CS325 - Compiler Design

Coursework report

Felix Bowyer 2064363
Department of Computer Science
University of Warwick

THE UNIVERSITY OF WARWICK

1 Parsing

The first decision made in creating the parser was deciding which nodes to use in the abstract syntax tree (AST). The aim when choosing which AST nodes to implement was to minimise the complexity, and write the parser such as to only include information that will be useful later for code generation. With this in mind, the following nodes were chosen:

- `ProgramASTnode` - to act as the root of the tree;
- `{Int,Bool,Float}ASTnode` - for literals of their respective types;
- `{Una,Bin}ExprASTnode` - associating opertaions with operands;
- `{Glob}VarDeclASTnode, VarExprASTnode and VarRefASTnode` - for variable declarations eg. `int x`, for assigning values to variables eg. `x=5`, and for referring to variables eg. `x`;
- `ParamASTnode, ProtoASTnode, FuncASTnode and CallASTnode` - for individual function parameters, function prototypes (signatures), function (body) definitions, and function calls;
- `ReturnASTnode` - for statements returning values from functions;
- `ExternASTnode` - for external function prototypes, derived from `extern` statements;
- `BlockASTnode` - for sections of code creating a new scope, derived from sections of code enclosed in { };
- `BranchASTnode and LoopASTnode` - to represent their respective control flow structures;

Before consideration, the grammar was augmented to make it more suitable for parsing. Precedence was added for operators, left recursion was removed, and left factoring was done where appropriate.

The grammar was used as a starting point for deciding on AST nodes. From there, 2 things were removed: single referencial nodes (those that serve only as a link between 2 other nodes, and so contribute no additional data) and list nodes (who were replaced by using a deque structure within other nodes).

The parser was implemented as a recursive descent LL(2) parser. Variable declarations and assignments were kept separate because MiniC does not allow variables to be assigned to before they are declared. Keeping this separation in the parser allows for separated operations during codegen.

ExternASTnode and ProtoASTnode were kept separate, despite ending up calling the same codegen methods. Whilst not nessecary, this was done to help keep the AST understandable and human readable.
Global and local variables were kept separate in parsing to facilitate easier separation of local and global variables later.

During parsing, particular attention was paid to preserving left associativity of binary operations. This was achieved by passing the left-hand-side of the operation through to the parsing function of the right-hand-side as a pointer, and only fully combining them once a leaf is reached in the parse tree.

Parse errors are shown to the user with an arrow annotating the location where it took place. In many cases, hints are shown to the user suggesting what may need to be changed to fix the problem.

2 Code generation

A large part of code generation is the type checking aspect. LLVM IR generation requires that any generated operation with 2 operands has the same type for both. Therefore, with every generation (for binary operations, return statements, parameter passing, etc), the type of the operand was checked against the expected type of the result, and converted if needed. If a conversion is necessary, a warning is shown to the user by the compiler.

In implementation, semantic errors were handled on a case-by-case basis during code generation. When type casting, a semantic error is raised if an unknown type is given. Semantic errors are also raised in relation to variable scope - if a variable is declared twice within the same scope, a suitable error is shown. If a variable is declared in an outer scope and redeclared in an inner scope, it is instead redeclared in the local scope only. This is achieved by maintaining a `varsFromThisScope` vector for each scope. On the instantiation of a variable, it is added to this vector; on the creation of a new scope, it is cleared, and on the closure of a scope, values are only copied back to their parent scope if they were not redeclared within the scope. This helps to preserve scope appropriately.

To provide a common mechanism for semantic and parse errors, a number of common methods are used, including `errorLocation`, which displays where the error took place in a useful way.

Where the need to show a warning or semantic error was ambiguous, the C99 standard was used as a basis.

3 Known limitations

There are a few edge cases known to fail using the compiler.

- There is an issue with statements of the form `a = b = <value>`, which under the MiniC grammar is valid. This fails regardless of the types of a, b or the value.
- An error was found to do with printing the AST for function calls with 0 arguments towards the end of the project. A provisional fix has been applied but may not be fully functional.
- There was not an attempt to implement lazy evaluation for operators && and ||.

4 Appendix

4.1 Final grammar

```
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_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_stmt'
return_stmt' ::= expr ";"
          | ";"
expr ::= IDENT "=" expr
          | rval
rval ::= rvalb rval'
rval' ::= "||" rvalb rval'
          | epsilon
rvalb ::= rvalc rvalb'
rvalb' ::= "&&" rvalc rvalb'
          | epsilon
rvalc ::= rvald rvalc'
rvalc' ::= "==" rvald rvalc' | "!=" rvald rvalc'
          | epsilon
rvald ::= rvale rvald'
rvald' ::= <=" rvale rvald' | "<" rvale rvald'
          | ">=" rvale rvald' | ">" rvale rvald'
          | epsilon
rvale ::= rvalf rvale'
rvale' ::= "+" rvalf rvale' | "-" rvalf rvale'
          | epsilon
rvalf ::= rvalg rvalf'
rvalf' ::= "*" rvalg rvalf' | "/" rvalg rvalf'
          | "%" rvalg rvalf'
          | epsilon
rvalg ::= "-" rvalg
          | "!" rvalg
          | rvalh
rvalh ::= "(" expr ")"
          | rvali
rvali ::= IDENT
          | IDENT "(" args ")"
          | rvalj
rvalj ::= INT_LIT | FLOAT_LIT | BOOL_LIT
args ::= arg_list
          | epsilon
arg_list ::= expr arg_list'
```

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

4.2 FIRST & FOLLOW sets

name	FIRST set	FOLLOW set
arg_list	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")"
arg_list	"," epsilon	")"
args	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT epsilon	")"
block	"{"	"!" "(" "-" ";" "bool" "else" "float" "if" "int" "return" "void" "while" "{" "}" EOF BOOL_LIT FLOAT_LIT IDENT INT_LIT
decl	"bool" "float" "int" "void"	"bool" "float" "int" "void" EOF
decl_list	"bool" "float" "int" "void"	EOF
decl_list	"bool" "float" "int" "void" epsilon	EOF
else_stmt	"else" epsilon	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
expr	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")" "," ";"
expr_stmt	"!" "(" "-" ";" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
extern	"extern"	"bool" "extern" "float" "int" "void"
extern_list	"extern"	"bool" "float" "int" "void"
extern_list	"extern" epsilon	"bool" "float" "int" "void"
fun_decl	"bool" "float" "int" "void"	"bool" "float" "int" "void" EOF
if_stmt	"if"	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT

local_decl	"bool" "float" "int"	"!" "(" "-" ";" "bool" "float" "if" "int" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
local_decls	"bool" "float" "int" epsilon	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
param	"bool" "float" "int"	")" ","
param_list	"bool" "float" "int"	")"
param_list	"," epsilon	")"
params	"bool" "float" "int" "void" epsilon	")"
program	"bool" "extern" "float" "int" "void"	EOF
return_stmt	"return"	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
return_stmt	"!" "(" "-" ";" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
rval	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")" "," ";"
rval′	"||" epsilon	")" "," ";"
rvalb	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	")" "," ";" "||"
rvalb′	"&&" epsilon	")" "," ";" "||"
rvalc	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"&&" ")" "," ";" "||"
rvalc′	"!=" "==" epsilon	"&&" ")" "," ";" "||"
rvald	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "&&" ")" "," ";" "==" "||"
rvald′	"<" ">" ">=" <=" epsilon	"!=" "&&" ")" "," ";" "==" "||"
rvale	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "&&" ")" "," ";" "<" "==" ">" ">=" "||" <="
rvale′	"+" "-" epsilon	"!=" "&&" ")" "," ";" "<" "==" ">" ">=" "||" <="
rvalf	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "&&" ")" "+" "," "-" ";" "<" "==" ">" ">=" "||" <="

rvalf′	"%" "*" "/" epsilon	"!=" "&&" ")" "+" "," "-" ";" "<" "==" ">" ">=" "||" <="
rvalg	"!" "(" "-" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "%" "&&" ")" "*" "+" "," "-" "/" ";" "<" "==" ">" ">=" "||" <="
rvalh	"(" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "%" "&&" ")" "*" "+" "," "-" "/" ";" "<" "==" ">" ">=" "||" <="
rvali	BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!=" "%" "&&" ")" "*" "+" "," "-" "/" ";" "<" "==" ">" ">=" "||" <="
rvalj	BOOL_LIT FLOAT_LIT INT_LIT	"!=" "%" "&&" ")" "*" "+" "," "-" "/" ";" "<" "==" ">" ">=" "||" <="
stmt	"!" "(" "-" ";" "if" "return" "while" "{" BOOL_LIT FLOAT_LIT IDENT INT_LIT	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT
stmt_list	"!" "(" "-" ";" "if" "return" "while" "{" BOOL_LIT FLOAT_LIT IDENT INT_LIT epsilon	"}"
type_spec	"bool" "float" "int" "void"	IDENT
var_decl	"bool" "float" "int"	"bool" "float" "int" "void" EOF
var_type	"bool" "float" "int"	IDENT
while_stmt	"while"	"!" "(" "-" ";" "if" "return" "while" "{" "}" BOOL_LIT FLOAT_LIT IDENT INT_LIT

5 External sources

- Slides and information on the module webpage; `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs325/`
- Stackoverflow;
  `https://stackoverflow.com/questions/7273326/getting-the-nth-line-of-...`
  `https://stackoverflow.com/questions/36802354/print-binary-tree-in-a-...`
- LLVM documentation `https://releases.llvm.org/9.0.1/docs/tutorial/LangImpl07.html`