Details on the SPL Parser and Declaration Checker
(`$Revision: 1.6 $`)

Gary T. Leavens
Leavens@ucf.edu

October 9, 2024

Abstract

This document gives some details about the parser and declaration checker for the SPL language, and building it with the `bison` parser generator.

1 Introduction

The third homework in COP 3402 is about building a parser and declaration checker for SPL. This document gives some more details about how to do that and some hints.

2 What to Read

You may want to refer to the *Bison 3.8.2* manual [2].

A good explanation of parsing, ASTs, and declaration checking is found in the book *Modern Compiler Implementation in Java* [1], in which we recommend reading chapters 3–5.

You might also want to read *Systems Software: Essential Concepts* [3] in which you might read chapters 5–6.

3 Overview

The SPL language itself is described in the *SPL Manual*, which is available in the Files section of Webcourses and in the homework zip file. The *SPL Manual* defines the grammar of the language and its semantics.

The following subsections specify the interface between the Unix operating system (as found on eustis.eecs.ucf.edu) and the parser as a program.

3.1 Inputs

The parser is passed a single file name as its only command line argument; this file should be the name of a (readable) text file containing the program that the parser should parse and check. Note that this program file is not necessarily legal according to the grammar for SPL, and part of the tasks you have is to produce appropriate error messages for parsing and declaration errors. For example, if the file name argument is `hw3-test1.spl` (and both the compiler executable, `./compiler`, and the `hw3-test1.spl` file are in the shell's current working directory), then the compiler should work on `hw3-test1.spl` and send

all its output to standard output, except for error messages, which are sent to standard error output. The follow command redirects both output streams to the file `hw3-test1.myo`:

```
./compiler hw3-test1.spl >hw3-test1.myo 2>&1
```

The same thing can also be accomplished using the `make` command on Unix as follows:

```
make compiler hw3-test1.myo
```

3.2 Outputs

The compiler prints its normal output, as specified below, to standard output (`stdout`). However, all error messages (e.g., for unreadable files, illegal characters or tokens, parse errors, and declaration errors) should be sent to standard error output (`stderr`). See subsection 4.4 for more details about error messages.

3.3 Exit Code

When the compiler finishes without any errors, it should exit with a zero error code (which indicates success on Unix). However, when the compiler encounters an error it should terminate with a non-zero exit code (which indicates failure on Unix).

4 Output

The output consists of the unparse of the program given as a command line argument, along with any lexical and syntax error messages, and then up to one declaration error message. The provided `unparser` module, specifically the function `unparseProgram`, is called with the abstract syntax tree (AST) produced by the parser, and walks the AST to produce an indented copy of the program (without any comments). (If there are no errors, then the unparsed output could be given as input to the compiler and then the unparsed output would be identical to the input in that case.)

4.1 A Simple Example

Consider the input in the file `hw3-test1.spl`, shown in Figure 1, which is included in the provided zip file `hw3-tests.zip` (which is found in the hw3 folder in the Files section of Webcourses).

```
begin
  print 49; % prints 49
  print 10  % prints 10
end.
```

Figure 1: The test file `hw3-test1.spl`.

This is expected to produce the output found in the Figure 2 (this is in the provided file that is named `hw3-test1.out`).

4.2 Provided Driver

We provide a driver (which is in the provided file `compiler_main.c`) to run the tests. Use the error message functions in the `utilities` module to achieve proper error message output formatting.

```
begin
  print 49;
  print 10
end
.
```

Figure 2: Expected output (on stdout) from running the compiler on `hw3-test1.spl`, which produced the file shown (which is `hw3-test1.out`).

Tests are found in the files (contained in the zip file) named `hw3-*.spl`. The expected output of each test is found in a file named the same as the test input but with the suffix `.out`. For example, the expected output of the SPL file `hw3-test1.spl` is in the file `hw3-test1.out`.

You can check your own parser by running the tests using the Unix command on eustis.eecs.ucf.edu:

```
make check-nondecl-outputs
```

Running the above command will generate files with the suffix `.myo`; for example your output from test `hw3-test3.spl` will be put into `hw3-test3.myo`.

4.3 Do Not Change the Provided Files

You must not change any of the provided `.h` or `.c` in your submission, although you are allowed to change them during your program's development (for example, to add extra debugging information), but any such changes must be reverted before your submission.

You must use bison and the provided files (including the ASTs and the unparser) in your program.

4.4 Errors that Must be Detected

Your code must detect the following errors (in addition to the lexical errors detected in the previous homework):

Syntax errors, whenever an input does not follow the concrete syntax of our subset of SPL as defined by the *SPL Manual*. The error message must state which tokens were expected (possible) and the file location that is noted in the error message should be the location of the unexpected token. Use the provided functions in the `utilities` module (`utilities.h` and `utilities.c`) to produce such error messages.

Undeclared identifier errors, whenever an identifier is used in a statement (including any conditions and expressions within it) that does not have a surrounding visible declaration. The error message must give the identifier being used and its location in the input file.

Duplicate declaration errors, whenever an identifier is declared more than once in a potential scope. The error message must state what the identifier was being declared as, the name of the identifier, and what kind of declaration (constant, variable, or procedure) was the previous declaration.

Error messages sent to `stderr` should start with a file name, followed by a line number. Use the provided functions in the `utilities` module (`utilities.h` and `utilities.c`) to produce such error messages.

There are examples of programs with syntax (i.e., parse) errors in the provided files with names of the form `hw3-errtest*.spl` and `hw3-parseerrtest*.spl`, where `*` is replaced by a number (or

letter). The expected output of each test is found in a file named the same as the test input but with the suffix `.out`. For example, the expected output for the test file named `hw3-parseerrtest3.spl` is in the file `hw3-parseerrtest3.out`.

There are also examples of programs with declaration errors in `hw3-declerrtest*.spl`, where `*` is replaced by a number (or letter). The expected output of each test is found in a file named the same as the test input but with the suffix `.out`; for example, the expected output of `hw3-declerrtest3.spl` is in the file `hw3-declerrtest3.out`.

4.5 Checking Your Work

You can check your own compiler by running the tests using the following `make` command on the machine eustis.eecs.ucf.edu, which uses the provided `Makefile` from the `hw3-tests.zip` file.

```
make check-outputs
```

Running the above command will generate files with the suffix `.myo`; for example your output from test `hw2-errtest3.spl` will be put into `hw2-errtest3.myo`.

4.5.1 Checking the Parser

Use the target `check-nondecl-outputs` in the following command to check only the provided parsing tests (using the provided unparser).

```
make check-nondecl-outputs
```

4.5.2 Checking the Declaration Checker

Use the target `check-decl-outputs` in the following command to check only the provided declaration tests.

```
make check-decl-outputs
```

You can also use the `check-outputs` target to check your declaration checker on all programs (but note that it may not make sense to bother with the inputs that have syntax or lexical syntax errors).

A Hints

The versions of flex (2.6.4) [4] and bison (3.8.2) [2] that are installed on eustis.eecs.ucf.edu do work together for this assignment. These also seem to be the versions available on WSL2 (using Ubuntu 22) and in cygwin on Windows computers. However, it seems that the versions of these tools available on an Apple Mac computer are *not* compatible with each other, so you will need to use eustis.eecs.ucf.edu to run flex and bison for this homework.

As noted in the assignment on Webcourses, it is sensible to start by implementing the parser and having the parser's actions build the ASTs, as the declaration checker will need the ASTs to work on and they are easiest to produce in the parser.

You need to write the following files:

`spl_lexer.l`, which you can reuse from homework 2. If your homework 2 solution does not work, and it is after the due date for homework 2, then the course staff can provide you with a solution that works; contact the course staff if you need that.

`spl.y`, which is the specification of the parser using the bison tool [2]. You can start by creating that file by using the make target `start-bison-file`, which will give you an empty `spl.y` file containing a top part that will connect to the lexer (generated by flex from your `spl_lexer.l` file).

A symbol table. We recommend that you write a module that is a stack of scope data structures (in a modules named `symtab`, perhaps) and scope data structure (in a module named `scope`, perhaps) that is instantiated for each potential scope that maps declared names to their attributes).

A declaration checking tree walk. This is a module (perhaps named `scope_check`) that walks the AST of the program, recording declarations and checking for undefined identifier uses and duplicate declarations.

We are providing several files for this homework, all of which are in the `hw3-tests.zip` file in the Files section's hw3 folder on Webcourses. These files include the following.

- A `Makefile` with targets `check-outputs` and `submission.zip` among others.
- A `compiler_main.c` file, which provides a driver for the compiler.
- The `unparser` module (in files `unparser.h` and `unparser.c`), which pretty prints an AST to a given output file. (This module is part of our testing infrastructure.)
- The `ast` module (in files `ast.h` and `ast.c`), which defines types for the various ASTs and functions to help build them.
- `bison_spl_y_top.y` which provides the top section of the `spl.y` file you need to write, and is used to initialize that file, with the make target `start-bison-file`. You would then edit the file `spl.y` using that as a starting point.
- A header file `parser_types.h` that helps connect the parser and the lexical analyzer by defining the type `YYSTYPE` to be the type `AST` defined in the `ast` module.
- The `utilities` module, in files `utilities.h` and `utilities.c`, which provide error message functions.
- The `file_location` module (in files `file_location.h` and `file_location.c`), which provide a type (`file_location`) that stores a file name and line number.
- A `lexer` module (in files `lexer.h` and `lexer.c`) which provides an interface to the lexical analysis routines in `spl_lexer.l`'s user code section. (This is the same as the `lexer` module from homework 2.)
- A `parser` module (in files `parser.h` and `parser.c`), which provides an interface to the parser produced by bison (in the files `spl.tab.h` and `spl.tab.c`).
- An `id_attrs` module (in files `id_attrs.h` and `id_attrs.c`), which defines attributes useful in declaration checking and some helping functions for them.
- An `id_use` module (in files `id_use.h` and `id_use.c`), which provides a way to group the attributes of a declaration together with information needed to create lexical addresses. We recommend using (pointers to) the `id_use` type as the result type for functions that query the symbol table.
- A `machine_types` module (from which you will need the file `machine_types.h` for this homework) that defines some types specific to the Simplified RISC Machine and SPL.

References

[1] Andrew Appel and Jens Palsberg. *Modern Compiler Implementation in Java: Second Edition*. Cambridge, 2002.

[2] Free Software Foundation. Bison 3.8.1. `https://www.gnu.org/software/bison/manual/`, Sep 2021.

[3] Euripides Montagne. *Systems Software: Essential Concepts*. Cognella Academic Publishing, 2021.

[4] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex, for flex 2.6.2. `https://westes.github.io/flex/manual/index.html`, 2012.