SCC by KREG

"Not-very-good C Compiler"

Krishna Marentes
Rebecca Castillo
Elijah Orozco
Geoff Knox

April 6 2020

Contents

1 Program Overview

This file serves as the documentation for our compiler, which currently can scan and parse C code and generate a linear IR. Our user guide can be found in "usage.pdf". Our compiler is written in Java and currently consists of two parts:

- Phase 1: Scanning/Parsing with C Grammar
  - Input: C code
  - Output: Abstract Syntax Tree
- Phase 2: IR Generation
  - Input: Abstract Syntax Tree
  - Output: Linear intermediate representation

A third-party tool, "Antlr," serves as the scanner/parser generator that takes our Grammar (stored in the .g4 file) as input. Antlr is prompted to generate and run scanning and parsing files with the specified grammar from the Driver file, which handles command line arguments, the compiler/Antlr interface, and general "main" function duties.

2 Scanning and Parsing - Antlr

2.1 Overview

Our compiler uses Antlr4, a tool similar to Flex/Bison that can be used for compilers written in Java. Antlr is both a scanner and parser, and uses a simple format for

specifying the language to parse.
We originally began our design handwriting our scanner code, with the same plan for the parser. However, Antlr was an attractive choice for us due to the advantages outlined below.

2.2 Advantages

Antlr allowed us to fast-track our parser development by greatly reducing the amount of code we needed to design and write. Antlr handles the parsing algorithm so that we can avoid writing tedious and error-prone code ourselves. Additionally, the grammar files Antlr uses are simple and easy to write. There is also a convenient plugin for our team's IDE of choice, IntelliJ, that seamlessly incorporates Antlr into our development environment, including a useful dynamic parse tree display.

2.3 Disadvantages

As simple as Antlr grammar files are and as much function as Antlr provides, there is still the learning curve associated with using and integrating such a complex tool. We are also aware that Antlr grammar files can be difficult to debug, but we are prepared to handle such issues, especially with the tools the IntelliJ Antlr plugin provides.

It was discovered when beginning Phase 2 that Antlr left us with no convenient way to convert the automatically-generated parse tree into an abstract syntax tree. Therefore, some extra effort was required to convert Antlr's tree data structure into our own design for AST generation and conversion to IR.

3 The Syntax Tree

3.1 Overview

As discussed before, Antlr generates a thorough parse tree of the source code according to the defined grammar, but leaves no obvious way to use the generated parse tree. Therefore, our compiler defined its own tree data structure and methods to convert the parse tree (defined in Antlr's own tree data structure) into our custom tree design. This conversion also trims the parse tree into an abstract syntax tree (AST) for clarity and convenience for generating the IR later. Tree conversion routines are held in the `ASTNode` class.

To aid IR generation, each AST node is resolved into the a certain subclass node according to the grammar. For example, a node holding the "funDeclaration" grammar rule in the parse tree would be resolved into a node of type "FunDeclaration" in the AST. The AST uses seven subclasses of AST nodes: Program, TypeSpecifier, VarDeclaration, FunDeclaration, StructDeclaration, Expression, and Statement. (Note that the StructDeclaration is not fully implemented in the current version.) Each subclass is a customized node with relevant members (such as “params” for FunDeclaration) that in some cases reduces the total number of nodes in the AST.

3.2 Advantages

The advantages of this design center on the ease of AST printing and IR generation. IR generation routines are customized for each subclass, so that IR generation can be successfully performed by recursively walking the tree and checking for what instance each node belongs to. This design is also useful when using an IDE debugger, since each node will have relevant members directly in the object, rather then needing to look through layers of child and parent nodes.

3.3 Disadvantages

For all the advantages, disadvantages do exist for our approach. For one, the overall design is somewhat complex. Using the AST requires fairly deep knowledge of each subclass as well as the grammar itself. In addition, it can be confusing to know when to look for information in a node’s members or in its children. However, we’ve found the current design to be sufficiently helpful for IR

4 Language Specification

4.1 Introduction

Here are the types of the various elements by type font or symbol for the grammar that follows:

- **Terminals, including terminal punctuation, are bolded**
- TOKENS ARE ALL CAPS
- Nonterminals are in “normal” font
- The symbol ϵ means the empty string
- “EOF” indicates the end of the file

The grammar also uses standard regex terms. Symbols indicating regex rules are never bolded so as not to be confused with terminal symbols. Parentheses are also used to avoid confusion. For example, an instance of “(*****)*” indicates that the * terminal may be seen zero or more times.

Single-line comments (denoted by “//”) and block comments (denoted by “/* */”) are successfully ignored by the grammar.

4.2 Tokens

- ID → (_ | [a-zA-Z])+ ([a-zA-Z] | DIGIT | _)*
- CHARCONST → ' CHARCHARS+ '
- STRINGCONST → " STRINGCHARS* "
- INT → DIGIT+
  | (**0x** | **0X**)HEXDIGIT+
  | **0**OCTALDIGIT+
  | (**0b** | **0B**)BINARYDIGIT+
  | FLOAT

The following are "helper tokens" that are not strictly considered tokens by our compiler but serve to help define the tokens above.

- CHARCHARS →

```
(~['\\\r\n] | '\\' (. | EOF))
```

- LETTER → [**a-zA-Z**]
- DIGIT → [**0-9**]
- HEXDIGIT → [**0-9A-Fa-f**]
- OCTALDIGIT → [**0-7**]
- BINARYDIGIT → [**0-1**]
- FLOAT → [**0-9**]+ **.** [**0-9**]+ EXP?(**f**|**F**)?
  | **.** [**0-9**]+ EXP?(**f**|**F**)?
  | [**0-9**]+ EXP(**f**|**F**)?
  | [**0-9**]+ (**f**|**F**)?
- EXP → (**e**|**E**) (+|-)? [**0-9**]+

4.3 The Grammar

1. program → declarationList **EOF**
2. declarationList → declarationList declaration | declaration
3. declaration → varDeclaration | funDeclaration | structDeclaration | enumDeclaration | **;**
4. structDeclaration → **static**? **struct** ID **{** unInitVarDecl* **}** ID? **;**

5. structInit → **static**? **struct** ID (*)* varDeclList **;**

6. enumDeclaration → **enum** ID **{** enumDeclList **}** ID? **;**

7. enumDeclList → enumDeclList **,** enumId | enumId | ϵ

8. enumId → ID ASSIGNMENT enumExpression | ID

9. enumInit → **enum** ID ID **;** | **enum** ID ID = expression **;**

10. unInitVarDecl → typeSpecifier unInitVarDeclList **;**

11. unInitVarDeclList → unInitVarDeclList **,** varDeclId | varDeclId | ϵ

12. varDeclaration → typeSpecifier varDecList **;** | scopedVarDeclaration **;** | structInit | enumInit

13. scopedVarDeclaration → scopedTypeSpecifier varDeclList

14. forLoopVars typeSpecifier varDeclList | expression List

15. varDeclList → varDeclList **,** varDeclInitialize | varDeclInitialize

16. varDeclInitialize → varDeclId | varDeclId = (expression | **{** expressionList **}**)

17. varDeclId → ID ([expression?])*

18. scopedTypeSpecifier → **static** typeSpecifier | typeSpecifier

19. typeSpecifier → (**int** | **float** | **double** | **char** | **long** | **unsigned** | **signed** | **void** | **short**)(*)*

20. funDeclaration → typeSpecifier ID **(** params **)** (compoundStmt | **;**)

21. params → params **,** parameter | parameter | ϵ

22. parameter → typeSpecifier paramId

23. paramId → ID ([])*

24. statement → expressionStmt | compoundStmt | selectionStmt | iterationStmt | returnStmt | breakStmt | gotoStmt | labelStmt

25. structExpressionList → expressionList **,** **.**? expression | **.**? expression | ϵ

26. expressionList → expressionList **,** expression | expression | ϵ

27. expressionStmt → expression **;** | **;**

28. compoundStmt → **{** statementList **}**

29. statementList → statementList (statement | varDelcaration) | **EPS**

30. defaultList → statementList (statement | varDeclaration) | statement

31. elsifList → elsifList **else if (** expression **)** statement | **EPS**

32. selectionStmt → **if (** expression **)** statement elsifList
    | **if (** expression **)** statement elsifList **else** statement
    | **switch (** expression **)** switchCase defaultList
    | **switch (** expression **) default :** defaultList
    | **switch (** expression **) {** switchList (**default :** defaultList)? **}**

33. switchList → switchList switchCase | switchCase | ϵ

34. switchCase → **case** (**INT** | **CHARCONST**) **:** (defaultList | statementList)

35. iterationStmt → **while (** expression **)** statement
    | **do** statement **while (** expression **);**
    | **for (** forLoopVars **;** expressionList **;** expressionList **)** statement

36. returnStmt → **return ;** | **return** expression **;**

37. breakStmt → **break ;**

38. gotoStmt → **goto** labelId **;**

39. labelStmt → labelId **:**

40. labelId → ID

41. expression → mutable = expression
    | mutable **+=** expression
    | mutable **-=** expression
    | mutable ***=** expression
    | mutable **/=** expression
    | mutable **%=** expression
    | mutable <<= expression
    | mutable >>= expression
    | mutable **&=** expression
    | mutable |= expression
    | mutable ˆ= expression
    | (mutable | immutable) << expression
    | (mutable | immutable) >> expression
    | (mutable | immutable) **&** expression
    | (mutable | immutable) | expression
    | (mutable | immutable) ˆ expression
    | simpleExpression

42. enumExpression → properUnaryOps INT << enumExpression
    | properUnaryOps INT >> enumExpression

| properUnaryOps INT & enumExpression
| properUnaryOps INT | enumExpression
| properUnaryOps INT ˆ enumExpression
| properUnaryOps INT || enumExpression
| properUnaryOps INT && enumExpression
| properUnaryOps INT < enumExpression
| properUnaryOps INT <= enumExpression
| properUnaryOps INT > enumExpression
| properUnaryOps INT >= enumExpression
| properUnaryOps INT + enumExpression
| properUnaryOps INT - enumExpression
| properUnaryOps INT * enumExpression
| properUnaryOps INT

43. properUnaryOps → (- | ˜ | !)*

44. simpleExpression → (simpleExpression || andExpression) | andExpression

45. andExpression → andExpression && unaryRelExpression | unaryRelExpression

46. unaryRelExpression → ! unaryRelExpression | relExpression

47. relExpression → sumExpression relop sumExpression | relExpression relop relExpression | sumExpression

48. relop → <= | < | > | >= | == | !=

49. sumExpression → sumExpression sumop mulExpression | mulExpression

50. sumop → + | -

51. mulExpression → mulExpression mulop unaryExpression | unaryExpression

52. mulop → * | / | %

53. unaryExpression → unaryop unaryExpression | mutable ++ | mutable – | – mutable | ++ mutable | factor

54. unaryop → - | * | ! | & | ˜

55. factor| → immutable | mutable

56. mutable → (*)* ID
    | mutable [expression]
    | mutable (. | ->) mutable
    | immutable (. | ->) mutable
    | (expression) (. | ->) mutable
    | (*)* (expression)

57. immutable → **(** expression **)** | call | constant

58. call → ID **(** args **)** | **sizeof (** (**struct** ID (*)* | typeSpecifier | ID) **)**

59. args → argList | ϵ

60. argList → argList **,** expression | expression

61. constant → INT | CHARCONST | STRINGCONST

4.4 Semantic Notes

- HEX, OCTAL, and BINARYDIGIT default to **int** when parsed
- Many variables can be declared and/or initialized in one statement, e.g. “int a = 1, b = 2;”

4.5 Limitations

The following are not supported by our grammar.

- Preprocessor statements
- Casting
- Ternary operations

We’ve attempted to implement the following in the grammar, but support can be considered to be in “beta mode” as there may be edge cases that have not been tested.

- Pointers
- Arrays
- Strings

5 Symbol Table

Our compiler utilizes a symbol table to keep track of declared variables and functions and their types. The symbol table is built from the complete abstract syntax tree. The general structure of our symbol table with examples is shown:

Name	Type
i	int
c	char
main	int

The “Name” field of our symbol table can be of two different types: a plain `SymbolEntry` type for variable declarations, or a `SymbolTable` type for function declarations. This way, function return types are stored in the symbol table belonging to the scope they are declared in, and each function’s scoped variables can be found by looking inside the symbol table of the function’s name.

For example, the following C code

```
int global_var;
char foo() {
    char i;
    return i;
}
int main() {
    int i;
    char j;
    return i;
}
```

produces the following symbol table:

Name		Type
global_var		int
foo		char
Name	Type	
i	char	
main		int
Name	Type	
i	int	
j	char	

The symbol table mainly serves to check for type and declaration errors in the source code. For example, if the source code tries to reference variable “i” and “i” is not found in the symbol table, the compiler will throw an error to the user.

6 Intermediate Language Design

6.1 Overview

Our linear intermediate representation (IR) follows a simple format not far from the original C code. It flattens all loops with `goto` statements, and uses `if` and `else` to handle conditional jumps. Our IR nearly follows the single static assignment (SSA) form.

Temporary variables and labels are named with the prefix “KREG” followed by a period and unique integer: “`KREG.0`”. Labels are differentiated from variables by angle brackets: “`<KREG.0>`”. We rely on the period in our temporary variable naming scheme to avoid naming collisions between IR-generated variables and C code variables.

Statements end with a semicolon, and labels end with a colon. Function bodies are denoted with curly brackets, each on their own line. Function parameters are denoted in parenthesis next to the function name, similar to C. All function declarations are preceded by the keyword “function”. Though our algorithm does sometimes result in seemingly redundant temporary variable assignments (see example below) by using pseudo-SSA, we find that this method smoothly and easily breaks the original code down into manageable lines for optimization.

For the following line of C code:

```
i++;
```

our compiler will produce the following IR:

```
KREG.1 = i;
i = i + 1;
KREG.2 = KREG.1;
```

Although in this case our IR unnecessarily inflates the original C code, it has no real impact on the program’s function. More importantly, producing the IR in this way allows us to use the proper values of variables for unary expressions. For example, if the C code was changed to

```
x = i++;
```

then our IR would produce

```
KREG.1 = i;
i = i + 1;
KREG.2 = KREG.1;
x = KREG.2;
```

which correctly assigns the un-incremented value of `i` to `x`.

Currently, our IR is stored as a single string, with each line of code separated by *EOL*. Design concepts for the best structure to store the IR for optimization and conversion to assembly are currently underway.

6.2 Limitations

The following features, in addition to those not supported by the grammar, are not supported by our IR:

- For loops
- Pointers
- Strings
- Structures and Enums
- Arrays
- Multiple boolean expressions in `if` statements and `while` loops. E.g.

```
while (i < 10 && j > 10)
```

Currently, our compiler only supports two levels of symbol scoping: the global scope, and scopes for functions. Therefore, loops do *not* have their own scope.