CSE423 Assignment 1 Documentation

Rebecca Castillo
Geoff Knox
Krishna Marentes
Elijah Orozco

February 16 2020

1 Program Overview

This file serves as the documentation for our compiler, currently in the scanning/parsing stages. Our user guide can be found [insert build file]. We have written our compiler in the Java language. Our compiler currently consists of three parts:

- Grammar
- Scanner/Parser Generator
- Driver

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 Usage

At present, our compiler takes input from a .c file, and can output results to the terminal or to a file. Command-line arguments are available for the following functions:
Put the stuff we have here

3 Design - Antlr

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 all at once, 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.

3.1 Advantages

Antlr allowed us to fast-track our parser development by greatly reducing the amount of code we need 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 parse tree display.

3.2 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.

4 Language Specification

4.1 The Grammar

1. start → program **EOF**
2. program → declarationList
3. declarationList → declarationList declaration | declaration
4. declaration → varDeclaration | funDeclaration
5. varDeclaration → typeSpecifier varDecList **;**
6. scopedVarDeclaration → scopedTypeSpecifier varDeclList **;**
7. varDeclList → varDeclList **,** varDeclInitialize | varDeclInitialize
8. varDeclInitialize → varDeclId | varDeclId **=** expression
9. varDeclId → ID
10. scopedTypeSpecifier → **static** typeSpecifier | typeSpecifier
11. typeSpecifier → **int** | **float** | **double** | **char** | **long** | **unsigned** | **signed** | **void** | **short**
12. funDeclaration → typeSpecifier ID **(** params **)** (compoundStmt | ;+)

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

14. parameter → typeSpecifier paramId

15. paramId → ID

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

17. expressionStmt → expression **;** | **;**

18. compoundStmt → **{** localDeclarations statementList **}**

19. localDeclarations → localDeclarations scopedVarDeclaration | **EPS**

20. statementList → statementList statement | **EPS**

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

22. selectionStmt → **if (** expression **)** statement elsifList | **if (** expression **)** statement elsifList **else** statement

23. iterationStmt → **while (** expression **)** statement | **do** statement **while (** expression **);**

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

25. breakStmt → **break ;**

26. gotoStmt → **goto** labelId **;**

27. labelStmt → labelId **:**

28. labelId → ID

29. expression → mutable **=** expression | mutable **+=** expression | mutable **-=** expression | mutable *= expression | mutable **/=** expression | simple-Expression

30. simpleExpression → simpleExpression || andExpression | andExpression

31. andExpression → andExpression unaryRelExpression | unaryRelExpression

32. unaryRelExpression → **!** unaryRelExpression | relExpression

33. relExpression → sumExpression relop sumExpression | relExpression relop relExpression | sumExpression

34. relop → <= | < | > | >= | **==** | **!=**

35. sumExpression → sumExpression sumop mulExpression | mulExpression

36. sumop → **+** | **-**

37. mulExpression → mulExpression mulop unaryExpression | unaryExpression

38. mulop → ***** | **/** | **%**

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

40. unaryop → **-** | ***** | **!** | |

41. factor| → immutable | mutable

42. mutable → ID | mutable **[** expression **]**

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

44. call → ID **(** args **)**

45. args → argList | **EPS**

46. argList → argList **,** expression | expression

47. constant → INT | CHARCONST | STRINGCONST

48. ID → (_ | LETTER)+ (LETTER | DIGIT | _)*

49. CHARCONST → **'** ALLCHARS+ **'**

50. STRINGCONST → **"** ALLCHARS* **"**

51. INT → DIGIT+ | (**0x** | **0X**)HEXDIGIT+ | **0**OCTALDIGIT+ | **0b**BINARYDIGIT+ | FLOAT

52. DIGIT → [**0-9**]

53. HEXDIGIT → [**0-9A-Fa-f**]

54. OCTALDIGIT → [**0-7**]

55. BINARYDIGIT → [**0-1**]

56. FLOAT → [**0-9**]+ **.** [**0-9**]+ EXP?(**f**|**F**)? | **.** [**0-9**]+ EXP?(**f**|**F**)? | [**0-9**]+ EXP(**f**|**F**)?

57. EXP → (**e**|**E**) (+|**-**)? [**0-9**]+

4.2 Semantic Notes

- HEX, OCTAL, and BINARYDIGIT default to **int** when parsed
- Many variables can be declared and/or initialized in one statement

4.3 Limitations

The following are not supported by our grammar.

- For loops
- Switch statements
- Preprocessor statements
- Casting
- Struct, enum
- Pointers, arrays, and strings
- Ternary operations