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

Program → DeclList
DeclList → Decl | Decl DeclList
Decl → Type id (Args) {Local_Decls stmtList} Type id = endofDecl;

Args → Arg | Arg , Args
Arg → Type id | ϵ

Local_Decls → Local_Decl | Local_Decl Local_Decls
Local_Decl → Type id; | Type id = EndOfDecl; |Type *id; | Type *id = EndOfDecl; | ϵ

StmtList → Stmt | Stmt StmtList
Stmt → ReturnStmt | AssignStmt | WhileStmt | IfStmt | FunctionCall | ϵ
ReturnStmt → return num; | return id; | return; | return expr; | return character; | return string;
AssignStmt → id = EndOfDecl;
WhileStmt → while (Conditional_Expr) {StmtList}
IfStmt → if (Conditional_Expr) {StmtList} | if (Conditional_Expr) {StmtList} else {StmtList}
FunctionCall → id (Params);

Params → Param | Param , Params
Param → Expr | string | character

Conditional_Expr → Expr Relop Expr
Relop → = = | ! = | > | >= | < | <=

EndOfDecl → Expr | string | character

Expr → Term + Expr | Term - Expr | Term
Term → Factor * Term | Factor / Term | Factor
Factor → num | (Expr) | id

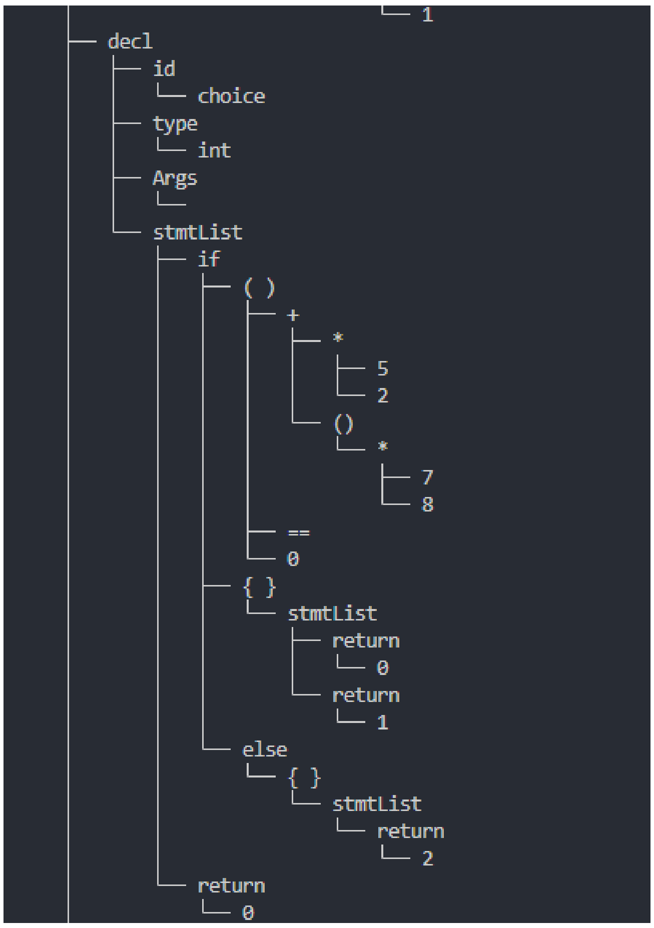
Type → NumType | void | char | TypeModifier
NumType → double | int | float
TypeModifier → signed | unsigned | long | short

string → $"[a-zA-Z]+"$
character → $'[a-zA-Z]'$
num → $[0-9]+ \mid [0-9]+.[0-9]+$
id → $[A-Za-z'underscore'][A-Za-z0-9'underscore']*$

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



```
│                               └── 1
├── decl
│   ├── id
│   │   └── choice
│   ├── type
│   │   └── int
│   ├── Args
│   │   └──
│   └── stmtList
│       ├── if
│       │   ├── ( )
│       │   │   ├── +
│       │   │   │   ├── *
│       │   │   │   │   ├── 5
│       │   │   │   │   └── 2
│       │   │   │   └── ()
│       │   │   │       └── *
│       │   │   │           ├── 7
│       │   │   │           └── 8
│       │   │   ├── ==
│       │   │   └── 0
│       │   ├── { }
│       │   │   └── stmtList
│       │   │       ├── return
│       │   │       │   └── 0
│       │   │       └── return
│       │   │           └── 1
│       │   └── else
│       │       └── { }
│       │           └── stmtList
│       │               └── return
│       │                   └── 2
│       └── return
│           └── 0
```

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 Three Address Code

```
Example -
main:
   L1:
      i = 5
   L2:
      t1 = 5 + 7
      t2 = t1 == 6
      t3 = 5 == 6
      if (t2 && t3) goto L3, else goto L4
   L3:
      return i
      goto L5
   L4:
      return 7
   L5:
      return hi
```

Our compiler will take our AST and turn it into three address code. The Three address code will show the scope, then the basic block and each line within the block. As of now, the compiler accepts while loops and if statements and will be shown within our 3 address code. During the creation of our three address code, a control flow graph will be built.

The 3 address code is not the 3 address code built for the first figure

5 Control Flow Graph and Dominator Graph

After our control flow graph is created, we will use the graph to create a dominator graph of our code. Each function will get a individual section in our graph. And if you choose to print either graph it will space out the nodes that aren't in the same function. Below are examples of a control flow graph 2 and A dominator graph output 3. The graphs are built from the 3 address code in the previous example.

If you choose to print out both graphs, you must close the first one to look at the next one. As of right now there is no way to show both graphs at once. Also, if there is only a single block, a dominator graph will not be printed.

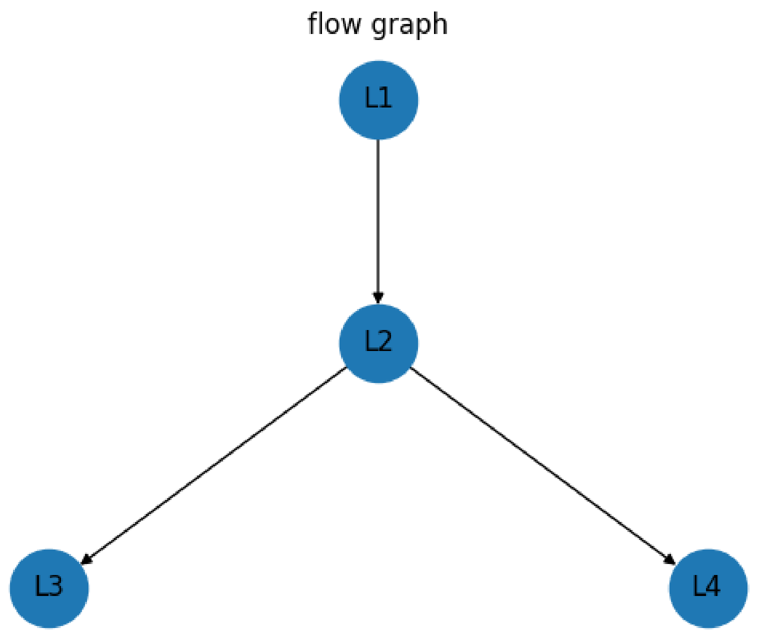


Figure 2: Control Flow Graph Example

6 Optimization

With the created 3 address code, we begin to run passes over the code to see if we can optimize it. If there is any of the following optimizations completed, we will pass over the code again looking to see if we can run the optimizations again. This will repeat until there are no more optimizations. It is worth noting that the reason we create a control flow graph and dominator graph is so we can perform invariant lifting on loops.

At this point in our compiler all floats and doubles are ignored, they will be converted to ints and rounded down

These are the following optimizations that we run over our three address code:

```
Constant Propagation
Constant Folding
Dead Code Removal
Copy Propagator
Invariant Lifting
```

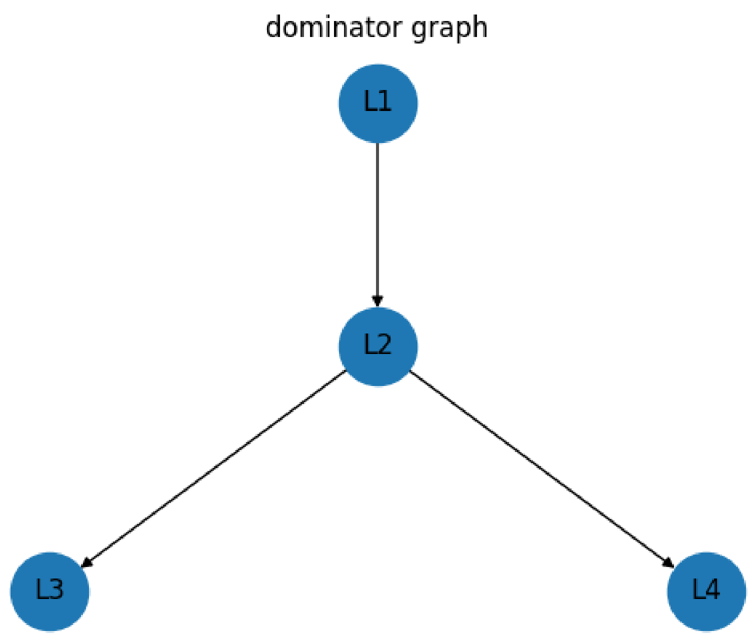


Figure 3: Dominator Graph Example

7 Assembly Code

Once the optimized 3 address code is created, it is passed to our asmcodeshpaer where it will convert the code to assembly code. We use 16 registers and do not optimize the code by conducting live analysis on it and assigning registers to variables. I attempted to implement this but I was unsuccessful. The assembly code should be similar to an x86 assembly.

At this point in our code, a lot of things aren't supported. Strings are no longer allowed and it will only accept the first character of the string. Pointers and memory addresses(variable memory addresses) are ignored and treated as normal varibales. Type modifiers are ignored. There are probably a lot more corner cases but those are the main ignored things.

Optimizations within Assembly:

```
Dividing and Multiplying using shifts if applicable
```

Out 3 address code supports up to 16 registers that are listed below.

```
rax
rbx
rcx
rdx
rsi
rdi
rbp
rsp
r8
r9
r10
r11
r12
r13
r14
r15
```

These are the supported Instructions

```
add
sub
mul
```

```
div
push
pop
call
ret
cmp
jmp
Ret
Leave
xor
shl
shr
```

The shaping of code for follows a somewhat similar structure like so:

```
mov r1, [address1]
mov r2, [address2]
add r1, r2
mov [addressx], r1
```

We will move any variable stored into the registers and perform the necessary operation, then we will save it back out to the memory address that is required. Since I am unable to perform live analysis the moving back into a memory address will always happen unless it is a temporary variable. We remove any temporary variables from our assembly code so it won't show up like that for those.

8 Usage

```
compiler.py [-h] [-t] [-p] [-s] [-a] [-o] [-g] [-d] File
```

8.1 Positional Arguments

File	A valid C input file.

8.2 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.
-a	Outputs the three address code of the input file.
-o	Outputs the optimized three address code of the input file.
-g	Outputs the flow graph of the input file.
-d	Outputs the dominator graph of the input file.

9 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

10 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