CS325 Compiler Design - Coursework Report

ID: 2158063

Design and implementation

This compiler consists of three main parts: the lexer, the parser and the intermediate code generator. The lexer reads the characters of the input code, groups them into lexemes, and produces a stream of tokens as output, which is sent to the parser [1]. This compiler is designed to perform lexical analysis first, where it checks whether a token can be created from an input character, printing a lexical error if an invalid symbol is found. Once this is finished, syntax analysis can begin.

Parser

The parser calls one token at a time from the lexer and checks if it conforms to the grammar. While it does this, it also generates a parse tree, which is later used for IR code generation. The parser implemented in this compiler is a top-down predictive, recursive-descent parser. For this implementation, the grammar needs to be transformed into LL(k) form and must be eliminated from any left recursion. Also, appropriate operator precedence, based on the C language [2], must be applied to make sure that operations in an expression are performed in the correct order. The transformed grammar is shown in the Appendix section.

To make the parser predictive, the FIRST and FOLLOW sets must be computed [1]. The FIRST set is used to determine which correct one production rule to choose out of multiple production rules, without making a wrong move that will cause backtracking. The FOLLOW set is used to determine whether an epsilon production rule is valid. The FIRST sets of all production rules are created, and these are used to compute the FOLLOW sets of productions rules that have a epsilon production, as seen in the Appendix.

The token buffer stores the token generated by the lexer, for the parser to pop out and use. It is set to store at most two tokens because the production rule *exprStart* requires a lookahead of two token to make its decision, whereas the other production rules only need a lookahead of one. Since the grammar has already been transformed to LL(k) form, the FIRST and FOLLOW set of *exprStart* is defined so that it can look-ahead two tokens.

The predictive parser is composed of a series of functions of Bool return type, each representing the non-terminal production rule. In a recursive manner, each function will call other non-terminal functions, with each function returning true once the current token matches with a terminal or epsilon. A chain of true Boolean values being returned to the starting function *parser* indicates that syntax analysis (parsing) is successful. If a function returns a value of false, this will cascade back as a sequence of false values to the starting production rule *parser*, indicating a failed parse and a syntax error.

AST node generation

In this compiler, it was decided that the functions representing each production rule do not have a AST node return type, like seen in the Kaleidoscope implementation [3]. The Boolean return types are kept for each function, but they will still collect the relevant names and types required to create each type of AST node and store them in public vector data stores and variables. Helper functions are also implemented to make use of this public data to create the necessary ASTnodes, such as *processStmt()* for creating AST nodes for control flow statements, and *createExprASTnode()* which is used to parse and create AST nodes for expressions. Calls to these helper functions are made in the correct place inside the production rule functions, to create the AST in the correct order while parsing.

The parser needs to generate a parse tree as an output. This tree should consist of AST nodes, that all derive from a root node. This implementation makes use of two base AST node classes: *ASTnode* and *TopLevelASTnode*. *TopLevelASTnodes* are examples of function declarations, global variables, and prototypes, that sit at the outer-most scope of the program. These consist of *ASTnodes* as children, which could be variable declarations, control flow statements etc. This means that the root node will be implemented as a vector of *TopLevelASTnodes*, since the AST is not binary, and the root node can have many children. When traversing the AST for printing and IR generation, each element of the vector, a *TopLevelASTnode* will be visited and its *to_string()* and *codegen()* functions will be called, which in turn, calls the *to_string()* and *codegen()* functions of all the child *ASTnodes* recursively.

Semantic checks during IR code generation

As part of semantic analysis, several semantic checks were implemented during the IR generation to make sure the input code follows the semantic rules of MiniC.

Firstly, as part of the MiniC special semantic rules [4], the compiler ensures that global variables are declared only once. It maintains this by checking the global symbol table for an existing variable with the same name. Global variables can be re-declared inside local scopes, such as in function declarations of control flow blocks, which is done by adding the new variable declarations into the local symbol table, essentially becoming a local variable.

To maintain scope of variables, a vector of symbol tables is created to store variables at each scope level, where the oldest (first) table represents the outmost scope variables whilst the most recent (last) symbol table represents the innermost scope variables. When a new block is created, a new symbol table is pushed back to the end of the vector. Once a block of code has been executed, the symbol table associated with that block will be discarded, by popping the symbol table at the end of the vector, and the variables declared within that block will no longer be accessible.

Another MiniC special semantic rule [4] specifies that the correct number of arguments must be passed into a function call. For a function that takes no arguments, its function call should not be passed with more than zero arguments, otherwise an error will be produced. The correct, exact number of arguments must be passed into a function declaration. This is done by retrieving the callee function and counting the number of parameters it has, before comparing it to the number of arguments that have been passed in.

In C operator arithmetic, undefined behaviour is produced when the second operand of a divide (/) or remainder (%) operation is zero [5]. To avoid this, a semantic check was implemented in this compiler to detect if the second operand of these two operations is equal to zero, and printing an error if a zero is found. Additionally, in C99, usual arithmetic conversions [6] are made for two operands that are involved in an arithmetic operation and do not have the same type. This is implemented in this compiler by allowing widening conversions to one of the operands such that both operands are of the same type before the operation can be performed, guaranteeing accuracy and correctness.

One major MiniC semantic rule that has been included in this compiler is the avoidance of narrowing conversions; the specification states that the compiler should flag narrowing conversions as errors and allowing widening conversions [4]. Despite being ambiguous, in the sense that MiniC should follow the semantics of C99 which allows narrowing conversions [7], the decision was made to avoid all narrowing conversions. This means that parameter passing, return statements, logical operators and assignments do not allow narrowing conversions, to avoid undefined behaviour caused by values going out of range. Incorrect types can still be passed to assignments, parameters or return statements and if possible, they can be widened to the correct type, otherwise a semantic error will be shown.

Limitations of compiler

Due to an oversight, the compiler fails to detect blocks (set of curly braces not associated with any function declarations or control flow statements) that is present in another block. In the current implementation, the compiler ignores the braces and adds the variables inside to the most recent symbol table. However, this block should represent a new scope and thus a new symbol table should be created to store its contained variables.

Additionally, the compiler can only perform lazy evaluation of Boolean expressions if left operand is a Bool value (false or true). If the left operand is an expression, it may not be possible to determine whether it gives a true or false value with the current implementation used for Boolean short-circuit code generation.

References

[1] Aho, A.V., Compilers: principles, techniques, and tools., chapters 3.1, 2.4.2, Pearson Education, 2013

[2] C Operator Precedence, https://en.cppreference.com/w/c/language/operator_precedence (Accessed 20/11/2023)

[3] Kaleidoscope: Implementing a Parser and AST, https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html (Accessed 20/11/2023)

[4] CS325 2023/24 Assessed Coursework, A Compiler for MiniC https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs325/courseworkcs325.pdf (Accessed 20/11/2023)

[5] C Arithmetic operators, https://en.cppreference.com/w/c/language/operator_arithmetic (Accessed 20/11/2023)

[6] C99 Language Standard: 6.3.1.8 Usual arithmetic conversions https://rgambord.github.io/c99-doc/sections/6/3/1/8/index.html (Accessed 20/11/2023)

[7] C99 Language Standard: 6.3.1.4 Real floating and integer https://rgambord.github.io/c99-doc/sections/6/3/1/4/index.html (Accessed 20/11/2023)

Appendix

Final LL(k) Grammar:

```
programStart ::= program $

program ::= extern_list decl_list
          | decl_list

extern_list ::= extern extern_list'

extern_list' ::= extern_list | epsilon

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

type_spec ::= "void"
            |  var_type

decl_list ::= decl decl_list'

decl_list' ::= decl_list | epsilon

var_type  ::= "int" |  "float" |  "bool"

decl ::= var_type IDENT decl' | "void" IDENT "(" params ")" block

decl' ::= ";" | "(" params ")" block

params ::= param_list
         |  "void" | epsilon

param_list ::= param param_list'

param_list' ::= "," 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_stmt'

return_stmt' ::= ";" | expr ";"
```

```
exprStart ::= IDENT "=" exprStart | epsilon

expr ::= exprStart rval_eight

rval_eight ::= rval_seven rval_eight'

rval_eight' ::= "||" rval_seven rval_eight'
        | epsilon

rval_seven ::= rval_six rval_seven'

rval_seven' ::= "&&" rval_six rval_seven'
        | epsilon

rval_six ::= rval_five rval_six'

rval_six' ::= "==" rval_five rval_six'
        | "!=" rval_five rval_six'
        | epsilon

rval_five ::= rval_four rval_five'

rval_five' ::= "<=" rval_four rval_five'
        | "<" rval_four rval_five'
        | ">=" rval_four rval_five'
        | ">" rval_four rval_five'
        | epsilon

rval_four ::= rval_three rval_four'

rval_four' ::= "+" rval_three rval_four'
        | "-" rval_three rval_four'
        | epsilon

rval_three ::= rval_two rval_three'

rval_three' ::= "*" rval_two rval_three'
        | "/" rval_two rval_three'
        | "%" rval_two rval_three'
        | epsilon

rval_two ::= "-" rval_two
        | "!" rval_two
        | rval_one

rval_one ::= "(" expr ")"
        | IDENT rval
        | INT_LIT | FLOAT_LIT | BOOL_LIT

rval ::= "(" args ")" | epsilon

args ::= arg_list
        |  epsilon

arg_list ::= expr arg_list'

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

Computed FIRST sets:

```
FIRST(programStart) = {"extern", "void", "int", "float", "bool"}
FIRST(program) = {"extern", "void", "int", "float", "bool"}
FIRST(extern_list) = {"extern"}
FIRST(extern_list') = {"extern"}
FIRST(extern) = {"extern"}
FIRST(type_spec) = {"void", "int", "float", "bool"}
FIRST(decl_list) = {"void", "int", "float", "bool"}
FIRST(decl_list') = {"void", "int", "float", "bool", epsilon}
FIRST(decl) = {"void", "int", "float", "bool"}
FIRST(decl') = {";", "("}
FIRST(var_type) = {"int", "float", "bool"}
FIRST(params) = {"int", "float", "bool", "void", epsilon}
FIRST(param_list) = {"int", "float", "bool"}
FIRST(param_list') = {",", epsilon}
FIRST(param) = {"int", "float", "bool"}
FIRST(block) = {"{"}
FIRST(local_decls) = {"int", "float", "bool", epsilon}
FIRST(local_decl) = {"int", "float", "bool"}
FIRST(stmt_list) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ";", "{",
"if", "while", "return", epsilon}
FIRST(stmt) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ";", "{", "if",
"while", "return"}
FIRST(expr_stmt) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ";"}
FIRST(while_stmt) = {"while"}
FIRST(if_stmt) = {"if"}
FIRST(else_stmt) = {"else", epsilon}
FIRST(return_stmt) = {"return"}
FIRST(return_stmt') = {";", "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(exprStart) = {IDENT, epsilon}
FIRST(expr) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval_eight) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval_eight') = {"||", epsilon}
FIRST(rval_seven) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval_seven') = {"&&", epsilon}
FIRST(rval_six) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval_six') = {"==", "!=", epsilon}
FIRST(rval_five) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval_five') = {"<=", "<", ">=", ">", epsilon}
FIRST(rval_four) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval_four') = {"+", "-", epsilon}
FIRST(rval_three) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval_three') = {"*", "/", "%", epsilon}
FIRST(rval_two) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
```

```
FIRST(rval_one) = {"(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(rval) = {"(", epsilon}
FIRST(args) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, epsilon}
FIRST(arg_list) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FIRST(arg_list') = {",", epsilon}
```

Computed FOLLOW sets:

```
FOLLOW(extern_list') = {"void", "int", "float", "bool"}
FOLLOW(decl_list') = {$}
FOLLOW(params) = {")"}
FOLLOW(param_list') = {")"}
FOLLOW(local_decls) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ";",
"{", "if", "while", "return", "}"}
FOLLOW(stmt_list) = {"}"}
FOLLOW(else_stmt) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ";", "{",
"if", "while", "return", "}"}
FOLLOW(exprStart) = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
FOLLOW(rval_eight') = {";", ")", ","}
FOLLOW(rval_seven') = {"||", ";", ")", ","}
FOLLOW(rval_six') = {"&&", "||", ";", ")", ","}
FOLLOW(rval_five') = {"==", "!=", "&&", "||", ";", ")", ","}
FOLLOW(rval_four') = {"<=", "<", ">=", ">", "==", "!=", "&&", "||", ";", ")", ","}
FOLLOW(rval_three') = {"+", "-", "<=", "<", ">=", ">", "==", "!=", "&&", "||", ";",
")", ","}
FOLLOW(rval) = {"*", "/", "%", "+", "-", "<=", "<", ">=", ">", "==", "!=", "&&",
"||", ";", ")", ","}
FOLLOW(args) = {")"}
FOLLOW(arg_list') = {")"}
```