CS316: Compilers Lab

Programming Assignment 3: Symbol Table, Due: 20/2/2022

1 Introduction

Your goal in this step is to process variable declarations and create a Symbol Table. A symbol table is a data structure that keeps information about non-keyword symbols that appear in source programs. Variable and function names are examples of such symbols. The symbols added to the symbol table will be used in many of the further phases of the compilation. In the previous assignment, your compiler answered whether an input program was syntactically valid or not. As a result, you didn't need token *values* and only the token types were used by the parser generator tools to guide the parsing. A compiler's job, in addition to answering if a program is syntactically valid, is to translate a syntactically valid program. So, in this assignment, your parser needs to get token values such as identifier names and string literals from your scanner. You also need to add semantic actions to create symbol table entries and add those to the symbol table.

2 Background

2.1 Semantic Actions

Semantic actions are steps that your compiler takes as the parser recognizes constructs in your program. Another way to think about this is that semantic actions are code that executes as your compiler matches various parts of the program (constructs like *variable declarations* or tokens like *identifiers*). By taking the right kind of action when the right kind of construct is recognized, you can make your compiler do useful work!

```
STRING foo := "Hello World";
```

Which produces the (partial) parse tree below:

We can create *semantic records* for each of the tokens `IDENTIFIER` and `STRINGLITERAL` that record their values ("foo" and "Hello World", respectively), and "pass them up" the tree so that those records are the records for `id` and `str`. We can then construct a semantic record for `string_decl` using the semantic records of its children to produce a structure that captures the necessary information for a string declaration entry in your symbol table (and even add the entry to the symbol table).

2.1.1 Parser-driven Semantic Actions

It may seem like a lot of work to keep track of pieces of data at each node of the parse tree and assemble them into more complicated records. But most parsers help do this *automatically*. As they build up the parse tree, they will call actions that execute to collect data from the parse tree and create semantic records. In essence, the parsers perform a *post-order* traversal of the parse tree as *they walk the tree*, and store information about the necessary semantic records in a way that you can easily retrieve them. Let's look at how this works for both ANTLR and bison.

For both of these, it is worth remembering two things:

1. Tokens become leaves in the parse tree. The semantic record for a token is either always the text associated with that token (if you're using ANTLR) or whatever you assign `yylval` to in your scanner (if you're using flex/bison).

2. *Every* symbol that shows up in a grammar rule will be a node in your parse tree, and that if you recognize a grammar rule, there will be a node in your parse tree associated with the left-hand-side of the rule and that node will have a separate child for each of the symbols that appear on the right hand side.

2.1.2 ANTLR

In ANTLR, you can put arbitrary Java code in braces (`{}`) at various points in the right hand side of a grammar rule in your `.g4` file; this code will execute as soon as the symbol immediately before the braces has been matched (if it's a token) or predicted and matched (if it's a non-terminal).

The "main" semantic action for a rule will go at the very end of the rule (in other words, it will execute once the entire rule has been matched). As part of this rule, you can assign a value to the semantic record that will be associated with the left hand side of the rule. You name the semantic record and tell ANTLR what type that record should have using a `returns` annotation.

You can then extract information from the semantic records of the children by giving variable names to the children you have matched. You can either access that child's semantic record by accessing `$a.x` where a is the name you gave the child and x is the name you gave to the child's semantic record in the `returns` annotation, or you can access the characters associated with the child (useful when matching tokens) with `$a.text`.

So suppose we wanted to return a structure called `StrEntry` from our `string_decl` rule. It might look something like this:

```
string_decl [returns StrEntry s] :  STRING id ASSIGNOP ex=str SEMICOLON
{$s = new StrEntry(); s.addID($id.text); s.addValue($ex.text);}
```

(Note that if you don't give a child a name (like `id` in the above example), ANTLR defaults to using the name of the token or non-terminal. If the same symbol shows up twice in a rule, you *must* give them names.)

2.1.3 Bison

In Bison, you can put arbitrary C (or C++) code in braces at various points in the right hand side of a grammar rule in your `.y` file. It works very similarly to ANTLR in that respect. The key difference in Bison is understanding how to pass data around.

To set the type of a semantic record for a non-terminal, you use a `%type` command:

```
%type <s_entry> string_decl
%type <s> id  str
```

The above tells that the type associated with the symbol `id` and `str` is `s`. Also, the type associated with the symbol `string_decl` is `s_entry`.

All of the types that you want to use (as part of semantic actions) need to be part of a union that determines the possible types for `yylval`, which you declare in a `%union` declaration:

```
%union{
    StrEntry * s_entry;
    string * s;
}
```

The above tells that `StrEntry` and `string` are the alternatives for type of a semantic record and the type has been given a name—`s_entry` for `StrEntry` and `s` for `string`. So the `%type` commands and `%union`

command in conjunction mean that the `string_decl` non-terminal will produce a semantic record named `s_entry` of type `StrEntry *` and the non-terminals `id` and `str` will produce semantic records of type string `*` named `s`.

(You can also do this by making `yylval` a `struct` instead of a `union`).

Then, when building a semantic action for a rule, `$$` refers to the semantic record you are building for the left hand side (whose type is determined by the `%type` command), and `$1, $2`, etc. refer to the semantic records for the right hand side, listed in order. So here is the equivalent action for creating a `StrEntry` object for the `str_decl` rule:

```
string_decl :  STRING id ASSIGN_OP str SEMI {$$ = new StrEntry(); $$->addID($2); $$->addValue($4);};
```

3 Symbol Tables

Your task in this step of the project is to construct symbol tables for each scope in your program. For each scope, construct a symbol table, then add entries to that symbol table as you see declarations. The declarations you have to handle are integer/float declarations, which should record the name and type of the variable, and string declarations, which should *additionally* record the value of the string. Note that typically function declarations/definitions would result in entries in the symbol table, too, but you do not have to record them for this step.

Nested Symbol Tables In this year's variant of Micro, there are multiple scopes where variables can be declared:

- Variables can be declared before any functions. These are "global" variables, and can be accessed from any function.

- Variables can be declared as part of a function's parameter list. These are "local" to the function, and cannot be accessed by any other function.

- Variables can be declared at the beginning of a function body. These are "local" to the function as well.

- Variables can be declared at the beginning of a then block, an else block, or a FOR statement. These are "local" to the block itself. Other blocks, even in the same function, cannot access these variables.

Note that the scopes in the program are *nested* (function scopes are inside global scopes, and block scopes are nested inside function scopes, or each other). You will have to keep track of this nesting so that when a piece of code uses a variable named "x" you know which scope that variable is from.

To handle this, we suggest you to follow the implementation strategy based on hash-tables discussed in class.

4 What you need to do

You should define the necessary semantic actions and data structures to let you build the symbol table(s) for Micro input programs.

At the end of the parsing phase, you should print out the symbols you found.

For each symbol table in your program, use the following format:

```
Symbol table <scope_name>
name <var_name> type <type_name>
name <var_name> type <type_name> value <string_value>;
...
```

The global scope should be named "GLOBAL", function scopes should be given the same name as the function name, and block scopes should be called "BLOCK X" where X is a counter that increments every time you see a new block scope. *Function parameters should be included as part of the function scope.*

The order of declarations matters! We expect the entries in your symbol table to appear in the same order that they appear in the original program. Keep this in mind as you design the data structures to store your symbol tables.

See the sample outputs for more complete examples of what we are looking for.

CAVEAT: You may be tempted to just print declarations as you see them, rather than building an actual tree of symbol tables. While that will suffice for step 3, it will not be sufficient for later steps of the project. We strongly suggest that you build the necessary data structures since you will have to, eventually, anyway.

Handling errors   Your compiler should output the string DECLARATION ERROR <var_name> if there are two declarations with the same name *in the same scope.*

Sample inputs and outputs are provided to you along with the starter files that you need to get started on this assignment.

5   What you need to submit

- Place all the necessary code for your compiler that you wrote yourself. You do not need to include the ANTLR jar files if you are using ANTLR.

- A Makefile with the following targets:

  1. compiler: this target will build your compiler. If you are using ANTLR, this should create a .jar file.

  2. clean: this target will remove any intermediate files that were created to build the compiler

  3. dev: this target will print the same information that you printed in previous PA.

- A shell script (this must be written in bash) called runme that runs your compiler. This script should take in two arguments: first, the input program file to be compiled and second, the filename where you want to put the compiler. You can assume that we will have run make compiler before running this script.

- You should tag your programming assignment submission as cs316pa3submission

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your Makefile and your runme script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation.