Compiler Construction Design Document

Tyler Ray

September 2023

Contents

1 Allowed Tokens

Symbols

```
@ ! \# \$ \% \& ( ) [ ] \{ \} < > + = - | / ; : ’ " , . ?
== <= >= != && || // \\ /* */  ++ --
```

Types

int float char void
: Basic data types.

Type Modifiers

long, unsigned, signed, short
: Type modifiers.

Keywords

if else while for return break continue
: Reserved keywords.

Whitespace

Whitespace characters are not tokenized.

Identifiers

Identifiers should match the regular expression: `r’^[a-zA-Z_][\w_]*$’`

Numbers

Numbers should match one of the following regular expressions:

- `r’^[\d]+$’`
- `r’^[\d]*\.[\d]+$’`
- `r’^0[xX][\dA-Fa-f]+$’` (Hexadecimal)
- `No negative Numbers`

Include

Include statements should match the regular expression: `r’^[A-Za-z_]*[.][h]$’`
**As of right now, our parser does not allow for include statements. This may be a feature added later

Multi-line Strings

Multi-line strings are allowed, but they should not allow for variable printing.

2 Grammar

Below is the current grammar our compiler allows. A capital letter symbolizes a non-terminal, a lowercase first letter symbolizes a literal token

$$
\begin{aligned}
\text{Program} &\rightarrow \text{DeclList} \\
\text{DeclList} &\rightarrow \text{Decl} \mid \text{Decl DeclList} \\
\text{Decl} &\rightarrow \text{Type id (Args) \{Local\_Decls stmtList\} Type id = endofDecl;} \\
\\
\text{Args} &\rightarrow \text{Arg} \mid \text{Arg , Args} \\
\text{Arg} &\rightarrow \text{Type id} \mid \epsilon \\
\\
\text{Local\_Decls} &\rightarrow \text{Local\_Decl} \mid \text{Local\_Decl Local\_Decls} \\
\text{Local\_Decl} &\rightarrow \text{Type id; } \mid \text{Type id = EndOfDecl; } \mid \epsilon \\
\\
\text{StmtList} &\rightarrow \text{Stmt} \mid \text{Stmt StmtList} \\
\text{Stmt} &\rightarrow \text{ReturnStmt} \mid \text{AssignStmt} \mid \text{WhileStmt} \mid \text{IfStmt} \mid \text{FunctionCall} \mid \epsilon \\
\text{ReturnStmt} &\rightarrow \text{return num; } \mid \text{return id; } \mid \text{return; } \mid \text{return expr; } \mid \text{return character; } \mid \text{return string;} \\
\text{AssignStmt} &\rightarrow \text{id = EndOfDecl;} \\
\text{WhileStmt} &\rightarrow \text{while (Conditional\_Expr) \{StmtList\}} \\
\text{IfStmt} &\rightarrow \text{if (Conditional\_Expr) \{StmtList\} } \mid \text{ if (Conditional\_Expr) \{StmtList\} else \{StmtList\}} \\
\text{FunctionCall} &\rightarrow \text{id (Params);} \\
\\
\text{Params} &\rightarrow \text{Param} \mid \text{Param , Params} \\
\text{Param} &\rightarrow \text{Expr} \mid \text{string} \mid \text{character} \\
\\
\text{Conditional\_Expr} &\rightarrow \text{Expr Relop Expr} \\
\text{Relop} &\rightarrow == \mid != \mid > \mid >= \mid < \mid <= \\
\\
\text{EndOfDecl} &\rightarrow \text{Expr} \mid \text{string} \mid \text{character} \\
\\
\text{Expr} &\rightarrow \text{Term + Expr} \mid \text{Term - Expr} \mid \text{Term} \\
\text{Term} &\rightarrow \text{Factor * Term} \mid \text{Factor / Term} \mid \text{Factor} \\
\text{Factor} &\rightarrow \text{num} \mid \text{(Expr)} \mid \text{id} \\
\\
\text{Type} &\rightarrow \text{NumType} \mid \text{void} \mid \text{char} \mid \text{TypeModifier} \\
\text{NumType} &\rightarrow \text{double} \mid \text{int} \mid \text{float} \\
\text{TypeModifier} &\rightarrow \text{signed} \mid \text{unsigned} \mid \text{long} \mid \text{short} \\
\\
\text{string} &\rightarrow "[a-zA-Z]+" \\
\text{character} &\rightarrow' [a-zA-Z]' \\
\text{num} &\rightarrow [0-9]+ \mid [0-9]+.[0-9]+ \\
\text{id} &\rightarrow [A-Za-z'underscore'][A-Za-z0-9'underscore']*
\end{aligned}
$$

3 ParseTree

My Parse Tree is a AST Tree that shows 4 overall things within a function declaration:

- Id - The name of the function
- Type - What type is the function
- Args - What arguments the function has
- Local Declarations - What declarations it creates at the beginning of a declaration
- Statement List - All of the statements after our local declarations

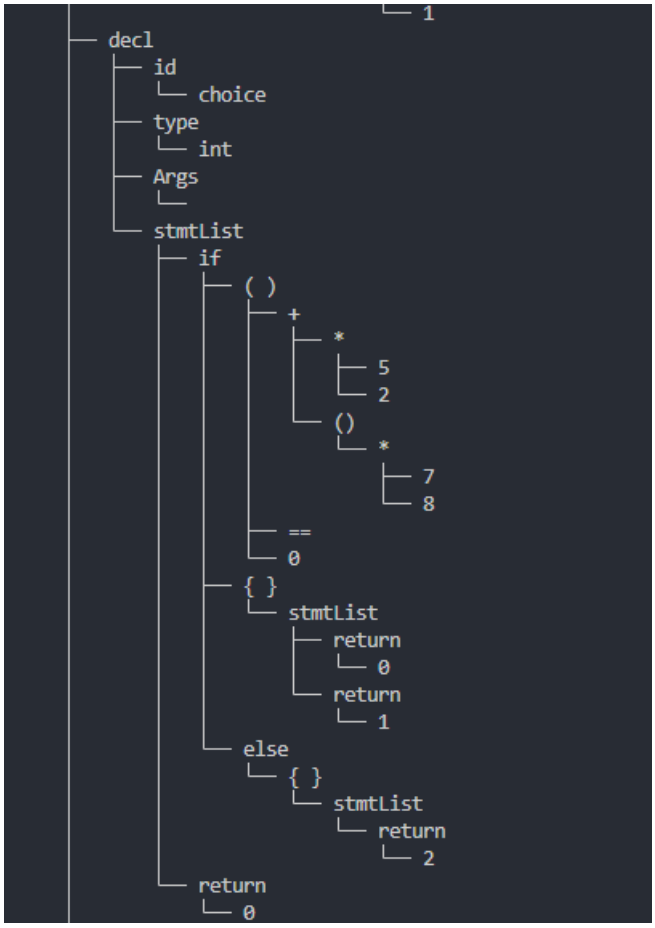

Figure 1: Parse Tree Example

In figure 1 is an example Parse Tree output within our program. At the top is our Id, type, and arguments. Within this example there are no local declarations so it doesn't show up on our tree. I decided to include the 'Args' just to show that there are none regardless. Within the first statement we see an if statement. The nodes after are the parenthesis brackets and in this case, an else statement. The () contain an expression. Our expression tree shows the operation, followed by the two numbers that it will perform that operation on. With multiple operations it will show the operation you must make before you make that one. Our parser will follow the order of operations

4 Usage

```
Compiler.py [-h] [-t] [-p] [-s] File
```

4.1 Positional Arguments

File A valid C input file.

4.2 Options

-h, –help Show this help message and exit.

-t Outputs a tokenized version of the input file.

-p Outputs a parse tree of the input file.

-s Outputs a symbol table of the input table.

5 Caveats

- Not all valid tokens will be allowed within our parser, I left them in for reference to the tokenizer. The parser grammar will be the best source on what tokens are allowed within our compiler.
- You cannot output the parse tree to a text file. It is because thy are viewed as invalid characters
- Comments are removed from out tokenizer before we pass it to our parser, they are ignored
- Parse tree prints out declarations in opposite order, It only does this for global declarations and local declarations. I tried to reverse the list but it will put the type, id and Args at the bottom of the printout.
- You cannot put parenthesis around multiple conditionals

6 Nice Error Messages

- Returning an invalid return type
- Returning an undeclared identifier
- Using an undeclared identifier in a expression
- Using a non-number type in an expression
- Assigning incompatible types
- Missing brackets
- Missing Parenthesis
- No Expr after a relation operation in an statement