

[3] 2020, *LLVM Language Reference Manual*, Available at: https://llvm.org/docs/LangRef.html (Accessed 2 November 2020).

[4]2020, *My First Language Frontend with LLVM Tutorial*, Available at: https://llvm.org/docs/tutorial/MyFirst (Accessed 5 November 2020)

[5] Mudalige, G, 2020, *CS325 - Compiler Design Intermediate Representations*, Available at: https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs325/cs325-6-intermediaterepresentation.pdf (Accessed 10 November 2020)

[6] Mudalige, G, 2020, *CS325 - Lecture Week 3*. Available at: https://warwick.ac.uk/fac/sci/dcs/teaching/material/compilerdesign-synchlect3.mp4 (Accessed 18 October 2020).

```
rval → term rval'
rval' → || term rval' | ϵ
term → equiv term'
term' → && equiv term' | ϵ
equiv → rel equiv'
equiv' → "==" rel equiv' | "!=" rel equiv' | ϵ
rel → subexpr rel'
rel' → "<=" subexpr rel'
       | "<" subexpr rel'
       | ">=" subexpr rel'
       | ">" subexpr rel'
       | ϵ
subexpr → factor subexpr'
subexpr' → "+" factor subexpr' | "-" factor subexpr' | ϵ
factor → element factor'
factor' → "*" element factor'
       | "/" element factor'
       | "%" element factor'
       | ϵ
element → "-" element
       | "!" element
       | "(" expr ")"
       | IDENT
       | IDENT "(" args ")"
       | INT_LIT
       | FLOAT_LIT
       | BOOL_LIT

program → extern_list decl_list | decl_list
extern_list → extern extern_list'
extern_list' → extern extern_list' | ϵ
extern → "extern" type_spec IDENT "(" params ")" ";"
decl_list → decl decl_list'
decl_list' → decl decl_list' | ϵ
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" | ϵ
param_list → param param_list'
param_list' → "," param param_list' | ϵ
param → var_type IDENT
block → "{" local_decls stmt_list "}"
local_decls → local_decls'
local_decls' → local_decl local_decls' | ϵ
```

```
local_decl → var_type IDENT ";"
stmt_list → stmt_list'
stmt_list' → stmt stmt_list' | ϵ
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 | ϵ
return_stmt → "return" ";" | "return" expr ";"
expr → IDENT "=" expr | rval
args → arg_list | ϵ
arg_list → expr arg_list'
arg_list' → "," expr arg_list' | ϵ
```

5 First and follow sets

FIRST(prorgram) = {extern, "void", int, "float", "bool"}
FIRST(extern_list) = {extern}
FIRST(extern_list') = {ϵ, extern}
FIRST(extern) = {extern}
FIRST(decl_list) = {"void", "int", "float", "bool"}
FIRST(decl_list') = {ϵ, "void", "int", "float", "bool"}
FIRST(decl) = {"void", "int", "float", "bool"}
FIRST(var_decl) = {"int", "float", "bool"}
FIRST(type_spec) = {"void", "int", "float", "bool"}
FIRST(var_type) = {"int", "float", "bool"}
FIRST(fun_decl) = {"void", "int", "float", "bool"}
FIRST(params) = {"void", "int", "float", "bool", ϵ}
FIRST(params_list) = {"int", "float", "bool"}
FIRST(params_list') = {",", ϵ}
FIRST(param) = {"int", "float", "bool"}
FIRST(block) = {"{"}
FIRST(local_decls) = {ϵ, "int", "float", "bool"}
FIRST(local_decls') = {ϵ, "int", "float", "bool"}
FIRST(local_decl) = {"int", "float", "bool"}
FIRST(stmt_list) = {ϵ, "{", ";", "if", "while", "return", IDENT, -, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT}
FIRST(stmt_list') = {ϵ, "{", ";", "if", "while", "return", IDENT, "-", "!", "(", INT_LIT , FLOAT_LIT , BOOL_LIT}
FIRST(stmt) = {"{", ";", "if", "while", "return", IDENT, "-", "!", "(", INT_LIT , FLOAT_LIT , BOOL_LIT}
FIRST(expr_stmt) = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT }
FIRST(while_stmt) = {"while"}

FIRST(if_stmt) = {"if"}
FIRST(else_stmt) = {"else", ϵ}
FIRST(return_stmt) = {"return"}
FIRST(expr) = {IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT
}
FIRST(args) = {ϵ, IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
BOOL_LIT}
FIRST(arg_list) = {IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT,
BOOL_LIT }
FIRST(arg_list') = {",", ϵ}
FIRST(rval) = {"-","!","(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT
}
FIRST(rval') = {"||", ϵ}
FIRST(term) = {"-","!","(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(term') = {"&&", ϵ}
FIRST(equiv) = {"-","!","(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(equiv') = {"==", "!=", ϵ}
FIRST(rel) = {"-","!","(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rel') = {"$\leq$", "<", "$\geq$", ">", ϵ}
FIRST(subexpr) = {"-","!","(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(subexpr') = {"+", "-", ϵ}
FIRST(factor) = {"-","!","(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(factor') = {"*", "/", "%", ϵ}
FIRST(element) = {"-","!","(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}

FOLLOW(prorgram) = {EOF}
FOLLOW(extern_list) = {"void", "int", "float", "bool"}
FOLLOW(extern_list') = {"void", "int", "float", "bool"}
FOLLOW(extern) = {"extern", "void", "int", "float", "bool"}
FOLLOW(decl_list) = {EOF}
FOLLOW(decl_list') = {EOF}
FOLLOW(decl) = {"void", "int", "float", "bool", EOF}
FOLLOW(var_decl) = {"void", "int", "float", "bool" EOF}
FOLLOW(type_spec) = {IDENT}
FOLLOW(var_type) = {IDENT}
FOLLOW(fun_decl) = {"void", "int", "float", "bool", EOF}
FOLLOW(params) = {")"}
FOLLOW(params_list) = {")"}
FOLLOW(params_list') = {")"}
FOLLOW(param) = {",", ")"}
FOLLOW(block) = {"void", "int", "float", "bool", EOF, "{",
";", "if", "while", "return", IDENT, -, !, '(', INT_LIT , FLOAT_LIT
, BOOL_LIT, "}", "else"}
FOLLOW(local_decls) = {"{", ";", "if", "while", "return", IDENT,
-, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(local_decls') = {"{", ";", "if", "while", "return", IDENT,

-, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(local_decl) = {int", "float", "bool", "{", ";", "if",
"while", "return", IDENT, -, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT,
"}"}
FOLLOW(stmt_list) = {"}"}
FOLLOW(stmt_list') = {"}"}
FOLLOW(stmt) = {"{", ";", "if", "while", "return", IDENT, -,
!, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(expr_stmt) = {"{", ";", "if", "while", "return", IDENT,
-, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(while_stmt) = {"{", ";", "if", "while", "return", IDENT,
-, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(if_stmt) = {"{", ";", "if", "while", "return", IDENT,
-, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(else_stmt) = {"{", ";", "if", "while", "return", IDENT,
-, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(return_stmt) = {"{", ";", "if", "while", "return", IDENT,
-, !, '(', INT_LIT , FLOAT_LIT , BOOL_LIT, "}"}
FOLLOW(expr) = {";", ")", ","}
FOLLOW(args) = {")"}
FOLLOW(arg_list) = {")"}
FOLLOW(arg_list') = {")"}
FOLLOW(rval) = {";", ")", ","}
FOLLOW(rval') = {";", ")", ","}
FOLLOW(term) = {"||", ";", ")", ","}
FOLLOW(term') = {"||", ";", ")", ","}
FOLLOW(equiv) = {"&&", "||", ";", ")", ","}
FOLLOW(equiv') = {"&&", "||", ";", ")", ","}
FOLLOW(rel) = {"==", "!=", "&&", "||", ";", ")", ","}
FOLLOW(rel') = {"==", "!=", "&&", "||", ";", ")", ","}
FOLLOW(subexpr) = {"≤", "<", "≥", ">", "==", "!=", "&&",
"||", ";", ")", ","}
FOLLOW(subexpr') = {"≤", "<", "≥", ">", "==", "!=", "&&",
"||", ";", ")", ","}
FOLLOW(factor) = {"+", "-", "≤", "<", "≥", ">", "==", "!=",
"&&", "||", ";", ")", ","}
FOLLOW(factor') = {"+", "-", "≤", "<", "≥", ">", "==", "!=",
"&&", "||", ";", ")", ","}
FOLLOW(element) = {"*", "/", "%", "+", "-", "≤", "<", "≥",
">", "==", "!=", "&&", "||", ";", ")", ","}