CS 445 – Fall 2024

Homework 2

Due on or before Sep 20 at 11:59 PM Pacific Time.

WARNING

Because the output of your program for this assignment will first be processed by an automatic comparison program before being examined by a human being, please follow formatting instructions/examples very carefully. The results your program produces will need to **look exactly like the examples**. Do not embellish with extra titles or other text such as "run complete" or "CS445 output" or even any extra spaces. Points will be deducted for this. (This is realistic. Most companies run test suites on their products and breaking the test suites is not looked upon well in industry.) The testing facility of the submit script will help you refine your software to the point that it is completely correct if you use it as intended.

1. The problem

In the last assignment you built a scanner for C- in flex and a “driver” in bison that printed out information about the tokens produced by the scanner. For this assignment you are to modify the “driver” of the last assignment in order to produce a parser for C-. Your parser will construct the abstract syntax tree (AST) corresponding to the input C- program. Building your parser should consist of the following steps:

1. Build a C- recognizer
2. Extend your C- recognizer to add the AST
3. Add user interface options

Your completed parser must produce the same AST as is specified in the provided examples. If your parser does not produce the same AST then your parser is flawed.

1.1 Building the Recognizer

A recognizer for a particular programming language (PL) simply determines whether or not a given program is a member of the PL that is generated by the grammar for the PL. Use the C- grammar provided on the course website to build a grammar that is suitable for consumption by bison. You will need to fix the dangling-else ambiguity and any other ambiguities or associativity/precedence errors that are currently in the C- grammar. You may not use any bison directives such as %assoc, %left, %right, or %expect in fixing these errors. After you have learned to fix errors “by hand” you can venture into the realm of trying to figure out what the tools are doing to you when they are supposed to be fixing errors for you. But, for now, you are to fix any errors by hand.

Once you have built a bison-ready C- grammar that contains no shift/reduce or reduce/reduce conflicts, you have successfully built a recognizer that needs to be tested. Test your recognizer by passing legal C- files into it and observing your recognizer. If your recognizer doesn’t emit a message indicating that there was an error (e.g. an unrecognized character or missing paren,) your recognizer recognized that

the C- program is legal. If your recognizer emits an error message or behaves in an abnormal way (e.g. SIGSEGV) your grammar is incorrect and needs to be repaired.

Once you have successfully tested your recognizer on all provided C- samples and are confident that it will recognize any legal C- program, it is time to extend the recognizer to build a parser.

1.2.1 Adding the AST

Typically compilers build a modified form of syntax tree known as an abstract syntax tree (AST) instead of building a full parse tree for a given program. An AST is like a distilled form of parse tree that only contains the information that we need to continue the compilation process. Your parser is to construct an AST for any legal C- program. An AST is of course just a tree, and the tree will contain nodes representing information that was gathered about the input C- program. The following is a sample AST node structure that has been previously used in this course:

```
typedef struct treeNode
{
    // connectivity in the tree
    struct treeNode *child[MAXCHILDREN];   // children of the node
    struct treeNode *sibling;              // siblings for the node

    // what kind of node
    int lineno;                            // linenum relevant to this node
    NodeKind nodekind;                     // type of this node
    union                                  // subtype of type
    {
        DeclKind decl;                     // used when DeclK
        StmtKind stmt;                     // used when StmtK
        ExpKind exp;                       // used when ExpK
    } subkind;

    // extra properties about the node depending on type of the node
    union                                  // relevant data to type -> attr
    {
        OpKind op;                         // type of token (same as in bison)
        int value;                         // used when an integer constant or boolean
        unsigned char cvalue               // used when a character
        char *string;                      // used when a string constant
        char *name;                        // used when IdK
    } attr;
    ExpType expType;                       // used when ExpK for type checking
    bool isArray;                          // is this an array
    bool isStatic;                         // is staticly allocated?

    // even more semantic stuff will go here in later assignments.
} TreeNode;
```

This structure was pulled from the Louden textbook and is just a suggestion. You can use whatever you want, as long as you can reliably represent the necessary information in memory and can produce the same results as in the provided examples. The following type definitions are used by the TreeNode structure above, and were also pulled from the Louden textbook:

```
// Kinds of Operators
// these are the token numbers for the operators same as in flex
typedef int OpKind;

// Kinds of Statements
//typedef enum {DeclK, StmtK, ExpK} NodeKind;
enum NodeKind {DeclK, StmtK, ExpK};

// Subkinds of Declarations
enum DeclKind {VarK, FuncK, ParamK};

// Subkinds of Statements
enum  StmtKind {NullK, IfK, WhileK, ForK, CompoundK, ReturnK, BreakK, RangeK};

// Subkinds of Expressions
enum ExpKind {OpK, ConstantK, IdK, AssignK, InitK, CallK};

// ExpType is used for type checking (Void means no type or value, UndefinedType means
undefined)
enum ExpType {Void, Integer, Boolean, Char, CharInt, Equal, UndefinedType};

// What kind of scoping is used?  (decided during typing)
enum VarKind {None, Local, Global, Parameter, LocalStatic};
```

Because this design was borrowed from the Louden textbook, you are free to use the example from the Tiny language compiler in the book (a link for the code for this is on the course website) as a reference to study and build from. You are free to use a C++ class rather than a struct to represent a TreeNode. You can use any structure that you like as long as your C- parser produces the same output as in the provided examples.

Building a correct AST is of critical importance because all future assignments will require successful construction and traversal of the AST corresponding to a given C- program. Any bugs that you produce in the construction and traversal of your AST are going to follow you for the remainder of the semester, so be Very Careful.

To encode a C- program as an AST, you will need to construct the right nodes at the right times during the parsing process. When you need to construct a node, you should use functions that you build like the newStmtNode function in the util.c file for the Tiny language compiler in the book (a link for the code for this is on the course website). These nodes will be passed up the tree as pointers and assembled to build the AST like in the example for the Tiny language in the Louden textbook.

The following is an example to use for specifying the set of types that can represent a yylval in your bison file (this was discussed in class).

```
%union {
    ExpType type;         // For passing types (i.e pass a type in a decl like int or bool)
    TokenData *tokenData; // For terminals. Token data comes from yylex() in the $ vars
    TreeNode * tree;      // For nonterminals. Add these nodes as you build the tree.
}
```

This is just a suggestion of what could be in your %union. Note, however, that you are expressly forbidden from accessing YYSTYPE as is done in the Louden book. Using YYSTYPE directly subverts type checking facilities and is too risky for you in this course.

You should, as mentioned earlier, build functions to construct different types of TreeNode that you want to put in your AST. The following are prototypes that can be used for these functions. If you examine the Tiny compiler code you will see that this is how TreeNode children are added to a given TreeNode in almost all cases (the initialization of a var is one exception). The use of default values for arguments in these prototypes simplifies the construction of a given TreeNode and also simplifies the corresponding code for the action in your bison file.

```
TreeNode *newDeclNode(DeclKind kind,
                      ExpType type,
                      TokenData* token=NULL,
                      TreeNode* c0=NULL,
                      TreeNode* c1=NULL,
                      TreeNode* c2=NULL);

TreeNode *newStmtNode(StmtKind kind,
                      TokenData* token,
                      TreeNode* c0=NULL,
                      TreeNode* c1=NULL,
                      TreeNode* c2=NULL);

TreeNode *newExpNode(ExpKind kind,
                     TokenData* token,
                     TreeNode* c0=NULL,
                     TreeNode* c1=NULL,
                     TreeNode* c2=NULL);
```

Note: Once you have your parser working correctly, make sure that you modify your parser to rename the unary – (minus) operator to the CHSIGN token and rename the unary * (asterisk) operator to the SIZEOF token. There is no need to leave these operators untokenized after we can differentiate between their use in a unary or binary context.

1.2.2 Printing the AST

Once you have successfully constructed an AST for a given program, you will need to print the AST in order to verify that your AST is correct. How do you know that your AST is correct? The structure of a correct AST for a given program is dictated by your bison grammar. You can also compare your printed AST with examples that are provided in the test dataset for this assignment.

To print your AST, you should make a function in your code called **printTree**. Your printTree function must print out the AST precisely as it is printed in the provided examples. Almost the same format is not good enough. The printTree function will be called after the AST has been constructed by your bison-generated parser. Your main() function in your bison file will look something like this:

```
int main(int argc, char* argv[]) {
       // some stuff here…

       // Call yyparse to build the AST and put its address in
       //    the “global” var named syntaxTree of type TreeNode*
       yyparse();
       if (printTreeFlag)
              // Print the contents of the AST to stdout
              printTree(stdout, syntaxTree);

       // other stuff here…
}
```

The `printTreeFlag` variable in the example above is one of the user interface options that you are going to add for this assignment, and is discussed later in this document. The printTree function prints the important information within a given node, and then recursively applies the printTree function to all of the non-NULL children of the node (starting with child[0]), and then recursively applies the printTree function to the sibling pointer if it is not NULL. The first sibling found is numbered 1. Reading the AST provided for given examples is a good way to figure out what to do. For example, given the following C- program as input:

```
// C-F24
int main()
{
    int x;
    int y[5];
    bool b;

    b := true or false and 11 + 22 * 33 / 44 > 55;

    if 666>777 then {
        x := 111;
    }
    else y[3] := x+222;

    while x>999 do {
        x := 333;
        if x<9 then break;
        x := 444;
        break;
   }

   for z := 1 to 8 do if z>x then x := z;
}
```

Your c- (the program containing the C- scanner, parser, and print routines) will scan the input, produce an AST, and print the following output:

```
Func: main  [line: 2]
.   Child: 1  Compound [line: 3]
.   .   Child: 0  Var: x  [line: 4]
.   .   Sibling: 1  Var: y  [line: 5]
.   .   Sibling: 2  Var: b  [line: 6]
.   .   Child: 1  Assign: := [line: 8]
.   .   .   Child: 0  Id: b [line: 8]
.   .   .   Child: 1  Op: or [line: 8]
.   .   .   .   Child: 0  Const true [line: 8]
.   .   .   .   Child: 1  Op: and [line: 8]
.   .   .   .   .   Child: 0  Const false [line: 8]
.   .   .   .   .   Child: 1  Op: > [line: 8]
.   .   .   .   .   .   Child: 0  Op: + [line: 8]
.   .   .   .   .   .   .   Child: 0  Const 11 [line: 8]
.   .   .   .   .   .   .   Child: 1  Op: / [line: 8]
.   .   .   .   .   .   .   .   Child: 0  Op: * [line: 8]
.   .   .   .   .   .   .   .   .   Child: 0  Const 22 [line: 8]
.   .   .   .   .   .   .   .   .   Child: 1  Const 33 [line: 8]
.   .   .   .   .   .   .   .   Child: 1  Const 44 [line: 8]
.   .   .   .   .   .   Child: 1  Const 55 [line: 8]
.   .   Sibling: 1  If [line: 10]
.   .   .   Child: 0  Op: > [line: 10]
.   .   .   .   Child: 0  Const 666 [line: 10]
.   .   .   .   Child: 1  Const 777 [line: 10]
.   .   .   Child: 1  Compound [line: 10]
.   .   .   .   Child: 1  Assign: := [line: 11]
.   .   .   .   .   Child: 0  Id: x [line: 11]
```

```
.   .   .   .   .   Child: 1  Const 111 [line: 11]
.   .   .   Child: 2  Assign: := [line: 13]
.   .   .   .   Child: 0  Op: [ [line: 13]
.   .   .   .   .   Child: 0  Id: y [line: 13]
.   .   .   .   .   Child: 1  Const 3 [line: 13]
.   .   .   .   Child: 1  Op: + [line: 13]
.   .   .   .   .   Child: 0  Id: x [line: 13]
.   .   .   .   .   Child: 1  Const 222 [line: 13]
.   .   Sibling: 2  While [line: 15]
.   .   .   Child: 0  Op: > [line: 15]
.   .   .   .   Child: 0  Id: x [line: 15]
.   .   .   .   Child: 1  Const 999 [line: 15]
.   .   .   Child: 1  Compound [line: 15]
.   .   .   .   Child: 1  Assign: := [line: 16]
.   .   .   .   .   Child: 0  Id: x [line: 16]
.   .   .   .   .   Child: 1  Const 333 [line: 16]
.   .   .   .   Sibling: 1  If [line: 17]
.   .   .   .   .   Child: 0  Op: < [line: 17]
.   .   .   .   .   .   Child: 0  Id: x [line: 17]
.   .   .   .   .   .   Child: 1  Const 9 [line: 17]
.   .   .   .   .   Child: 1  Break [line: 17]
.   .   .   .   Sibling: 2  Assign: := [line: 18]
.   .   .   .   .   Child: 0  Id: x [line: 18]
.   .   .   .   .   Child: 1  Const 444 [line: 18]
.   .   .   .   Sibling: 3  Break [line: 19]
.   .   Sibling: 3  For [line: 22]
.   .   .   Child: 0  Var: z  [line: 22]
.   .   .   Child: 1  Range [line: 22]
.   .   .   .   Child: 0  Const 1 [line: 22]
.   .   .   .   Child: 1  Const 8 [line: 22]
.   .   .   Child: 2  If [line: 22]
.   .   .   .   Child: 0  Op: > [line: 22]
.   .   .   .   .   Child: 0  Id: z [line: 22]
.   .   .   .   .   Child: 1  Id: x [line: 22]
.   .   .   .   Child: 1  Assign: := [line: 22]
.   .   .   .   .   Child: 0  Id: x [line: 22]
.   .   .   .   .   Child: 1  Id: z [line: 22]
```

Note that this output, including the dots to indicate indenting levels, is what your program output is going to be compared against. Your c- must produce this exact same output for this particular input.

When building your AST, if there is an optional expression or statement that is absent, be sure to set the corresponding child pointer to NULL. For example, compound statements can contain declarations, but sometimes they don't. If a compound statement doesn't contain declarations, then the child[0] pointer for that compound statement should be set to NULL. As another example, a return statement in C- may or may not contain an expression. If a given return statement doesn't contain an expression, the child[0] pointer corresponding to the TreeNode for the return statement should be set to NULL. The default value for all pointers, including unnecessary children and siblings, should always be NULL. This means that your code must set these pointers to NULL and consistently check the value of a pointer before attempting to dereference the pointer.

The bison code in the Tiny compiler example source code (on the course website) is a good example of how to connect the TreeNode nodes that you construct. The calculator example on the course website is a good example of how to organize the interaction between the scanner (flex) and the parser (bison).

The following are two examples of functions that can be used to connect sibling pointers and to pass information down a list of siblings. These functions also demonstrate the use of some of the type variables in struct TreeNode.

```
// Adds a TreeNode to a list of siblings.
// Adding a NULL node to the sibling list is probably a programming error!
TreeNode *addSibling(TreeNode *t, TreeNode *s)
{
    if (s == NULL) {
        printf("ERROR(SYSTEM): never add a NULL to a sibling list.\n");
        exit(1);
    }
    if (t != NULL) {
        TreeNode *tmp;

        tmp = t;
        while (tmp->sibling!=NULL) tmp = tmp->sibling;
        tmp->sibling = s;
        return t;
    }
    return s;
}

// Passes the isStatic and type attributes down the sibling list.
void setType(TreeNode *t, ExpType type, bool isStatic)
{
    while (t) {
        t->expType = type;
        t->isStatic = isStatic;
        t = t->sibling;
    }
}
```

1.3 User Interface Options

Once you have successfully built your parser, you are ready to make changes to the interface of your c- program, which is now starting to look like a compiler. Your program must be named c- (note the lowercase) as was the case in the last assignment. Your c- must be able to read the stream of tokens contained within a file whose name is passed as an argument on the command line, or to read the stream of tokens from standard input (stdin) if no argument is passed on the command line. This is the same behavior that was present in the last assignment.

1.3.1 The -p Option

Your c- will *always* construct an AST for a legal C- program. If the user specifies the -p option on the command line, your c- will print the AST that it built to stdout. For example, if the user types the following command:

```
$bash> c- -p mergeSort.c-
```

Your c- will build the AST for the program in mergeSort.c- and will print the AST to stdout. Whereas if the user types the following command:

```
$bash> c- mergeSort.c-
```

Your c- will build the AST for the program in mergeSort.c- but will *not* print the AST to stdout. The -p option will set a variable named `printTreeFlag` (as shown earlier in this assignment) so that you can determine whether or not to print the AST after it has been built. The -p option is the first of many options that we will be adding to the compiler interface during the semester. There is a Unix function called getopt that is used to handle command line arguments that you are free to use. A custom version called ourgetopt is provided on the course website. This custom version is much more portable than standard getopt implementations, and is provided so that you can reap the benefits of getopt while building your compiler on platforms other than Unix.

1.3.2 The -d Option

Your c- will also take the -d option on the command line. This option enables debugging of the parser by setting the yydebug flag in bison to 1. For example if the user enters the following command:

```
$bash> c- -d mergeSort.c-
```

Your -c will build an AST for the program in mergeSort.c- and information about what is happening during the parsing of the file will be printed to stdout. The information that is printed will automatically be printed by the parser generated by bison once you set yydebug to 1. If the user doesn't specify the -d option, no information about what is happening during parsing will be printed. In order to access the yydebug flag that is inside the parser generated by bison, be sure to declare it as an external integer (extern int yyedebug) inside your code.

2. Deliverables and Submission

Your homework will be submitted as an *uncompressed* .tar file that contains no subdirectories. Your .tar file must contain all files and code necessary to build and test your compiler. If you use ourgetopt, *be sure to include the ourgetopt files in your .tar archive***.**

The .tar file is submitted to the class submission page. You can submit as many times as you like. **The last file you submit before the deadline will be the only one graded.** No late submissions are accepted for this assignment. For all submissions you will receive email at your uidaho address showing how your file performed on the pre-grade tests. The grading program will use more extensive tests, so thoroughly test your program with inputs of your own.

Your code must compile successfully and have no shift/reduce or reduce/reduce conflicts from bison. Your program must run with no runtime errors such as segmentation violations (SIGSEGVs).