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 and parsing stages. Our user guide can be found in "usage.pdf". Our compiler is written in Java and 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 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.

2.1 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.

2.2 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.

3 Language Specification

3.1 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.

3.2 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**]+

3.3 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 ^= expression
    | (mutable | immutable) << expression
    | (mutable | immutable) >> expression
    | (mutable | immutable) & expression
    | (mutable | immutable) | expression
    | (mutable | immutable) ^ expression
    | simpleExpression

42. 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 > 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

3.4 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;"

3.5 Limitations

The following are not supported by our grammar.

- Preprocessor statements
- Casting
- Ternary operations

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

- Pointers
- Arrays
- Strings