Compiler Design Coursework Report (1830744)

Parser Design

The parser was designed by harnessing the fact that the given grammar can be transformed into an LL(k) form to implement a recursive descent parser, using k look-ahead symbols select the correct production at each step of the top-down parse and prevent backtracking. After eliminating left recursion and left-factoring to get the grammar into a form matching the LL(1) condition [1], the grammar was found to be in an LL(3) form ($k = 3$), requiring at most 3 look-ahead symbols to correctly parse the incoming stream of tokens from the lexer without backtracking, however in most cases it sufficed to use only 1 look-ahead symbol.

After computing the FIRST and FOLLOW sets of the grammar, the recursive descent parser was implemented via predictive parsing, where the look-ahead symbols and the FIRST and FOLLOW sets were used to uniquely determine the correct production to pick at each step of the parse [1]. The implementation mostly trivial and has a parse method for each non-terminal in the grammar, but some productions were non-trivial to parse and required more than 1 look-ahead symbol to parse, specifically `decl` requiring 3 and `expr` requiring 2. Since `decl` may either be a `var_decl` or `fun_decl`, which are both not nullable and have overlapping FIRST sets, and the second symbol of the `var_decl` and `fun_decl` productions is an `IDENT`, we look at the third look-ahead symbol, which is either `;` for `var_decl` or `(` for `fun_decl`, to determine the correct production. Similarly, for `expr`, we must look at the second look-ahead symbol to determine if it's a stand-alone `IDENT` or an `rval`, since `IDENT` is in the FIRST set for `rval` and it is not nullable.

During the parse method for each production, the Abstract Syntax Tree (AST) is constructed, by recursively constructing an AST node for each production, encoding the non-trivial contents of each production. Each production has an AST node deriving from the AST node base class, and for some non-terminals which have many productions (for example `stmt` and `element`), they have their own base class deriving from the base class of the AST node, which each of its productions derive from (thus creating a depth 3 inheritance). Such an approach has the advantage of polymorphism, enabling non-terminals with many productions to be elegantly and efficiently encoded in the tree, rather than bloating a single class with many optional values.

Regarding the contents of each AST node class, only the non-trivial contents of the production are contained within them (i.e. non-constant data), such as pointers to the non-terminals in its productions, and tags which clarify which production was taken (for example, a boolean to uniquely distinguish two productions). One interesting design choice was to use two vectors encoding the operand values and the operators for expression AST nodes (`rval`, `term`, `equiv`, `rel`, `subexpr` and `factor`). Such vectors are ordered from left-to-right of appearance, with the operator at index i in the operators vector being applied to the operands at indexes $i - 1$ and i in the operands vector. Such an approach enables easy application of left-to-right associativity when computing the result of the expression (which applies to our operators), via a typical left fold implementation (for example, `foldl| in Haskell`).

To print the AST, the AST nodes require a virtual `to_string` method to be implemented, which recursively serializes the contents of the node and its children (pointers to other AST nodes) into a `StringCollection` (a custom tree of strings data-structure that is string-serializable), which is thus serialized into a space-indented string in order to print the AST. This approach is taken to isolate string formatting to the `StringCollection` class, which is concisely implemented recursively and uniformly, rather than repeating formatting in the `to_string` for each AST node which can easily be inconsistent.

Code Generation Semantic Checks

The AST is converted into LLVM IR code by using LLVM's API to emit instructions. Each AST node has a `codegen` virtual method which is implemented to emit its corresponding LLVM IR code, and check the generated code for any semantic errors.

One of the most common semantic errors that can arise is a type error, where for example a value is assigned an incorrect type. However, checking type conformance isn't as trivial as checking a direct match between a variable's type and its value, since in C up-casting occurs with its primitive values, where a boolean can be up-casted into an int, and an integer can be up-casted into a float, since such conversions do not lose any data from the value in the old type. Thus, up-casting has been implemented, along with a series of helper methods for casting for code-reuse. Indeed, in situations where down-casting is required (casting and potentially losing data from the value in the original type), a semantic error is reported. Such a check is done by obtaining the "maximum" type between the original and destination types, and if the types are not equal and the destination type is maximal, then we know the original type is lesser, thus a down-cast is required (here, we rank types with boolean being least and float being maximal). Type non-conformance errors may occur when: a variable is assigned a value of an incorrect type; when the value returned from a function doesn't match its return type; when conditions evaluate to void values (and thus cannot be evaluated as booleans); when the type of the arguments in a function call do not match their type in the function's prototype; and when the modulo (%) operator is applied with non-integer operands (every other operator works with operands of any type, up-casting to bring the operands to the same type as necessary).

The next most common semantic errors relate to the scoping and declaration of variables and functions. Whenever a variable or function is referenced (via a variable expression, function call or variable assignment), we look them up in the global maps storing the variables and functions in the current scope, and a semantic error is reported if they do not exist (and are thus undeclared). However, for variables, such a lookup prefers a local declaration over a global declaration if it exists, since in C global variables may be re-declared locally. Regarding re-declaration, a semantic error is reported when a local variable is re-declared in the same block, or when a global variable or function is re-declared. However, local variables may be re-declared in different blocks, favoring the value in the deepest block. Such behaviour is implemented by saving the value of all of the local variables that are re-declared in a block before its code is emitted, and after it's fully emitted the values of the local variables are restored to their values in the outer scope (and the local variables that did not exist outside the block are removed). Finally, a semantic error is reported if the number of values passed to a function in a function call does not match the number of arguments in its definition.

The last semantic error that is reported is when a function returns a value, but not all of its potential code-paths return a value (and thus we cannot guarantee the function returns a value). Since a function's body is a `block` consisting of a list of statements, where each `block` can be one of an `expr_stmt`, `block`, `if_stmt`, `while_stmt` or `return_stmt`, we check each statement in the function's `block` in-tern seeking a statement that guarantees a return, and if necessary, recursively check the statement if it's composed of its own `block`s. Firstly, if there exists any `return_stmt` in the function's original `block`, then since this return statement is at the top level in the function's body, it's guaranteed to be reached, so the function always returns a value. Next, for each `block` inside the function's `block`, we recursively check if there is a return from all code-paths inside it, and if any block satisfies this check then we can guarantee a return from the function since the block will always be reached. Finally, for each `if_stmt` (if statement), we check whether it has an else block, and recursively check if there is a return from all code-paths in its if and else body. Any if statement matching such a check guarantees a return from the if statement as a whole (since either the if block or else block will be entered, both of which always return) so we can thus guarantee a return from the function since the if statement will always be reached. For all other statement types (`expr_stmt` and `while_stmt`), we can never guarantee a return from them, so they are not checked. If no check succeeded, we report a semantic error since we cannot guarantee a return from the function.

Other Sources

Code and ideas for this coursework were harnessed from online sources such as the LLVM Tutorial, LLVM's Documentation, and StackOverflow. In particular, Chapter 2 in the LLVM Tutorial was a very helpful reference for implementing the parser, and Chapters 3, 5 and 7 contained useful information for code generation. In addition, Oliver Cole's (a recent Computer Science graduate from Warwick) test suite was used, which contained more test cases with respect to code generation and semantic checks (he publicly shared it).

Limitations

While my solution works for all of the test cases, it has several limitations. One improvement would be to split the implementation into multiple files, instead of being all contained within `mccomp.cpp`, which would greatly assist with maintainability. Another improvement would be to add line numbers to the error messages from code generation / semantic analysis (one way to achieve this would be to add line numbers to the AST nodes) and in general add more context to the error messages. Finally, the compiler could have an additional semantic check for the overflow of integer and float data types.

References

[1] G. Mudalige. *University of Warwick CS325 Semantic Analysis Slides*. URL: `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs325/` (visited on Nov. 22, 2021).

Grammar

⟨*program*⟩ ::= ⟨*extern_list*⟩ ⟨*decl_list*⟩
  | ⟨*decl_list*⟩

⟨*extern_list*⟩ ::= ⟨*extern*⟩ ⟨*extern_list*⟩
  | ⟨*extern*⟩

⟨*extern*⟩ ::= `extern` ⟨*type_spec*⟩ IDENT `(` ⟨*params*⟩ `)` `;`

⟨*decl_list*⟩ ::= ⟨*decl*⟩ ⟨*decl_list*⟩
  | ⟨*decl*⟩

⟨*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_list2*⟩

⟨*param_list2*⟩ ::= `,` ⟨*param*⟩ ⟨*param_list*⟩
  | ϵ

⟨*param*⟩ ::= ⟨*var_type*⟩ IDENT

⟨*block*⟩ ::= `{` ⟨*local_decls*⟩ ⟨*stmt_list*⟩ `}`

⟨*local_decls*⟩ ::= ⟨*var_decl*⟩ ⟨*local_decls*⟩
  | ϵ

⟨*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 ⟨expr_stmt⟩

⟨expr⟩ ::= IDENT = ⟨expr⟩
  |  ⟨rval⟩

⟨rval⟩ ::= term ⟨rval2⟩

⟨rval2⟩ ::= || ⟨term⟩ ⟨rval2⟩
  |  ϵ

⟨term⟩ ::= ⟨equiv⟩ ⟨term2⟩

⟨term2⟩ ::= && ⟨equiv⟩ ⟨term2⟩
  |  ϵ

⟨equiv⟩ ::= ⟨rel⟩ ⟨equiv2⟩

⟨equiv2⟩ ::= == ⟨rel⟩ ⟨equiv2⟩
  |  != ⟨rel⟩ ⟨equiv2⟩
  |  ϵ

⟨rel⟩ ::= ⟨subexpr⟩ ⟨rel2⟩

⟨rel2⟩ ::= <= ⟨subexpr⟩ ⟨rel2⟩
  |  < ⟨subexpr⟩ ⟨rel2⟩
  |  >= ⟨subexpr⟩ ⟨rel2⟩
  |  > ⟨subexpr⟩ ⟨rel2⟩
  |  ϵ

⟨subexpr⟩ ::= ⟨factor⟩ ⟨subexpr2⟩

⟨subexpr2⟩ ::= + ⟨factor⟩ ⟨subexpr2⟩
  |  - ⟨factor⟩ ⟨subexpr2⟩
  |  ϵ

⟨factor⟩ ::= ⟨element⟩ ⟨factor2⟩

⟨factor2⟩ ::= * ⟨element⟩ ⟨factor2⟩
  |  / ⟨element⟩ ⟨factor2⟩
  |  % ⟨element⟩ ⟨factor2⟩
  |  ϵ

⟨element⟩ ::= - ⟨element⟩
  |  ! ⟨element⟩
  |  ( ⟨expr⟩ )
  |  IDENT
  |  IDENT ( ⟨args⟩ )
  |  INT_LIT
  |  FLOAT_LIT
  |  BOOL_LIT

⟨args⟩ ::= ⟨arg_list⟩
  |  ϵ

⟨arg_list⟩ ::= ⟨expr⟩ ⟨arg_list2⟩

⟨arg_list2⟩ ::= , ⟨expr⟩ ⟨arg_list2⟩
  |  ϵ
```

FIRST Sets

```
arg_list2 = {",", ""}
arg_list = {IDENT, "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
args = {IDENT, "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, ""}
factor2 = {"*", "/", "%", ""}
element = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
factor = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
subexpr = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
subexpr2 = {"+", "-", ""}
rel = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
rel2 = {"<=", "<", ">=", ">", ""}
equiv = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
equiv2 = {"==", "!=", ""}
term = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
term2 = {"&&", ""}
rval = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
rval2 = {"||", ""}
expr = {"-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
return_stmt = {"return"}
else_stmt = {"else", ""}
if_stmt = {"if"}
while_stmt = {"while"}
expr_stmt = {";", "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT}
block = {"{"}
stmt = {";", "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, "{", "if",
"while", "return"}
stmt_list = {";", "-", "!", "(", IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, "{",
"if", "while", "return", ""}
var_type = {"int", "float", "bool"}
local_decls = {"int", "float", "bool", ""}
param = {"int", "float", "bool"}
param_list2 = {",", ""}
param_list = {"int", "float", "bool"}
params = {"int", "float", "bool", "void", ""}
type_spec = {"void", "int", "float", "bool"}
var_decl = {"int", "float", "bool"}
fun_decl = {"void", "int", "float", "bool"}
decl = {"int", "float", "bool", "void"}
decl_list = {"int", "float", "bool", "void"}
extern = {"extern"}
extern_list = {"extern"}
program = {"extern", "int", "float", "bool", "void"}
```

FOLLOW Sets

```
program = {"eof"}
extern_list = {"int", "float", "bool", "void"}
extern = {"int", "float", "bool", "void", "extern"}
decl_list = {"eof"}
decl = {"eof", "int", "float", "bool", "void"}
var_decl = {"eof", "int", "float", "bool", "void"}
type_spec = {IDENT}
var_type = {IDENT}
fun_decl = {"eof", "int", "float", "bool", "void"}
params = {")"}
param_list = {",", ")"}
param_list2 = {",", ")"}
```

```
param = {",", ")"}
local_decls = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{",
"if", "while", "return", "}"}
stmt_list = {"}"}
stmt = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{", "if",
"while", "return", "}"}
expr_stmt = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{",
"if", "while", "return", "}"}
while_stmt = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{",
"if", "while", "return", "}"}
if_stmt = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{", "if",
"while", "return", "}"}
else_stmt = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{",
"if", "while", "return", "}"}
return_stmt = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{",
"if", "while", "return", "}"}
block = {";", IDENT, "-", "!", "(", INT_LIT, FLOAT_LIT, BOOL_LIT, "{", "if",
"while", "return", "}", "eof", "int", "float", "bool", "void"}
args = {")"}
arg_list = {")"}
arg_list2 = {")"}
expr = {";", ")", ","}
rval = {";", ")", ","}
rval2 = {";", ")", ","}
term = {"||", ";", ")", ","}
term2 = {"||", ";", ")", ","}
equiv = {"&&", "||", ";", ")", ","}
equiv2 = {"&&", "||", ";", ")", ","}
rel = {"==", "!=", "&&", "||", ";", ")", ","}
rel2 = {"==", "!=", "&&", "||", ";", ")", ","}
subexpr = {"<=", "<", ">=", ">", "==", "!=", "&&", "||", ";", ")", ","}
subexpr2 = {"<=", "<", ">=", ">", "==", "!=", "&&", "||", ";", ")", ","}
factor = {"+", "-", "<=", "<", ">=", ">", "==", "!=", "&&", "||", ";", ")", ","}
factor2 = {"+", "-", "<=", "<", ">=", ">", "==", "!=", "&&", "||", ";", ")", ","}
element = {"*", "/", "%", "+", "-", "<=", "<", ">=", ">", "==", "!=", "&&", "||",
";", ")", ","}
```