Homework 3: Parser and Declaration Checker for PL/0

See Webcourses and the syllabus for due dates.

1 Purpose

In this homework your team [Collaborate] will implement a parser for the context-free syntax of the PL/0 language [UseConcepts] [Build], and also a declaration checker for the PL/0 language without procedure declarations and procedure calls. This subset of PL/0 is defined in Section 7 below.

2 Directions

For this homework, we are providing several files in the `hw3-tests.zip` file in the course homeworks directory. Do not change any of these provided files, as they are used in our testing.

Your compiler will both produce and use abstract syntax trees (ASTs). Your compiler's parser will produce ASTs that will be tested by the unparser we provide and also passed to your compiler's declaration checker. Thus, your code must use the abstract syntax trees (ASTs) whose type is declared in the provided file `ast.h`. Your code should also use the functions provided in the file `ast.c` to aid in building ASTs. In addition, your code must use the error message facilities provided in `utilities.h` and `utilities.c` as well as the unparser provided in the files `unparser.c` and `unparser.h` (and also a private header file `unparserInternal.h`). The error messages and unparsing are part of our testing, and the unparser expects that the ASTs conform to the declarations given in the provided file `ast.h`.

For the implementation, your code must be written in 2017 ANSI standard C and must compile with gcc and run correctly on Eustis, when compiled with the `-std=c17` flag.[1] We recommend using the gcc flags `-std=c17 -Wall` and fixing all warnings before turning in this assignment.

You must also use your lexer from homework 2 and the tokens it generates in your parser (for this we provide all the files we provided in homework 2, including `token.h`). (If you do not have a working lexer, talk to the course staff.)

Note that we will randomly ask questions of students in the team to ensure that all team members understand their solution; there will be penalty of up to 10 points (deducted from all team members' scores for that assignment) if some team member does not understand some part of the solution to an assignment.

For this homework, you are *not* allowed to submit code generated by a parser generator (such as `yacc` or `ANTLR`).

1. (150 points) Implement and submit your code for a PL/0 parser that checks for syntax errors and build ASTs, together with the output of our tests named `hw3-asttest*.pl0` (where `*` is a digit or letter) and `hw3-parseerrtest*.pl0`. Your group must implement a recursive-descent parser and have it build the abstract syntax trees (ASTs) declared in the provided file `ast.h`; for this you may use the functions provided in the file `ast.c` (see Section 7.2 below). Our tests for this part require your code to use the unparser provided in `unparser.c`, `unparser.h`, and `unparserInternal.h` to print a textual representation of these ASTs.

[1] See this course's resources page for information on how to access Eustis.

Have your `main` function call a function in the parser to open the file provided on the command line (we suggest naming it `parser_open` and putting it in a file named `parser.c`); that function should call your `lexer_open` function. Then call a function to parse the program and return its AST. (We suggest naming the function that parses a program something like `parseProgram`.) During parsing your parser should report any syntax errors using the function `parse_error_unexpected`, which is declared in the provided file `utilities.h` (and implemented in `utilities.c`). After parsing, if there were no parse errors, then your `main` function must call our provided function `unparseProgram`, and pass it the `FILE* stdout` and the AST returned by the parser for the program.

Note that the provided tests named `hw3-astttest*.pl0` are syntactically legal and should not have parse errors (or declaration errors). However, the tests named `hw3-parseerrtest*.pl0` contain one or more parse errors.

2. (100 points) Implement and submit your code for the declaration checker (which can be submitted along with the parser of the previous problem) as well as the output of our tests in the files named `hw3-asttest*.pl0` and `hw3-declerrtest*.pl0`. For full credit, your declaration checker must work by traversing the ASTs returned by the parser (and declared in the provided file `ast.h`). (If your declaration checker works in some other way, then we will deduct 40 points from your group's score for this part.) That is, your `main` function must pass the AST returned by your parser to a function that walks the AST, builds a symbol table during its walk over the constant and variable declarations ASTs, while noting any duplicate declarations, and must produce an error report for all uses of identifiers (in statements and expressions) that have not been declared in the program. Use the function `parse_error_general`, declared in the provided file `utilities.h` (and implemented in `utilities.c`) for error reporting, as that is assumed in our testing.

3 What to Turn In

Your team must submit on Webcourses a single zip file containing solutions to both problems. (That is, you only need to submit one zip file in total for this homework, not one for each problem.) This zip file must include:

1. A plain text file, called `sources.txt`, that names all the `.c` source files used to implement your compiler. (The file names in `sources.txt` should be separated by either blanks or newlines.) The files needed for your lexer should also be included. For example, your `sources.txt` file might look like the following:

   ```
   ast.c token.c reserved.c lexer.c file_location.c id_attrs.c
   parser.c unparser.c utilities.c
   scope_symtab.c scope_check.c compiler_main.c
   ```

   (The order of these names should not matter if you include the header files in each `.c` file that declare all the names used in that `.c` file.)

2. Each source file that is needed to compile both your compiler and VM with `gcc -std=c17` on Eustis, including all needed header files.

3. The output files that result from running our tests. These are the `.myo` files created by the provided `Makefile`.

You can use the Unix command

```
make submission.zip
```

on Eustis to create a zip file that has all these files in it, after you have created your `sources.txt` file and run our tests (using the command `make check-outputs`) to create the `.myo` files.

We will take points off for not passing the tests and some points off for: code that does not work properly, duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. Avoid duplicating code by using helping functions, or library functions. It is a good idea to check your code for these problems before submitting.

Don't hesitate to contact the staff if you are stuck at some point. Your code should compile properly; if it doesn't, then you probably should keep working on it. Email the staff with your code file if you need help getting it to compile or have trouble understanding error messages. If you don't have time to get your code to compile, at least tell us that you didn't get it to compile in your submission.

4 What to Read

You may want to read *Systems Software: Essential Concepts* (by Montagne) in which we recommend reading chapter 6 (pages 103-117).

5 Overview

The following subsections specify the interface between the Unix operating system (as on Eustis) and the parser as a program.

5.1 Inputs

Your compiler will be passed a single file name as its only command line argument. This file name is the name of a file that contains the input PL/0 program to be compiled. Note that this input program file is not necessarily legal according to the grammar for the subset of PL/0 you are to implement. For example, if the file name argument is `hw3-asttest1.pl0` (and both the compiler executable, `./compiler`, and the file `hw3-asttest1.pl0` are in the current directory), then the following command line (given to the shell on Eustis)

```
./compiler hw3-asttest1.pl0 > hw3-asttest1.myo 2>&1
```

will run your compiler on the program in `hw3-asttest1.pl0` and put the unparsed AST and any error messages into the file `hw3-asttest1.myo`.

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

```
make hw3-asttest1.myo
```

5.2 Outputs

The output of the unparser, which is a textual display of the AST for the program, must be sent to standard output (`stdout`). All error messages must be sent to standard error output (`stderr`). See subsection 7.9 for more details about error messages.

5.3 Exit Code

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

6 What Must be Done

Your compiler (i.e., its `main` function) must make the following happen:

1. Run a function to initialize the parser to work on the given input file; this will involve (eventually) calling `lexer_open` on the input file name that is passed to your program.
2. Parse the program (using tokens obtained by calling `lexer_next` and generate an AST for it, checking for parse errors. The parser will return a pointer to an AST; in what follows we suppose this is called `progast`.
3. Close the input file (this should involve calling `lexer_close`).
4. Run our provided unparser by calling the function `unparseProgram` with the arguments `stdout` and the program's AST (`progast`).
5. Perform any required initialization on your compiler's symbol table (we suggest calling a function to initialize it, which might involve initializing any static variables in your symbol table module).
6. Using the AST (`progast`), build the symbol table and check the AST for identifiers that are declared more than once or uses of identifiers for which there is no corresponding declaration, issuing an error message for such problems.
7. Return a success exit code (we assume that any errors have already caused the program to exit with a failure exit code).

(Steps 5 and 6 are needed for declaration checking; if your compiler is not (yet) doing declaration checking, then these steps could be omitted.)

Using the error message facilities in the provided utilities module (composed of the files `utilities.h` and `utilities.c`) will cause your compiler to exit with a failure exit code when it issues its first error message (but see Section 8 below).

7 PL/0 Subset

The language you will be compiling for this homework is a subset of PL/0 without procedures or the **`call`** statement.

Since this subset of PL/0 does not contain the reserved words "`procedure`" or "`call`," these can be treated as identifiers by your lexical analyzer. However, at your option, you may leave these as reserved words for future use (in a later homework). (Our tests will not use these strings of characters.)

⟨program⟩ ::= ⟨block⟩ **.**

⟨block⟩ ::= ⟨const-decls⟩ ⟨var-decls⟩ ⟨stmt⟩

⟨const-decls⟩ ::= **{**⟨const-decl⟩**}**
⟨const-decl⟩ ::= **`const`** ⟨const-def⟩ **{**⟨comma-const-def⟩**}** **;**
⟨const-def⟩ ::= ⟨ident⟩ = ⟨number⟩
⟨comma-const-def⟩ ::= **,** ⟨const-def⟩

⟨var-decls⟩ ::= **{**⟨var-decl⟩**}**
⟨var-decl⟩ ::= **`var`** ⟨idents⟩ **;**
⟨idents⟩ ::= ⟨ident⟩ **{**⟨comma-ident⟩**}**
⟨comma-ident⟩ ::= **,** ⟨ident⟩

⟨stmt⟩ ::= ⟨ident⟩ **:=** ⟨expr⟩
| **`begin`** ⟨stmt⟩ **{**⟨semi-stmt⟩**}** **`end`**
| **`if`** ⟨condition⟩ **`then`** ⟨stmt⟩ **`else`** ⟨stmt⟩
| **`while`** ⟨condition⟩ **`do`** ⟨stmt⟩
| **`read`** ⟨ident⟩
| **`write`** ⟨expr⟩
| **`skip`**
⟨semi-stmt⟩ ::= **;** ⟨stmt⟩
⟨empty⟩ ::=

⟨condition⟩ ::= **`odd`** ⟨expr⟩
| ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨rel-op⟩ ::= = | <> | < | <= | > | >=

⟨expr⟩ ::= ⟨term⟩ **{**⟨add-sub-term⟩**}**
⟨add-sub-term⟩ ::= ⟨add-sub⟩ ⟨term⟩
⟨add-sub⟩ ::= ⟨plus⟩ | ⟨minus⟩
⟨term⟩ ::= ⟨factor⟩ **{**⟨mult-div-factor⟩**}**
⟨mult-div-factor⟩ ::= ⟨mult-div⟩ ⟨factor⟩
⟨mult-div⟩ ::= ⟨mult⟩ | ⟨div⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨sign⟩ ⟨number⟩ | (⟨expr⟩)
⟨sign⟩ ::= ⟨plus⟩ | ⟨minus⟩ | ⟨empty⟩

Figure 1: Context-free grammar for the concrete syntax of this homework's subset of PL/0. The grammar uses a `terminal font` for terminal symbols, and a **`bold terminal font`** for reserved words. As in EBNF, curly brackets $\{x\}$ means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Also note that an **`else`** clause is required in each if-statement.

⟨ident⟩ ::= ⟨letter⟩ **{**⟨letter-or-digit⟩**}**
⟨letter⟩ ::= `a` | `b` | `...` | `y` | `z` | `A` | `B` | `...` | `Y` | `Z`
⟨number⟩ ::= ⟨digit⟩ **{**⟨digit⟩**}**
⟨digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨letter-or-digit⟩ ::= ⟨letter⟩ | ⟨digit⟩
⟨plus⟩ ::= `+`
⟨minus⟩ ::= `-`
⟨mult⟩ ::= `*`
⟨div⟩ ::= `/`

⟨ignored⟩ ::= ⟨blank⟩ | ⟨tab⟩ | ⟨vt⟩ | ⟨formfeed⟩ | ⟨eol⟩ | ⟨comment⟩
⟨blank⟩ ::= "A space character (ASCII 32)"
⟨tab⟩ ::= "A horizontal tab character (ASCII 9)"
⟨vt⟩ ::= "A vertical tab character (ASCII 11)"
⟨formfeed⟩ ::= "A formfeed character (ASCII 12)"
⟨newline⟩ ::= "A newline character (ASCII 10)"
⟨cr⟩ ::= "A carriage return character (ASCII 13)"
⟨eol⟩ ::= ⟨newline⟩ | ⟨cr⟩ ⟨newline⟩
⟨comment⟩ ::= ⟨pound-sign⟩ **{**⟨non-nl⟩**}** ⟨newline⟩
⟨pound-sign⟩ ::= #
⟨non-nl⟩ ::= "Any character except a newline"

Figure 2: Lexical grammar of PL/0. The grammar uses a `terminal font` for terminal symbols. Note that all ASCII letters (a-z and A-Z) are included in the production for ⟨letter⟩. Again, curly brackets **{**x**}** means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Some character classes are described in English, these are described in a Roman font between double quotation marks (" and "). Note that all characters matched by the nonterminal ⟨ignored⟩ are ignored by the lexer (except that each instance of ⟨eol⟩ ends a line).

7.1 Syntax

The context-free grammar for this subset of PL/0 is defined in Figure 1 and its lexical grammar is defined in Figure 2.

Note that the lexical grammar is unchanged from the prior homework in which you implemented a lexer. However, you need not treat "`procedure`" or "`call`" as reserved words, although you may do that if you want to use the lexer without change from your previous homework.

7.2 ASTs

The type for abstract syntax trees (ASTs) is defined in the provided files `ast.h`, with helping functions in `ast.c`.

The file `ast.h` declares a type named `AST` and a type `AST_list`. The type `AST_list` is a (linked) list of ASTs.

7.2.1 The AST Type

An `AST` is a C **`struct`** containing the following fields:

- A field named `file_loc` that gives the AST's (starting) file location. That is, it gives information about the place in a PL/0 source file where (the start of) the first token that was parsed into the AST was found: its filename, line number, and column number.

  The file location has the type `file_location`, which is a type defined in `file_location.h`, which is a provided file. The file location is used in error messages.

- A field named `type_tag` that indicates what kind of AST is held in the `data` field (see the next item below).

  These type tags are members of the enumerated type `AST_type`, which is declared in the provided file `ast.h`. Each of these tags corresponds to a nonterminal symbol in the abstract syntax of PL/0 (see Figure 4). For example, `program_ast` corresponds to the nonterminal ⟨program⟩ and `const_decl_ast` corresponds to a list of ⟨const-decl⟩. Each of these may be the tag of a node in the AST returned by the parser, with the exception of `op_expr_ast`, which is only intended to be used to return intermediate results from parsing an ⟨add-sub-term⟩ or ⟨mult-div-factor⟩ in the concrete syntax (see Figure 1).

- A field named `data` that holds the struct type corresponding to the type tag (see the previous item above). This field is a C union, meaning that it can hold any one of the declared struct types corresponding to the tags of the union, which can be thought of as subfields.

  For example, when the `type_tag` field is `program_ast`, then the union holds a `program_t` struct named `program`. Thus if `progast` has the type `AST *`, and if `progast->type_tag` is `program_ast`, then `progast->data.program` is a struct of type `program_t` with fields `cds`, `vds`, and `stmt` that can be accessed by the expressions `progast->data.program.cds`, `progast->data.program.vds`, and `progast->data.program.stmt`, respectively. The type of `progast->data.program.cds` will be `AST_list`, and each element of that list will be an AST with `type_tag` field `const_decl_ast`. Suppose that your compiler has a variable of type `AST *` named `cd`, that holds such a constant declaration AST (i.e., where `cd->type_tag` is `const_decl_ast`). Then the data field will hold a `const_decl_t` with fields `name` and `num_val` that can be accessed by the expressions `cd->data.const_decl.name` and `cd->data.const_decl.num_val`, respectively.

The combination of the type tag and the union make the AST into what is called a "tagged, discriminated union," where the type tag always describes the contents of the union part. The correspondence is given in Figure 3.

A good example of how to write a tree walk over the ASTs is given in the provided file `unparser.c`.

7.2.2 AST Lists

In some ASTs some of the fields hold lists of other AST. In the provide `ast.h` file there is a type `AST_list` that describes such lists and several operations that work on such lists.

The ASTs that hold lists, are as follows.

- A program AST (with type tag `program_ast`) contains two lists: a list of constant declaration ASTs (in the field `cds`) and a list of variable declaration ASTs (in the field `vds`).

type tag (AST_type)	corresponding (sub)field	corresponding struct type
program_ast	program	program_t
const_decl_ast	const_decl	const_decl_t
var_decl_ast	var_decl	var_decl_t
assign_ast	assign_stmt	assign_t
begin_ast	begin_stmt	begin_t
if_ast	if_stmt	if_t
while_ast	while_stmt	while_t
read_ast	read_stmt	read_t
write_ast	write_stmt	write_t
skip_ast	skip_stmt	skip_t
odd_cond_ast	odd_cond	odd_cond_t
bin_cond_ast	bin_cond	bin_cond_t
op_expr_ast	op_expr	op_expr_t
bin_expr_ast	bin_expr	bin_expr_t
ident_ast	ident	ident_t
number_ast	number	number_t

Figure 3: Correspondence between type tags, union tags (names of subfields of `data`), and the type of the struct that appears in the union when the type tag is as is given. See the provided file `ast.h`.

⟨program⟩ ::= **{**⟨const-decl⟩**}** **{**⟨var-decl⟩**}** ⟨stmt⟩
⟨const-decl⟩ ::= **const** ⟨ident⟩ = ⟨number⟩
⟨var-decl⟩ ::= **var** ⟨ident⟩
⟨stmt⟩ ::= ⟨ident⟩ **:=** ⟨expr⟩
| **begin** ⟨stmt⟩ **{**⟨stmt⟩**}**
| **if** ⟨condition⟩ **then** ⟨stmt⟩ **else** ⟨stmt⟩
| **while** ⟨condition⟩ **do** ⟨stmt⟩
| **read** ⟨ident⟩
| **write** ⟨expr⟩
| **skip**
⟨condition⟩ ::= **odd** ⟨expr⟩ | ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨rel-op⟩ ::= = | <> | < | <= | > | >=
⟨expr⟩ ::= ⟨expr⟩ ⟨bin-arith-op⟩ ⟨expr⟩ | ⟨ident⟩ | ⟨number⟩
⟨bin-arith-op⟩ **::=** + | − | * | /

⟨op-expr⟩ **::=** ⟨bin-arith-op⟩ ⟨expr⟩

Figure 4: Abstract Syntax for PL/0. Here the curly brackets, as in **{**x**}**, means a possibly empty list of x. Note that ⟨op-expr⟩ is intended only as an intermediate form during parsing, and is not used directly by any of the other rules.

- A begin statement AST (with type tag `begin_stmt`), contains a list of statement ASTs in the field `stmts`.

The ast module (in the provided files `ast.h` and `ast.c`) defines the following functions that work on lists of (pointers to) ASTs:

```
/* $Id: ast.h,v 1.13 2023/02/21 03:32:51 leavens Exp leavens $ */
// Return an AST list that is empty
extern AST_list ast_list_empty_list();

// Return an AST list consisting of just the given AST node (ast)
extern AST_list ast_list_singleton(AST *ast);

// Return true just when lst is an empty list (and false otherwise)
extern bool ast_list_is_empty(AST_list lst);

// Requires: !ast_list_is_empty(lst)
// Return the first element in an AST_list
extern AST *ast_list_first(AST_list lst);

// Requires: !ast_list_is_empty(lst)
// Return the rest of the AST_list (which is null if it is empty)
extern AST_list ast_list_rest(AST_list lst);

// Requires: !ast_list_is_empty(lst) and ast_list_is_empty(ast_list_rest(lst))
// Make newtail the tail of the AST_list starting at lst
extern void ast_list_splice(AST_list lst, AST_list newtail);

// Return the last element in the AST list lst.
// The result is only NULL if ast_list_is_empty(lst);
extern AST_list ast_list_last_elem(AST_list lst);
```

7.3 Creating ASTs

The ASTs are similar to what would be the desired parse trees for the abstract syntax shown in Figure 4. The abstract syntax is related to the concrete syntax (shown in Figure 1) by omitting intermediate levels (for example, ⟨block⟩, ⟨term⟩, and ⟨factor⟩), punctuation (e.g., semicolons and commas), and reserved words.

For example a program AST consists of a list of ASTs for the constant declarations, a list of ASTs for the variable declarations, and an AST for the program's statement. In the list of ASTs for the constant and variable declarations, there is one item (i.e., one AST) in the list per declaration, so the (the grouping present in the original program is ignored. To be specific the following PL/0 program (in our subset)

```
const a = 1, b = 2;
const c = 3;
var x, y;
var z;
skip.
```

will be represented by a program AST that contains: a constant declaration list consisting of 3 constant declaration ASTs (one each for `a`, `b`, and `c`, each of which will have the type tag `const_decl_ast`), a variable declaration list consisting of 3 variable declaration ASTs (one each for `x`, `y`, and `z`, each of which will have the type tag `var_decl_ast`), and a statement AST (with type tag `skip_ast`).

The ASTs for statements have distinct type tags that depend on the particular kind of statement (see Figure 3). For example while an if-statement and a while-statement are both kinds of statements, their ASTs have different type tags (`if_ast` and `while_ast`, respectively).

The functions declared in `ast.h` (and implemented in `ast.c`) can be used to create different kinds of ASTs with the corresponding type tags.

7.4 Semantics

This subsection describes the semantics of PL/0. For this homework, you are not implementing this semantics, except to issue error messages for duplicate declarations of an identifier or uses of an identifier that has not been declared.

Nonterminals discussed in this subsection refer to the nonterminals in the context-free grammar defined in Figure 1.

A ⟨program⟩ consists of zero-or-more constant declarations (⟨const-decls⟩), zero-or-more variable declarations (⟨var-decls⟩), followed by a statement.

All constants and variables are (short) integers. The execution of a program declares the named constants and variables, and initializes the constants and variables. Then it runs the statement.

It is an error if an ⟨ident⟩ is declared more than once, as either a constant or a variable.

7.4.1 Constant Declarations

The ⟨const-decls⟩ specify zero or more constant declarations.

Each constant declaration, of the form ⟨ident⟩ = ⟨number⟩, declares that `ident` is a (short) integer constant that is initialized to the value given by ⟨number⟩. The scope of such a constant declaration is the area of the program's text that follows the declaration.

It is an error for an ⟨ident⟩ to be declared as a constant more than once. It is an error for the program to refer to the ⟨ident⟩ on the left hand side of an assignment statement.

7.4.2 Variable Declarations

The ⟨var-decls⟩ specify zero or more variable declarations.

Each variable declaration, of the form ⟨ident⟩, declares that ⟨ident⟩ is a (short) integer variable that is initialized to the value 0. The scope of such a variable declaration is the area of the program's text that follows the declaration.

It is an error for an ⟨ident⟩ to be declared as a variable if it has already been declared as a constant or as a variable.

Unlike constants, variable names may appear on the left hand side of an assignment statement.

7.4.3 Statements

A program contains a single statement that is run when the program starts executing.

Assignment Statement An assignment statement has the form ⟨ident⟩ `:=` ⟨expr⟩. It evaluates the expression ⟨expr⟩ to obtain a value and then it assigns it to the variable named by ⟨ident⟩. Thus, immediately after the execution of this statement, the value of the variable ⟨ident⟩ is the value of ⟨expr⟩.

It is an error if the left hand side ⟨ident⟩ has not been declared as a variable.

Begin Statement A begin statement has the form **begin** $S_1; S_2; \ldots; S_n$ **end** (where $n \geq 1$) and is executed by first executing statement S_1, then if S_1 finishes without encountering an error S_2 is executed, and so on, in sequence.

Conditional Statement A conditional statement has the form **if** C **then** S_1 **else** S_2 and is executed by first evaluating the condition C. When C evaluates to true, then S_1 is executed; otherwise, if C evaluates to false (i.e., if it does not encounter an error), then S_2 is executed.

Note that there are no parentheses around the condition.

While Statement A while statement has the form **while** C **do** S and is executed by first evaluating the condition C. If C evaluates to false, then S is not executed and the while statement finishes its execution. When C evaluates to true, then S is executed, followed by the execution of **while** C **do** S again. Note that C is evaluated each time, not just once.

Read Statement A read statement of the form **read** x, where x is a declared variable identifier, reads a single character from standard input and puts its ASCII value into the variable x. The value of x will be set to -1 if an end-of-file or an error is encountered on standard input.

It is an error if x has not been previously declared as a variable.

Write Statement A write statement of the form **write** e, first evaluates the expression e, and if that expression yields a value in the range 0 to 255, then it writes that value to standard output as an ASCII character. Otherwise, if e yields a value outside the range 0 to 255 (i.e., a value less than 0 or greater than 255), then an error occurs.

Skip Statement A skip statement of the form **skip** does nothing and does not change the program's state.

7.4.4 Conditions

A ⟨condition⟩ is an expression that has a Boolean value: either true or false.

Odd Condition A ⟨condition⟩ of the form **odd** e first evaluates the expression e. If the value of e is an odd integer (i.e., it is equal to 1 modulo 2), then the value of the condition is true. If the value of e is even, then the value of the condition is false.

Relational Conditions A ⟨condition⟩ of the form $e_1\ r\ e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the condition as a whole encounters that error.) Then it compares v_1 to v_2 according to the relational operator r, as follows:

- if r is =, then the condition's value is true when v_1 is equal to v_2, and false otherwise.
- if r is <>, then the condition's value is true when v_1 is not equal to v_2, and false when they are equal.
- if r is <, then the condition's value is true when v_1 is strictly less than v_2, and false otherwise.
- if r is <=, then the condition's value is true when v_1 is less than or equal to v_2, and false when $v_1 > v_2$.
- if r is >, then the condition's value is true when v_1 is strictly greater than v_2, and false otherwise.

- if r is >=, then the condition's value is true when v_1 is greater than or equal to v_2, and false when $v_1 < v_2$.

7.5 Expressions

An ⟨expr⟩ of the form $e_1\ o\ e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the expression as a whole encounters that error.) Then it combines v_1 and v_2 according to the operator o, as follows:

- An expression of the form e_1+e_2 (i.e., a binary operator expression where the operator o is +) yields the value of $v_1 + v_2$, according to the semantics of the type **`short int`** in C.
- An expression of the form e_1−e_2 yields the value of $v_1 - v_2$, according to the semantics of the type **`short int`** in C.
- An expression of the form $e_1 \star e_2$ yields the value of $v_1 \times v_2$, according to the semantics of the type **`short int`** in C.
- An expression of the form e_1 / e_2 yields the value of v_1/v_2, according to the semantics of the type **`short int`** in C. The expression encounters an error if v_2 is zero.

There are also a few other cases of expressions that do not involve binary operators. These have the following semantics:

- An identifier expression, of the form x, has as its value the value of the integer stored in the constant or variable named x.

  It is an error if x has not been previously declared as a constant or variable.
- An expression of the form sn, where s is a ⟨sign⟩ and n is a ⟨number⟩ yields the value of the base 10 literal n if the sign s is + or ⟨empty⟩. However, if the sign s is −, then the value is the negated value of the base 10 literal n according to the semantics of the type **`short int`** in C.

  Note that there is no AST for negating a number, since the AST can hold the negation; thus the negated value is simply stored as a number AST.
- An expression of the form `(`e`)` yields the value of the expression e.

7.6 Simple Examples of Inputs and Outputs

7.6.1 Inputs without Errors

Consider the following input in the file `hw3-asttest3.pl0`, (note that the suffix is lowercase 'P', lowercase 'L', and the numeral zero, i.e., '0') which is included in the `hw3-tests.zip` file in the course homeworks directory.

```
# $Id: hw3-asttest3.pl0,v 1.1 2023/02/19 20:12:22 leavens Exp $
var x, y;
x := y.
```

This should produce the expected output found in the following file (`hw3-asttest3.out`).

```
var x;
var y;
x := y
.
```

In this case the input has no syntax errors and no declaration errors, so the output is from the unparser, which uses the AST built by the compiler's parser.

7.7 Inputs with Parse Errors

Consider the following input in the file `hw3-parseerrtest2.pl0`, which is included in the `hw3-tests.zip` file in the course homeworks directory.

```
# $Id: hw3-parseerrtest2.pl0,v 1.1 2023/02/19 20:12:22 leavens Exp $
begin
  write 49;
  .
end.
```

This should produce the expected output (on stderr), which is found in the following file (`hw3-parseerrtest2.out`).

```
hw3-parseerrtest2.pl0: line 4, column 3: syntax error, Expecting one of: identsym, beginsyr
```

In this case the input has a syntax error, as indicated by the error message, and the compiler exits with a failure exit code when it issues an error message; thus no AST is ever given to the unparser, so the error message is the only output.

7.8 Inputs with Declaration Errors

Undeclared Identifiers Consider the following input in the file `hw3-declerrtest0.pl0`, which is included in the `hw3-tests.zip` file in the course homeworks directory.

```
# $Id: hw3-declerrtest0.pl0,v 1.1 2023/02/19 20:12:22 leavens Exp $
x := 0.
```

This should produce the expected output which is found in the following file (`hw3-declerrtest0.out`).

```
x := 0
.
hw3-declerrtest0.pl0: line 2, column 1: identifer "x" is not declared!
```

In this case the input has a declaration error: as indicated by the error message there is no declaration for the identifier "`x`," that is used in the statement. Since the parser finished without detecting any syntax errors, the AST is actually created and so the output shows the unparsed AST (which was sent to stdout) followed by the error message (which was sent to stderr).

Multiple Declarations Consider the following input in the provided file `hw3-declerrtest1.pl0`.

```
# $Id: hw3-declerrtest1.pl0,v 1.1 2023/02/19 20:12:22 leavens Exp $
var x, y, x;
x := 7.
```

This should produce the expected output, which is found in the following file (`hw3-declerrtest1.out`).

```
var x;
var y;
var x;
x := 7
.
hw3-declerrtest1.pl0: line 2, column 11: variable "x" is already declared as a variable
```

In this case the identifier "x," is declared twice, as indicated by the error message. Since the parser finished without detecting any syntax errors, the AST is actually created and so the output shows the unparsed AST (which was sent to stdout) followed by the error message (which was sent to stderr).

7.9 Errors that Must be Detected

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

1. Syntax errors, whenever an input does not follow the concrete syntax of our subset of PL/0 as defined by Figure 1 (and Figure 2). 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.

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

3. Declaration errors, whenever an identifier is used in a statement (including any conditions and expressions within it) that has not been previously declared. The error message must give the identifier being used and its (starting) location in the input file.

Error messages sent to `stderr` should start with a file name, a colon, a space and the line and column numbers, followed by a space. Use the provided function lexical_error (found 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 `hw3-parseerrtest*.pl0`, 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.pl0` is in the file `hw3-parseerrtest3.out`.

There are also examples of programs with declaration errors in `hw3-declerrtest*.pl0`, 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.pl0` is in the file `hw3-declerrtest3.out`.

7.10 Checking Your Work

You can check your own compiler by running the tests using the following Unix shell command on Eustis, which uses the `Makefile` from the `hw3-tests.zip` file in the course homeworks directory.

```
make check-outputs
```

Running the above command will generate files with the suffix `.myo`; for example your output from test `hw3-errtest3.pl0` will be put into `hw3-errtest3.myo`.

8 Error Recovery

While not part of this homework assignment, you might be interested in exploring how to improve the errors in the compiler. However, you should only investigate this issue once you have gotten your compiler working perfectly.

The basic issue is that the compiler stops whenever it encounters an error, only issuing one error message. There are several ways in which this could be improved, including:

lexical error recovery In lexical error recovery, the lexer keeps on working after issuing an error message. The easiest way to do this is to discard illegal characters (after issuing an error message), and to truncate identifiers and numeric literals that are too large (after issuing an error message). Reaching the end-of-file during a comment could simply terminate the comment and return an end-of-file token to the parser, after issuing an error message.

syntactic error recovery In syntactic error recovery, the parser keeps on working after issuing an error message. In a recursive-descent parser of the sort you are implementing, one can make special cases for certain errors (e.g., assuming that a missing semicolon is present). However, to prevent infinite loops in the parser, it is best not to insert tokens, but to delete tokens, e.g., when parsing a statement, to delete tokens up to the next semicolon or **`end`**. See the survey article by Hammond and Rayward-Smith [1] for more information and references.

A Hints

We will give more hints in the class's lecture and lab sections.

Recursion is your friend in this assignment. Write code trusting that the functions you call work properly and concentrate on understanding what each function is responsible for doing (e.g., which tokens it consumes and what kind of ASTs it creates).

Note that we are providing (in the `hw3-tests.zip` file in the course homeworks directory) both a declaration of the relevant types involved in ASTs (in `ast.h` and file_location .h) and several functions to create and return (pointers to dynamically allocated) ASTs.

Use the functions declared in the provided ast module (`ast.h` and `ast.c`) to create ASTs and manipulate the lists of ASTs involved.

For a good example of how to do a tree walk on the ASTs (e.g., to build a symbol and check declarations and identifier uses), see the provided file `unparser.c`.

To check declarations, you will need a symbol table. A symbol table is a mapping from identifiers (i.e., strings) to attributes. You can use the provided id_attrs module (files id_attrs .h and id_attrs .c) for the attributes.

You are allowed to use a fixed-sized array for your symbol table, as long as it holds at least 4K identifiers and their associated attributes.

You do not need to make use of the lexer_output module (in the provided files lexer_output .h and lexer_output .c) in this homework, although that may be helpful for debugging the (lexical) syntax of your own tests. Nevertheless, the sources for the lexer_output module are provided, in case your lexer uses them.

References

[1] K. Hammond and V. Rayward-Smith. A survey on syntactic error recovery and repair. *Computer Languages*, 9(1):51–67, 1984.