hw4

Start Assignment

Due Nov 29 by 11:59pm **Points** 200 **Submitting** a file upload **File Types** zip

HW4: Code Generation for PL/0

In this homework your group **[Collaborate]** (https://webcourses.ucf.edu/courses/1443631/pages/course-learning-objectives-and-outcomes#OutCollaborate) will implement the remaining parts of a compiler, which consists of the code generator, for the PL/0 language **[UseConcepts]** (https://webcourses.ucf.edu/courses/1443631/pages/course-learning-objectives-and-outcomes#OutUseConcdpts) **[Build]** (https://webcourses.ucf.edu/courses/1443631/pages/course-learning-objectives-and-outcomes#OutBuild) , except that it is not necessary to implement procedures.

The Assignment

(200 points) Using **the provided files** (https://webcourses.ucf.edu/courses/1443631/files/103404752?wrap=1) (https://webcourses.ucf.edu/courses/1443631/files/103404752/download?download_frd=1) , finish the compiler for **the PL/0 language,** (https://webcourses.ucf.edu/courses/1443631/files/103124030?wrap=1) (https://webcourses.ucf.edu/courses/1443631/files/103124030/download?download_frd=1) by implementing code generation for all of PL/0, except for procedure declarations and procedure calls. That is, it is not necessary to implement code generation for call statements or procedure declarations. Because procedure declarations need not be implemented, the running code will not be able to access names declared in surrounding scopes. (This restriction should simplify the compiler.)

Directions

For this homework, we are providing several source code files in the **hw4-tests.zip** (https://webcourses.ucf.edu/courses/1443631/files/103404752?wrap=1) (https://webcourses.ucf.edu/courses/1443631/files/103404752/download?download_frd=1) file, which is available from the hw4 folder in **the files section** (https://webcourses.ucf.edu/courses/1443631/files) here on Webcourses.
Do not change any of these provided files, aside from the Makefile, which you can minimally adjust as needed.

Your compiler will parse PL/0 programs, building abstract syntax trees (ASTs), which will be passed to your compiler's declaration checker, and then to the code generator, as in the provided file `compiler_main.c`. Your code must use these abstract syntax trees (ASTs) and cannot ignore them. The

type AST and the AST types used for each nonterminal are declared in the provided `ast` module (files `ast.h` and `ast.c`). The functions provided by the ast module are as in homework 3.

Implementation in C

For the implementation of the PL/0 compiler, your code must be written in 2017 ANSI standard C and must compile with `gcc` and run correctly on Eustis, when compiled with the `-Wall` and -std=c17 flags. (The provided `Makefile` has one exception, in that it turns off some warnings when compiling `pl0_lexer.c`, since flex seems to generate code that produces a few insignificant warnings. We recommend fixing all warnings from gcc before turning in this assignment.

(See **the course resources pages for details on how to access Eustis (http://www.cs.ucf.edu/~leavens/COP3402/resources.shtml\#course)** .)

SRM Implementation Provided

In the provide **hw4-tests.zip (https://webcourses.ucf.edu/courses/1443631/files/103404752?wrap=1)** **(https://webcourses.ucf.edu/courses/1443631/files/103404752/download?download_frd=1)** file there is an implementation of the Simplified RISC Machine (SRM) in the subdirectory named `vm`, along with its documentation (see the file srm-vm.pdf in the vm subdirectory of the provided files), an assembler, and a disassembler. The disassembler is useful for looking at the assembly language form of the binary object files produced by the compiler. The SRM is basically the same as what you implemented in homework 1, but there are two differences:

1. Tracing is off by default, although you can turn it on by using the -t option on the command line, and the traces include more of the runtime stack than previously,
2. A new instruction, PINT, was added to print an integer value (using the `%d` format from C's `printf` function).

Due to these changes, we recommend using the provided implementation of the SRM, although it would only have these minor differences from the SRM you implemented in homework 1.

The provided files, in essence, include our solutions for homeworks 2 and 3 also. We recommend using these instead of your own lexer and parser and declaration checker for PL/0, but you can substitute yours if you wish.

Using flex and bison

The provided `Makefile` will run flex and bison when building the compiler, although the provided zip file already has the outputs of running flex on `pl0_lexer.l` and bison on `pl0.y`, both of which are provided. Note that the versions of **flex (https://westes.github.io/flex/manual/)** (2.6.4) and **bison (https://www.gnu.org/software/bison/manual/)** (3.8.2) that are installed on Eustis work together well. These also seem to be the versions available on **WSL2 (https://learn.microsoft.com/en-us/windows/wsl/)** (using Ubuntu 22) and in **cygwin (https://cygwin.org/)** on Windows computers. However, the versions of these tools available on an Apple Mac computer are not compatible with each

other, so you may need to run flex and bison on Eustis if you have other versions of these tools on your own computer.

Hints

Don't wait until the last minute to get started on this assignment, as the semester is running out.

See the document **Details on the PL/0 Code Generator (https://webcourses.ucf.edu/courses/1443631/files/103162658?wrap=1)** **(https://webcourses.ucf.edu/courses/1443631/files/103162658/download?download_frd=1)** , which is available in the **Files section of Webcourses (https://webcourses.ucf.edu/courses/1443631/files)** in the hw3 folder for more details and hints on this assignment.

Your task, in essence, is to write C code to implement the gen_code module (in file `gen_code.c`) and to implement the literal_table module (in file `literal_table.c`). We provide header files (`gen_code.h` and `literal_table.h`) that specify the design of these modules and the interface used by the rest of the compiler (see in particular `compiler_main.c`), although the literal_table module is only used by the gen_code module. The gen_code module uses the literal table to track numeric literals used in the program, and to lay these out in the binary object file's data section. For example, if the PL/0 program being compiled uses the number 3402, then the gen_code module would look up 3402 in the literal table, and receive back from the literal table's lookup function (`literal_table_lookup`) an offset to the location in the global data area where 3402 is stored. (As a side effect of this lookup, if 3402 was not previously in the literal table, it is added to the table). The literal table also provides functions to iterate over the entries, so that the literal values can be stored in the binary object file's data section for use by the running program.

The gen_code module has functions for each kind of AST in the program, and the function `gen_code_program` walks the program's AST and uses the BOFFILE it receives to write the program's code into the BOF file, which can then be executed by the SRM. The tree walk would be similar to what you have done for declaration checking in homework 3 and also is like what the unparser's tree walk.

Code Generation Facilities

The provided files include a module named code (provided files `code.h` and `code.c`), which can be used to create individual instructions for the SRM, as each instruction has a corresponding function (for example the SLL instruction can be created by the function `code_sll`). The code module also includes functions that work on code sequences (the type `code_seq`), which are singly-linked lists of code structures.

Provided Attribute Structures: id_attrs and id_use

We provide a module `id_attr`s (files `id_attrs.h` and `id_attrs.c`), which allow access to each declared name's file location (using the provided `file_location` module), kind (constant, variable, or procedure), and an offset count. The `id_use` module is used in the results of queries to the symbol table, and when

uses of identifiers are found during declaration checking, the `id_use` pointer is stored in the AST. Each `id_use` data structure contains the attributes of a name (as defined in `id_attrs.h`) and the number of levels outward from the current potential scope where the declaration of that name was found. The number of levels outward is part of the lexical address of a name, and can be used to find the base of the activation record where the name is stored on the runtime stack. This is not so useful for this assignment, as we are promising not to test your compiler with any inputs containing procedures or procedure calls.

Example: The FLOAT Calculator

A fully worked-out example of a compiler built with the techniques we are discussing, can be accessed through **the example code page for the course (https://www.cs.ucf.edu/~leavens/COP3402/example-code/index.html#FloatCalc)** . This is the FLOAT language's compiler. However, note that there are many differences between **the FLOAT language (https://www.cs.ucf.edu/~leavens/COP3402/example-code/float-calculator/language-definition.pdf)** and PL/0, and FLOAT has multiple data types and compiles to a different VM.

What to Turn In

On webcourses, your group must submit a zip file containing all your source code, your `Makefile`, the output of running our tests (the `*.myo` files) and our tests and expected outputs. This zip file can be created by using the make target submission.zip, that is by running the following command:

```
make submission.zip
```

Code Generation

Criteria	Ratings		Pts
Passing Test Cases Passing the provided test cases	**160 pts** **Partial credit for tests passing** -3 points for each hw4-*test*.pl0 test case that fails on Eustis,	**0 pts** **No test cases pass**	160 pts
Parsing: Compilation on Eustis without warnings The parsing code compiles on Eustis without warnings using the standard options in the provided Makefile (such as running gcc with the options -Wall -std=c17)	**20 pts** **Full Marks**	**0 pts** **No Marks**	20 pts
Parsing: Code readability How easy the code is to follow in the gen_code.c and literal_table.c files.	**20 pts** **Readable code** Code can be followed easily.	**0 pts** **Code that is too hard to follow**	20 pts
			Total Points: 200