CS1319: Programming Language Design and Implementation
Department of Computer Science

Assignment 4: Guide

This guide is to help you get started with Assignment 4. We would recommend going through this before beginning your work. Attached to this guide is a complete three address code generator for the following grammar:

$L \rightarrow LS$
$L \rightarrow S$
$S \rightarrow \texttt{id} = E$
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow E / E$
$E \rightarrow (E)$
$E \rightarrow -E$
$E \rightarrow \texttt{num}$
$E \rightarrow \texttt{id}$

Exercise: Is this grammar ambiguous or unambiguous?

This guide will go over all of the files attached and explain what each of them does. The code of relevant files has been added to this document to make understanding the explanations easier. At the end, you will find the commands to compile this into an executable that accepts another program as an input and gives the corresponding three address code as the output.

`Parser.h` is a header file which contains function definitions and structure definitions which are used in other files. It contains the definition for a symbol table which stores up to 20 different symbols. Each symbol has a corresponding name and value stored at it.

Notice that we use `enum` to map tokens to integers. We declare a variable `opcodeType` which allows us to use numbers in our quads that correspond to these operators. This makes printing them (as defined in the function `print_quad` in the file `Parser.c`) very convenient.

```
#ifndef __PARSER_H
#define __PARSER_H

/* Symbol Table */
typedef struct symtab {
        char *name;
        int value;
} symboltable;

symboltable *symlook(char *);

#define NSYMS 20
extern symboltable symtab[NSYMS];

/* TAC generation support */
symboltable *gentemp();

void emit_binary(
        char *s1,  // Result
        char *s2,  // Arg 1
        char c,    // Operator
        char *s3); // Arg 2
```

```
void emit_unary(
        char *s1, // Result
        char *s2, // Arg 1
        char c);  // Operator

void emit_copy(
        char *s1,  // Result
        char *s2); // Arg 1

/* Support for Lazy TAC generation through Quad array */
typedef enum {
        PLUS = 1,
        MINUS,
        MULT,
        DIV,
        UNARYMINUS,
        COPY,
} opcodeType;

typedef struct quad_tag {
        opcodeType op;
        char *result, *arg1, *arg2;
} quad;

quad *new_quad_binary(opcodeType op1, char *s1, char *s2, char *s3);

quad *new_quad_unary(opcodeType op1, char *s1, char *s2);

void print_quad(quad* q);
#endif // __PARSER_H
```

`Parser.c` contains various functions that are used by the parser. `symlook` is a function that checks whether a symbol with the same name already exists. If it does, then a pointer to that symbol is returned, and if not, that symbol is added to the symbol table.

`gentemp` generates temporary variables needed for three address code generation with a simple counter. It uses the aformentioned `symlook` function to ensure that temporary variables are also added to the symbol table.

The various `emit` functions are used to print three address code directly. Read them carefully and see how they are used to handle different types of operators (unary, binary and assignment). Note that these are used in sections of the parser file which are **commented out**. However, think about how you might use them in your own code if you choose to.

The various `new_quad` functions are used to create quads which are stored in an array. Read them carefully and see how they are used to handle different types of operators (unary, binary and assignment). These are used in the parts of the parser file which actually run. Think about how you might use them in your own code if you choose to.

`print_quad` is a simple function which accepts a quad as input and prints it as readable three address code.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "Parser.h"

extern void yyerror(char *s);
```

```
symboltable *symlook(char *s) {
        symboltable *sp;
        for (sp = symtab; sp < &symtab[NSYMS]; sp++) {
                /* is it already here? */
                if (sp->name &&
                        !strcmp(sp->name, s))
                        return sp;
                if (!sp->name) {
                        /* is it free */
                        sp->name = strdup(s);
                        return sp;
                }
                /* otherwise continue to next */
        }
        yyerror("Too many symbols");
        exit(1); /* cannot continue */
} /* symlook */

/* Generate temporary variable */
symboltable *gentemp() {
        static int c = 0; /* Temp counter */
        char str[10]; /* Temp name */
        /* Generate temp name */
        sprintf(str, "t%02d", c++);
        /* Add temporary to symtab */
        return symlook(str);
}

/* Output 3-address codes */
void emit_binary(
        char *s1, // Result
        char *s2, // Arg 1
        char c,   // Operator
        char *s3) // Arg 2
{
        /* Assignment with Binary operator */
        printf("\t%s = %s %c %s\n", s1, s2, c, s3);
}

void emit_unary(
        char *s1, // Result
        char *s2, // Arg 1
        char c)   // Operator
{
        /* Assignment with Unary operator */
        printf("\t%s = %c %s\n", s1, c, s2);
}

void emit_copy(
        char *s1, // Result
        char *s2) // Arg 1
{
        /* Simple Assignment */
        printf("\t%s = %s\n", s1, s2);
}
```

```c
quad *new_quad_binary(opcodeType op1, char *s1, char *s2, char *s3) {
        quad *q = (quad *)malloc(sizeof(quad));
        q->op = op1;
        q->result = strdup(s1);
        q->arg1 = strdup(s2);
        q->arg2 = strdup(s3);
        return q;
}

quad *new_quad_unary(opcodeType op1, char *s1, char *s2) {
        quad *q = (quad *)malloc(sizeof(quad));
        q->op = op1;
        q->result = strdup(s1);
        q->arg1 = strdup(s2);
        q->arg2 = 0;
        return q;
}

void print_quad(quad* q) {
        if ((q->op <= DIV) && (q->op >= PLUS)) { // Binary Op
                printf("%s = %s ", q->result, q->arg1);
                switch (q->op) {
                        case PLUS: printf("+"); break;
                        case MINUS: printf("-"); break;
                        case MULT: printf("*"); break;
                        case DIV: printf("/"); break;
                }
                printf(" %s\n", q->arg2);
        }
        else
        if (q->op == UNARYMINUS) // Unary Op
                printf("%s = - %s\n", q->result, q->arg1);
        else // Copy
                printf("%s = %s\n", q->result, q->arg1);
}
```

The Flex and Bison files attached are fairly simple and nothing unlike what has been covered in class. They are small and you should go over them once to figure out exactly what they are doing. Note that the Bison file is structured in a way that it first generates all the three address code quads that correspond to the given input, and then at the end prints them using a for-loop:

```c
int main() {
        yyparse();
        for (int i = 0; i < quadPtr; i = i + 1)
                print_quad(qArray[i]);
        return 0;
}
```

You could also choose to use `emit` to print them directly, rather than storing them first. The Bison file also outputs a text file named `Trace.txt` when it is given an input. This prints every single grammar rule that is applied to reduce the input and check for syntactical correctness. In case the grammar is unclear to you, go over this file once you have run the parser on a test input.

Also attached is a short program called `Test.calc`. Make some more test cases and play around with the parser if this guide does not elucidate things adequately.

Lastly, you can use the following commands to compile these files into an executable:

```
bison -d Parser.y
flex -o lex.yy.c Lexer.l
gcc lex.yy.c Parser.tab.c Parser.c -lfl -Werror -o parser
```

This will produce a file called `parser`. We have also added a Makefile so you can alternatively just type `make build`. To test this program using `Test.calc` as an input, run the following command:

```
./parser < Test.calc
```

or alternatively just `make test`.

The output printed to your console should look something like this:

```
t00 = 1
a = t00
t01 = 2
b = t01
t02 = a + b
c = t02
t03 = 3
t04 = c * t03
d = t04
```

Without a good understanding of what this TAC generator does, working on Assignment 4 meaningfully will be extremely difficult, so we encourage you to look through the code as many times as you need to.