University of Central Florida
Department of Computer Science
COP 3402: System Software
Fall 2022

Homework #3 Parser- Code Generator

This is a solo or team project

Assignment Overview:

In this assignment, you will implement a Recursive Descent Parser and Code Generator for PL/0. **Parsers** analyze a program's input to verify that it is syntactically correct according to the language's grammar, report errors if present, and if not, populate a symbol table with the input's procedures, constants, and variables. **Code Generators** use the symbol table created by the parser to translate the program into assembly language for the virtual machine to execute. We provide instruction on an approach that interleaves the parser and code generation functionality: emitting code and populating the symbol table in the same pass through the input program.

You will implement the function

*instruction *parse(int code_flag, int table_flag, lexeme *list);*

within the project file **parser.c**.

The lexeme array, **list**, holds the entirety of the input in token format. It is the very same array that your work in HW2 returned if there were no lexical errors. Just as in HW2, the token_type enum definition will be useful to you when identifying an index's type. Because the driver will only call *parse* if there were no errors in the lexical analyzer, the **list** will never contain errors. Functionally, the parser has no way of knowing how many entries of the **list** array are populated (remember that all arrays are at most ARRAY_SIZE), but practically, this is not necessary information. Error recognition should prevent the program from looping infinitely, if implemented correctly.

Your parser will analyze the input program for syntactical correctness. The PL/0 language is defined by an EBNF grammar (see next section). When the input doesn't obey the grammar, it is syntactically incorrect. We give you a list of errors (Appendix B) which define all the ways an input may violate the grammar. If the program is incorrect, your parser will print the error messages. If the program was syntactically correct, the parser will

print print out the Symbol Table if **table_flag** is true (1) and the Assembly Code if **code_flag** is true (1) before returning the instruction array.

The array of instructions, which we will refer to as the code array, should look familiar. It is the input you worked with in HW1. The instruction struct is defined in compiler.h, and a copy is included here:

```
typedef struct instruction {
     int op;
     int l;
     int m;
} instruction;
```

The instruction types your parser is tasked with emitting (adding to the code array) are defined in Appendices A and B of HW1. We assume it's obvious why you would emit a particular instruction at some point in the grammar (ie if you have a read statement, you'll need a RED instruction), but when it's not obvious we try to include an explanation of why it's present. There is a second enum in compiler.h, opcode_name, which defines all the three letter opcode names with their numerical vales (ex. DIV = 4). You can use this when emitting instructions. When we specify that an instruction should be emitted, but we don't specify what the L or M values should be, it is usually safe to assume that they are 0. For example, L is zero for all instructions except LOD, STO, and CL. For OPR instructions, we just say to emit ADD, etc and expect you to know that ADD is the M field and you should use OPR for the op field.

<u>The Grammar:</u>
Based on Wirth's definition for EBNF we have the following rules:
[ ] means an optional item.
{ } means repeat 0 or more times.
Terminal symbols are enclosed in quote marks. Identifiers and Numbers should be treated as terminal symbols as well.
A period is used to indicate the end of the definition of a syntactic class.

program ::= block "**.**" **.**
block ::= declarations statement**.**
declarations ::= { const | var | proc }.
const ::= ["**const**" ident "**:=**" [“**-**“] number "**;**"]**.**
var ::= ["**var** "ident “**;**"]**.**
proc ::= ["**procedure**" ident "**;**"].
statement ::= [ident "**:=**" expression
| "**call**" ident
| "**begin**" statement { “**;**” statement } "**end**"
| "**if**" condition "**then**" statement

| "**while**" condition "**do**" statement
| "**read**" ident
| "**write**" expression
| “**def**” ident “**{**” block “**}**”
| “**return**”] **.**

condition ::= expression rel-op expression**.**
rel-op ::= “==“|“**!**="|"<"|"<="|">"|">=“**.**
expression ::= term { ("+"|"-") term}**.**
term ::= factor {("*"|"/") factor}**.**
factor ::= ident | number | "**(**" expression "**)**“ .

Error Handling

When your program encounters an error, it should print out an error message. There are some errors the parser can recognize and move past (non-stopping), and others which will halt the program completely upon discovery (stopping). Appendix B details all the errors and whether the program should continue past them.

We do not want you to use system calls or the exit function to achieve this. If you do, your submission will lose points. Instead, you should use an error flag. Create a global integer variable and initialize it to zero. When a function finds an error, it should call the *print_parser_error* function with the error code and set the global variable to 1 if the error is non-stopping, -1 if the error is stopping. If the error is stopping, the function should return. Stopping errors should cause the flow of control to backtrack to *parser* so the program can end. After a function calls another function (ex. whenever statement calls expression), you should check the error flag. If it is -1, then a stopping error was found and the function should return. Finally in *parser*, after the final error check, if the error flag is still zero (meaning there were no errors), call the functions to print the assembly code and symbol table if the corresponding flags are true, then return the code array.

Implementing the Parser and Code Generator:

Variables:

All the global variables you need have already been defined for you in the *parser_skeleton* file. Any additional variables you use should be local to the functions you use them in, as this program is recursive and recursive calls will overwrite the values if you make them global. You will need to use all the global variables we do give you. The integer *error* is explained in the Error Handling section above. The other integer *level* holds the current lexographical level at your current place in the input program. Additionally, there are three arrays and three corresponding indexing variables:

*lexeme *tokens* and *token_index* - *tokens* does not need to be defined, assign the input argument **list** for this variable

*symbol *table* and *table_index* - *table* will store the information for all the procedures, variables, and constants in the program; you should use ARRAY_SIZE and calloc for it's definition; *table_index* always points to the next empty space in the table

*instruction *code* and *code_index* - *code* will store the instructions and will eventually be returned if there were no errors; again you should use ARRAY_SIZE and calloc for it's definition; *code_index* always points to the next empty space in the code

It is essential that you do not alter the names of these variables, as they are referenced within the support functions we are giving you.

Functions:

You must implement a function for every non-terminal character defined in the grammar, with two exceptions. Program is the only non-recursive non-terminal character, it is only called once when the parser begins processing, as such it is not strictly necessary for program to be its own function and you can combine it with the *parser* function. The second exception is rel-op which acts more as a category of tokens than as a grammar structure.

We provide you with three printing functions and five support functions:

- *void print_assembly_code();* - call at the end before returning if there were no errors and input argument *code_flag* is true
- *void print_symbol_table();* - call at the end before returning if there were no errors and input argument *table_flag* is true
- *void print_parser_error(int error_code, int case_code);* - call when you find an error to print the error message; as you'll see in Appendix B, some errors have multiple cases; if an error doesn't have cases, the value of case_code doesn't matter
- *void emit(int op, int l, int m);* - adds an instruction to the code array and increments *code_index*
- *void add_symbol(int kind, char name[], int value, int level, int address);* - adds a symbol to the code array and increments *table_index*
- *void mark();* - called at the end of block; marks all the symbols that belong to the active procedure so other procedures know they are inaccessible
- *int multiple_declaration_check(char name[]);* - used in declaration functions to check if a potential symbol name has already been used by another symbol in the active procedure

- *int find_symbol(char name[], int kind)*; - used in statement and factor to find the symbol being referenced; prioritizes symbols belonging to the active procedure of symbols that belong to its parent procedures

You shouldn't need to implement any additional support functions, but you won't be penalized if you do choose to do so.

Annotated Grammar:

Below is an annotated version of the grammar, augmented to mark where all the errors are found (in **red**) and where all the instructions are emitted (in **green**). [] means optional, () means choose of the options, {} means repeat zero or more times, non-terminal characters are capitalized, terminal characters are in **blue**

```
PROGRAM ::= BLOCK (.|1) {21} [HLT]
BLOCK ::= DECLARATIONS STATEMENT
DECLARATIONS ::= {CONST|VAR|PROC} INC
CONST ::= [const (ident [3] | 2-1) (:=|4-1) [-] (number|5)
          (;|6-1)]
VAR ::= [var (ident [3] | 2-2) (;|6-2)]
PROC ::= [procedure (ident [3] | 2-3) (;|6-3)]
STATEMENT ::= [ ident [8-1|7] (:=|4-2) EXPRESSION STO
              | call (ident [8-2|9] | 2-4) CAL
              | begin STATEMENT {; STATEMENT} (end|6-4|10)
              | if CONDITION JPC (then|11) STATEMENT
              | while CONDITION JPC (do|12) STATEMENT JMP
              | read (ident [8-3|13] | 2-5) RED STO
              | write EXPRESSION WRT
              | def (ident [8-4|14|22|23] | 2-6) ({|15) JMP
                  BLOCK [RTN] (}|16)
              | return [HLT|RTN] ]
CONDITION ::= EXPRESSION (==|!=|<|<=|>|>=|17) EXPRESSION
              (EQL|NEQ|LSS|LEQ|GTR|GEQ|OPR)
EXPRESSION ::= TERM {(+|-) TERM (ADD|SUB)}
TERM ::= FACTOR {(*|/) FACTOR (MUL|DIV)}
FACTOR ::= ident [8-5|18] (LOD|LIT)
         | number LIT
         | ( EXPRESSION ()|19)
         | 20
```

To implement this, work through the grammar using the token list, starting with the actions of PROGRAM which you can implement in *parse*. Process the tokens of the program by following the path of the grammar. When you come to a non-terminal, process it by calling the corresponding function. When you come to a terminal character, process it by

incrementing token_index. When you come to an error, process it by calling the print_parser_error function and determine whether to stop (see Appendix B for more error instructions). Remember not to increment the token_index if there was an error. When you come to an instruction, process it by calling the emit function. Brackets and parentheses generally mean you'll need a if-else-if structure. Curly braces mean you'll need a while loop. See Appendix A for additional notes on how to implement the annotated grammar.

Each non-terminal symbol from the grammar should be it's own function (except for **program** which you can put in main). The appendices indicate where and how each error can be found and indicate whether the error is stopping, non-stopping, or sometimes stopping. For the errors marked sometimes stopping in the pseudocode, you will need to reference the error list in Appendix B for the follow set.

Testing Your Program:

As explained in the project overview of HW1, your assignment is one part of a multi-file C program. We understand that this is the first time many of you a re working with a multi-file C project. To test your program, you need to upload all of the project files to Eustis3:

- driver.c - the driver program; no change from HW1
- compiler.h - the header file; no change from HW1
- vm.o and lex.o - compiled versions of our implementation of vm.c and lex.c. Please know that both have been updated for bug fixes and you'll need to download the new versions. Since this assignment is about parser.c, we don't want you to worry about your previous work. We will use vm.o and lex.o for grading.
- parser.c - where you'll write your code
- Any input and output files you want to use.

To compile, use the command "`gcc driver.c lex.o parser.c vm.o`"

To execute, use the command "`./a.out input.txt -c -s`"

- Replace input.txt with the name of whatever input file you're using
- "`-c`" is a tag for the driver so your program knows to print the assembly code output. This sets code_flag to true
- "`-s`" is a tag for the driver so your program knows to print the symbol table output. This sets table_flag to true

To compare your output to correct output, use the command "`./a.out input.txt -c -s > output.txt`" to generate your output. Then compare your output to correct output with the command "`diff -w -B your_output.txt correct_output.txt`" This will print out any differences between the two files. If the command doesn't print anything, then the two files are exactly the same (the desired outcome).

Making Your Own Test Cases:

We're providing you with several test cases to use when developing your program, but we will be using different test cases to grade, so you may want to write your own test cases. Input files are written according to the grammar. We will not be including test cases with lexical errors, because when they occur the parser is not called. To get the correct output for your test cases, we're providing the "magic" file. "magic" is a compiled version of our implementation of the project. It works like a.out. You must be on Eustis3 to run it. You may get an error with permissions, use the command "`chmod +x magic`" if this happens. To get the parser output for your input program, use the command "`./magic input.txt -c -s`". If your input had lexical errors, "magic" will print them and this will prevent it from calling the parser. To write the output to a file, add "`> output.txt`" to the end of the command. Make sure that your test cases have at most 499 characters, any more will cause a segfault because the maximum array size is 500.

Debugging Tip:

In the version you submit for a grade, the symbol table and assembly code should only be printed if there were no errors in the program. However, while you're debugging your code it can be really helpful to print the symbol table and assembly code for error cases. It can help you find where in your program your bug is hiding. You can simply call the print functions. They will print out from the beginning of their respective arrays to the current value of their respective indexing variable. We've also included the file "magic_debug" which prints out the symbol table and assembly code if their respective driver tags are used, even if there were errors in the input.

Administrative Guidelines:

1. The parser/code generator must be written in C and must run on Eustis3. If it runs on your PC but not on Eustis3, for us it does not run. If you need help setting up your computer to access Eustis3, reach out to a TA or Dr. Aedo.
2. Do not change compiler.h or driver.c.
3. Do not change the grammar.
4. Do not try to resubmit vm.c or lex.c
5. Do not add a main function to parser.c
6. Include comments in your program.
7. If you submit a program from another semester or section or from the internet, this is considered plagiarism. We regard this as cheating. At a minimum, you will receive a zero on this assignment.
8. Submit to Webcourses:
    a) The source code of your parser, which should be named "parser.c"
    b) Student names should be written in the header comment of the source code file and in the comments of the submission.

Rubric:

5	Compiles
5	Produces lines of meaningful output before segfaulting or looping infinitely
36	Every error/error case is recognized in the correct location
22	Errors/error cases that should stop do so
15	Grammar Structures are processed correctly (the correct instructions are emitted in the correct order, even if they have incorrect L or M values)
4	Correct instructions are chosen for complex grammar structures explicit vs implicit returns, LOD vs LIT in factor
5	Symbol Table
8	Complex instructions have the correct L and M values CAL, STO, JPC, JMP

Appendix A: Implementing the Grammar

A few notes:

- You must add a symbol to the table if there was not an error in its declaration. If there was a non-stopping error, you don't have to add a symbol to the table but we find it helpful in debugging to do so. We provide alternative field values when there are errors.
- You must add the instructions to the code if there were no errors, but if there was a non-stopping error, you don't have to add the instructions. However, we find it helpful in debugging to do so. We provide alternative field values when there are errors.

```
PROGRAM ::= BLOCK (.|1) {21} [HLT]
```

- You can implement PROGRAM within the *parse* function or have it separately as its own function. This is because PROGRAM is never called by another non-terminal.
- Before calling BLOCK, you must initialize level to -1. This is because BLOCK is called once for every procedure, it increments level at the beginning and decrements at the end, and this initial call is for the "main" procedure of the input program whose level is 0.
- For CAL instructions, the M field should be the index in the code array where the procedure starts (aka its address field in the symbol table). However, when we emit CAL instructions (in STATEMENT), the procedure may not have been defined yet (its address field is -1). So instead, when we emit CAL instructions, we use the index of the procedure in the symbol table for the M field. This way, at the very end once all the procedures are defined and we know their addresses, we can replace the M fields with the proper values.
    1. Find every CAL instruction in the code array
    2. If the M field is -1, this means there was some error with identifier and we don't/can't correct this, move to the next one
    3. Otherwise the current M field is the index of the desired procedure in the symbol table. Check for error 21. If there is an address set, set the M field of the instruction to it.
- The HLT emit is optional here because it's possible that the last statement was an explicit return. We know this has happened if the last instruction in code is a HLT instruction. If this is not the case, the HLT is implicit and so you need to emit it in PROGRAM.

```
BLOCK ::= DECLARATIONS STATEMENT
```

- level should be incremented before DECLARATIONS is called so that it reflects the current lexographical level in the input program

- After STATEMENT is called, the symbols belonging to the current procedure must be marked so the parent procedures can't use them. This is done by calling support function *mark*
- Lastly, level should be decremented at the end so it reflects that the procedure is done being processed.

```
DECLARATIONS ::= {CONST|VAR|PROC} INC
```

- We emit INC in DECLARATIONS because every procedure (including main) begins with an INC instruction to create space on the stack for the activation record. It must happen before BLOCK calls STATEMENT so that it's the first instruction of the procedure, and it needs to happen after all the symbol declarations because the M value of the INC instruction is dependent on the number of variables declared. We choose to emit INC in DECLARATIONS rather than in BLOCK, because this way we don't need to return a value to BLOCK.
- You'll need to keep track of how many variables have been declared. Start at 0 and increment after each call to VAR.
- The {} means you'll need a while loop. Because each of the declaration functions is optional, they should only be called if the current token matches the first token of the declaration statement (ie only call CONST if tokens[token_index].type == *keyword_const*). You only loop while the current token matches one of the three first tokens (*keyword_const, keyword_var, keyword_procedure*).
- Because the address field of variable symbols should match their position within the activation record, the VAR function should be passed the number of variables declared.
- The M field of the INC instruction is equal to the number of variables declared + 3 (the activation record size). INC is always emitted, even if there weren't any symbols at all.

```
CONST ::= [const (ident [3] | 2-1) (:=|4-1) [-] (number|5)
          (;|6-1)]
```

- If error 2-1 is non-stopping, you can use "null" for the symbol name.
- Since the *minus* is optional, there's no error if it's missing. If there is a *minus*, you'll need to multiply the *number* value before adding the symbol to the table.
- If error 5 is non-stopping, you can use 0 for the symbol value.
- You must use the global level for the level of the symbol.
- Constants do not have addresses, use 0 for this field.

```
VAR ::= [var (ident [3] | 2-2) (;|6-2)]
```

- As mentioned in DECLARATIONS, this function needs to take an argument: the number of variables that were declared before the current one.
- If error 2-2 is non-stopping, you can use "null" for the symbol name.

- You must use the global level for the level of the symbol.
- Variables don't have values, use 0 for this field.
- The address of the symbol is the function's argument + 3 so it reflects the position of the variable in the AR of the procedure on the stack.

```
PROC ::= [procedure (ident [3] | 2-3) (;|6-3)]
```

- If error 2-3 is non-stopping, you can use "null" for the symbol name.
- You must use the global level for the level of the symbol.
- Procedures don't have values, use 0 for this field.
- Because we want to be able to know if a procedure has been defined yet and all procedures go into the symbol table undefined, you must set the address field to -1.

There are several different statement types. Luckily they all have a unique first token. You can structure the STATEMENT function with if-else-if or a switch case based on the current token. Since STATEMENT is nullable (because it's wrapped in optional brackets), the final else or default case should just be a return. There's no error if the current token does not match one of the tokens in STATEMENT's first set. I'll section each statement type on its own.

```
ident [8-1|7] (:=|4-2) EXPRESSION STO
```

- We refer to this statement type as an assignment statement.
- Use the support function find_symbol to get the index of the desired symbol in the symbol table. You'll need to save it for when you emit the STO instruction. If find_symbol returns -1, you'll have an error. See Appendix B for more details.
- If you had error 8-1 or error 7, you should use -1 for both the L and M fields of the STO instruction.
- If you did not have an error, the L value is global level minus the level field of the symbol from the table. The M value is the address field of the symbol from the table.

```
call (ident [8-2|9] | 2-4) CAL
```

- If you have a non-stopping error, you can use -1 for both the L and M fields of the CAL instruction.
- Use the support function find_symbol to get the index of the desired symbol in the symbol table. You'll need to save it for when you emit the CAL instruction. If find_symbol returns -1, you'll have an error. See Appendix B for more details.
- If you did not have an error, the L value is global level minus the level field of the symbol from the table. The M value is the index of the symbol in the table (the value returned by find_symbol). The reason we use this value instead of the address field is because it's possible that the procedure hasn't been defined yet and the address field will still be -1. We go back in PROGRAM/parse to correct this.

```
begin STATEMENT {; STATEMENT} (end|6-4|10)
```

- We recommend a do-while loop. Within the loop, increment token_index before calling STATEMENT. Loop while the current token is a semicolon.

```
if CONDITION JPC (then|11) STATEMENT
```

- For if statements, we jump if the result of the condition operation was false. Thus we use JPC. The JPC must be emitted before the true action is processed (before calling STATEMENT), but because we won't know how long the true action is until after we've processed it, we need to save the location of the JPC instruction in the code array so we can go in later and fix the M value. When you emit the instruction, use 0 for the M value. You'll know where to jump to after you call STATEMENT: the index of the next instruction in the code array.

```
while CONDITION JPC (do|12) STATEMENT JMP
```

- For while statements, we jump if the result of the condition operation was false. Thus we use JPC. The JPC must be emitted before the true action is processed (before calling STATEMENT), but because we won't know how long the true action is until after we've processed it, we need to save the location of the JPC instruction in the code array so we can go in later and fix the M value. When you emit the instruction, use 0 for the M value.
- Because we're looping and we process the condition at the beginning of every iteration, the last instruction of the true action must be a JMP to the index of the first instruction of the condition calculation. We emit this after our call to STATEMENT. The M value should be the index of the first instruction of the condition calculation. Thus we'll need to save this information before we call condition so we'll have it when we emit JMP
- We fix the JPC once we know where the last instruction of the true action is. Thus after we emit the JMP, we can fix the JPC's M value so it's the index of the next instruction in code.

```
read (ident [8-3|13] | 2-5) RED STO
```

- Use the support function find_symbol to get the index of the desired symbol in the symbol table. You'll need to save it for when you emit the STO instruction. If find_symbol returns -1, you'll have an error. See Appendix B for more details.
- If you had an error, you should use -1 for both the L and M fields of the STO instruction.
- If you did not have an error, the L value is global level minus the level field of the symbol from the table. The M value is the address field of the symbol from the table.

```
write EXPRESSION WRT
```

- This one is as basic as it gets. Increment token_index after *keyword_write*, call EXPRESSION, return if there was a stopping error, emit WRT.

```
def (ident [8-4|14|23|22] | 2-6)({|15) JMP BLOCK [RTN] (}|16)
```

- If you have error 2-6 and it's non-stopping, use a flag to remember not to try to save the procedure address later on.
- Once we call BLOCK, we'll start emitting code for the procedure.
- Before we do that, we need to emit a JMP so we don't try to execute the definition of the procedure unless we call it. Since we don't know how much code we'll emit once we call BLOCK, save the index of the JMP in the code array so we can correct the M value later.
- Additionally, now we'll know where the first instruction of the procedure is in the code array AKA its address. If we didn't have an error, set the address field of the procedure in the table to code_index.
- The last instruction of every subprocedure is a RTN. If there was an explicit RTN, meaning the last statement was a return and thus the last instruction is a RTN, we don't need to do anything. Otherwise the return is implicit and we need to emit a RTN instruction.
- Now we know how long the procedure is and where we need to jump to to skip it. Thus we can fix the M value of the JMP from before so it equals the index of the next instruction in code.

```
return [HLT|RTN]
```

- We refer to this as an explicit return. For subprocedures, we emit RTN instructions. For main, we emit a HLT. We know if we're in main if the global level equals 0.

```
CONDITION ::= EXPRESSION (==|!=|<|<=|>|>=|17) EXPRESSION
                  (EQL|NEQ|LSS|LEQ|GTR|GEQ|OPR)
```

- After the first call to EXPRESSION, when you check for the relational operator you must save which relational operator it is so you know which instruction to emit after the second EXPRESSION call.
- If you had error 17, you can emit an OPR instruction with M value -1.

```
EXPRESSION ::= TERM {(+|-) TERM (ADD|SUB)}
```

- After the first TERM call, you'll loop while the current token is *plus* or *minus*. You'll need to save the token type so you know which instruction to emit after the second TERM call.

```
TERM ::= FACTOR {(*|/) FACTOR (MUL|DIV)}
```

- After the first FACTOR call, you'll loop while the current token is *times* or *division*. You'll need to save the token type so you know which instruction to emit after the second FACTOR call.

```
FACTOR::= ident [8-5|18] (LOD|LIT) | number LIT |
          ( EXPRESSION ()|19) | 20
```

- For the *identifier* case
  - It's possible that find_symbol returns an index for both the constant and the variable case.

    - If it returns -1 for both, you'll have an error (check Appendix B for more details).
    - If it returns just a constant, emit a LIT instruction, M is the value of the symbol from the table
    - If it returns just a variable, emit a LOD instruction, L is the global level minus the level of the symbol from the table, M is the address of the symbol from the table
    - If it returns an index for both, you decide whether to use the constant (and emit a LIT) or the variable (and emit a LOD) based on the level fields of the two indices. Select the index with the higher level field (the global level is the highest possible level, and we prefer the symbols of the current procedure over the symbols of the parent procedures).
- For the *number* case, the M field of the LIT instruction is the value of the *number* token
- If the current token is not an *identifier*, a *number*, or a *left_parenthesis*, you have error 20 (see Appendix B for more details).

Appendix B: Error Messages

There are three types of error messages in PL/0:

A. Always non-stopping - these errors can always be continued past; error 1 is always non-stopping because the period is the final symbol in a program, all the other errors of this category (3, 7, 8-1, 8-2, 8-3, 8-4, 8-5, 9, 13, 14, 18, 21, 22, and 23) have to do with an improperly used identifier: the grammar expects an identifier and one is present, but it can't be used for some reason, usually because of a symbol table conflict

B. Always stopping - there is only one error of this type: error 6-4. Usually missing symbol errors fall under type C, but this error is always stopping because if you wanted to continue past it, you would have to move backwards in the grammar, which is not feasible.

C. Sometimes non-stopping - these errors occur because a symbol is missing; they can be non-stopping if the symbol present is in the desired symbol's follow set, otherwise they are stopping; for all these errors, the grammar line where the error is found is shown with the desired symbol in red and the acceptable follow in blue.

Error messages for the tiny PL/0 Parser:

1. **missing .** - always non-stopping because this is the very last symbol of any program
2. **missing identifier**
    1. **after keyword const** - sometimes non-stopping

```
const ::= ["const" ident ":=" ["-"] number ";"]
```

    2. **after keyword var** - sometimes non-stopping

```
var ::= [ "var" ident ";"]
```

    3. **after keyword procedure** - sometimes non-stopping

```
proc ::= [ "procedure" ident ";" ]
```

    4. **after keyword call** - sometimes non-stopping

```
statement    ::= [ … | "call" ident | … ]
```

Because the desired symbol is the last symbol in the statement, the acceptable follow set is the follow set for the statement non-terminal. To find this, we need to find all the occurrences of statement in the grammar:

```
block ::= declarations statement
```

Because statement is the last character in the block definition, its follow set includes the follow set for the block non-terminal. To find this, we need to find all the occurrences of block in the grammar:

```
program ::= block "."
statement ::= [ … | "def" ident "{" block "}" | … ]
```

So the follow set for block is [. }]

On to the next occurrence of statement:

```
statement ::= [ …|"begin" statement {";" statement} "end"|… ]
statement ::= [ … | "if" condition "then" statement | … ]
statement ::= [ … | "while" condition "do" statement | … ]
```

The last two occurrences of statement are the last characters of statement, so we can ignore them. Thus the follow set for the identifier after keyword call is [. } ; end]

5. **after keyword read** - sometimes non-stopping

```
statement    ::= [ … | "read" ident | … ]
```

Because the desired symbol is the last symbol in the statement, the acceptable follow set is the follow set for the statement non-terminal. Luckily we already found this in error 2-4. The follow set is [. } ; end]

6. **after keyword def** - sometimes non-stopping

```
statement ::= [ … | "def" ident "{" block "}" | … ]
```

3. **identifier is declared multiple times by a procedure** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in the three declarations: const, var, proc. It is found when there is an identifier present, but it has already been used by another symbol in the procedure. We know if this has happened by calling support function multiple_declaration_check. We want this function to return -1, meaning the name hasn't been used. If it returns any other value, we have error 3.
4. **missing := in**
    1. **constant declaration** - sometimes non-stopping

```
const ::= ["const" ident ":=" ["-"] number ";"]
```

Because the minus token is optional, the token that follows it, number, is also in the follow set for the assignment symbol.

2. **assignment statement** - sometimes non-stopping

```
statement    ::= [ … | ident ":=" expression | … ]
```

Because the token that follows the assignment_symbol is the non-terminal expression, our follow set includes the first set of expression.

```
expression ::= term { ("+"|"-") term}
```

Because expression starts with the non-terminal term, its first set includes the first set of term.

```
term ::= factor {("*"|"/") factor}
```

Because term starts with the non-terminal factor, its first set includes the first set of factor.

```
factor ::= ident | number | "(" expression ")"
```

Because factor is not nullable, meaning there must be at least one token found when factor is called and we can't just skip past it, then term is not nullable and expression is not nullable, so we don't need to include the follow set of statement. Thus the follow set for the assignment_symbol in an assignment statement is [ident number (]

5. **missing number in constant declaration** - sometimes non-stopping

```
const ::= ["const" ident ":=" ["-"] number ";"]
```

6. **missing ; after**
    1. **constant declaration** - sometimes non-stopping

```
const ::= ["const" ident ":=" [“-“] number “;"]
```

Because the semicolon is the last token of the const non-terminal definition, the follow set includes the follow set of the non-terminal const. To find this we need to find all the occurrences of const in the grammar:

```
declarations ::= { const | var | proc }
```

Because declarations is a loop of const, var, or proc, const can be followed by another const, a var, or a proc. Thus the follow set includes the first set of all these non-terminals:

```
const ::= ["const" ident ":=" [“-“] number “;"]
var ::= [ “var" ident “;”]
proc ::= [ "procedure" ident ";" ]
```

All of these are nullable (because they are all wrapped in optional brackets), but they are also all the last token of their declaration definition and they only occur in the declarations non-terminal. Thus the follow set of const (and var and proc) includes the follow set of the non-terminal declarations. To find this we need to find all the occurrences of declarations in the grammar:

```
block ::= declarations statement
```

Because the next token is the statement non-terminal, the follow set includes the first set of statement:

```
statement ::= [ ident ":=" expression
                | "call" ident
                | "begin" statement { “;” statement } "end"
                | "if" condition "then" statement
                | "while" condition "do" statement
                | "read" ident
                | "write" expression
                | “def” ident “{” block “}”
                | “return”]
```

In summary the first set of statement is [**ident call begin if while read write def return**]

Finally, because statement is nullable (because the entire definition is wrapped in optional brackets), this means declarations can be the last token of the block definition, and thus the follow set of declarations includes the follow set of block. Luckily we already found this in error 2-4: [**. }**]

So in summary, the follow set of the semicolon at the end of constant declaration include the first set of all the symbol declarations [**const var procedure**] and the follow set of the declarations non-terminal which includes the first set of the statement non-terminal [**ident call begin if while read write def return**] and the follow set of the block non-terminal [**. }**]. In total the follow set is [**const var procedure ident call begin if while read write def return . }**]

2. **variable declaration** - sometimes non-stopping

```
var ::= [ "var" ident ";"]
```

Because the semicolon is the last token of the var non-terminal definition, the follow set includes the follow set of the non-terminal var. As we saw in error 6-1, the follow sets of const, var, and procedure are all the same. So the follow set is [**const var procedure ident call begin if while read write def return . }**]

3. **procedure declaration** - sometimes non-stopping

```
proc ::= [ "procedure" ident ";"]
```

Because the semicolon is the last token of the proc non-terminal definition, the follow set includes the follow set of the non-terminal proc. As we saw in error 6-1, the follow sets of const, var, and procedure are all the same. So the follow set is [**const var procedure ident call begin if while read write def return . }**]

4. **statement in begin-end** - always stopping. To understand why this error is always stopping, let's start by looking at the begin-end statement.

```
statement ::= [ …|"begin" statement {";" statement} "end"|… ]
```

Control in begin-end continuously calls statement, it loops on whether the current token is a semicolon. When this condition is no longer true, the looping stops. Following the logic of the missing token errors, we should be able to continue processing if the present token is within the follow set. The follow set will be the first set of statement which we found above in error 6-1 [**ident call begin if while read write def return**], and since statement is nullable, we also include what follows the statement token within the definition [**end**].

There are two issues with this. The first, more pressing issue, is how to continue. Since the absence of the semicolon caused the looping to stop, we have stopped looping. There's no feasible way for us to resume. Thus this error is always stopping. The other issue is that if the token present is instead **end**, then we don't actually have an error because **";" statement** is wrapped in curly braces, meaning it's optional.

So, we stop looping if the current token is not a semicolon. The next token should be the end keyword. If the next token is not the end keyword, we have an error. You might be thinking to yourself, what about error 10 (begin must be followed by end), how do we tell the difference. The solution is this, if the current token is in the follow set of the semicolon [**ident call begin if while read write def return**] (with keyword_end removed because we only have an error at all if the current token isn't end), then we have error 6-4 and we stop. Otherwise, we'll have error 10. See error 10 for the derivation of the follow set to determine when error 10 stops.

7. **procedures and constants cannot be assigned to** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in assignment statements. It is found when there is an identifier present, but we can't find a variable in the symbol table with the desired

name. We know we have an error if the support function find_symbol returns -1 when we give it the identifier name and kind 2. The support function would return the index of the desired symbol in the table if one is available; it returns -1 when it can't find anything. You might be wondering, how do we know that we have error 7 and not error 8-1, which is found in the same place? We know that we have error 7 if there **is** a procedure or a constant in the symbol table with the desired name. Thus if find_symbol(desired name, kind 2) returned -1 and find_symbol(desired name, kind 3) or find_symbol(desired name, kind 1) return something other than -1, we have error 7. If find_symbol(desired name, kind 2) returned -1 and find_symbol(desired name, kind 3) and find_symbol(desired name, kind 1) both also return -1, we have error 8-1.

8. **undeclared identifier used in**
    1. **assignment statement** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in assignment statements at the same place as error 7, see error 7 for an explanation on how to determine if there is an error and which error is present if there is one.
    2. **call statement** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in call statements at the same place as error 9, see error 9 for an explanation on how to determine if there is an error and which error is present if there is one.
    3. **read statement** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in read statements at the same place as error 13, see error 13 for an explanation on how to determine if there is an error and which error is present if there is one.
    4. **define statement** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in define statements at the same place as error 14, see error 14 for an explanation on how to determine if there is an error and which error is present if there is one.
    5. **arithmetic expression** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in factor at the same place as error 18, see error 18 for an explanation

on how to determine if there is an error and which error is present if there is one.

9. **variables and constants cannot be called** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in call statements. It is found when there is an identifier present, but we can't find a procedure in the symbol table with the desired name. We know we have an error if the support function find_symbol returns -1 when we give it the identifier name and kind 3. The support function would return the index of the desired symbol in the table if one is available; it returns -1 when it can't find anything. You might be wondering, how do we know that we have error 9 and not error 8-2, which is found in the same place? We know that we have error 9 if there **is** a variable or a constant in the symbol table with the desired name. Thus if find_symbol(desired name, kind 3) returned -1 and find_symbol(desired name, kind 2) or find_symbol(desired name, kind 1) return something other than -1, we have error 9. If find_symbol(desired name, kind 3) returned -1 and find_symbol(desired name, kind 2) and find_symbol(desired name, kind 1) both also return -1, we have error 8-2.
10. **begin must be followed by end** - sometimes non-stopping; please look at error 8-4 for how to know if error 10 is present. To know whether error 10 is stopping, we need to check if the current token is in the follow set of the end token.

`statement ::= [ …|"begin" statement {";" statement} "end"|… ]`

Because the end keyword is the last token of the definition, the follow set is the follow set of the statement non-terminal. Luckily we already found this in error 2-4: [**. } ; end**]. Of course we omit the end keyword, because we won't have this error if it is present.

11. **if must be followed by then** - sometimes non-stopping

`statement ::= [ … | "if" condition "then" statement | … ]`

Because the then keyword is followed by the non-terminal statement, the follow set includes the first set of statement. Luckily we already found this in error 6-1: [**ident call begin if while read write def return**]. Since statement is nullable (see error 6-1 for explanation) and the last token of this definition, we also need to include the follow set of statement. Luckily we already found this in error 2-4: [**. } ; end**]. Thus the follow set of keyword then is [**. } ; end ident call begin if while read write def return**].

12. **while must be followed by do** - sometimes non-stopping

`statement ::= [ … | "while" condition "do" statement | … ]`

The structure of this definition is functionally the same as the if statement explored in error 11. Thus the follow set of keyword do is the same as keyword then's in error 11: [**. } ; end ident call begin if while read write def return**].

13. **procedures and constants cannot be read** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in read statements. It is found when there is an identifier present, but we can't find a variable in the symbol table with the desired name. We know we have an error if the support function find_symbol returns -1 when we give it the identifier name and kind 2. The support function would return the index of the desired symbol in the table if one is available; it returns -1 when it can't find anything. You might be wondering, how do we know that we have error 13 and not error 8-3, which is found in the same place? We know that we have error 13 if there **is** a procedure or a constant in the symbol table with the desired name. Thus if find_symbol(desired name, kind 2) returned -1 and find_symbol(desired name, kind 3) or find_symbol(desired name, kind 1) return something other than -1, we have error 13. If find_symbol(desired name, kind 2) returned -1 and find_symbol(desired name, kind 3) and find_symbol(desired name, kind 1) both also return -1, we have error 8-3.
14. **variables and constants cannot be defined** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in define statements. It is found when there is an identifier present, but we can't find a procedure in the symbol table with the desired name. We know we have an error if the support function find_symbol returns -1 when we give it the identifier name and kind 3. The support function would return the index of the desired symbol in the table if one is available; it returns -1 when it can't find anything. You might be wondering, how do we know that we have error 14 and not error 8-4, which is found in the same place? We know that we have error 14 if there **is** a variable or a constant in the symbol table with the desired name. Thus if find_symbol(desired name, kind 3) returned -1 and find_symbol(desired name, kind 2) or find_symbol(desired name, kind 1) return something other than -1, we have error 14. If find_symbol(desired name, kind 3) returned -1 and find_symbol(desired name, kind 2) and find_symbol(desired name, kind 1) both also return -1, we have error 8-4.
15. **missing {** - sometimes non-stopping

```
statement ::= [ … | "def" ident "{" block "}" | … ]
```

Because the left_curly_brace is followed by non-terminal block, the follow set includes the first set of block.

```
block ::= declarations statement
```

Because the declarations non-terminal is the first token of the block definition, the follow set includes the first set of declarations.

```
declarations ::= { const | var | proc }
```

Because declarations can start with the non-terminals const, var, or proc, the follow set includes the first sets of all three. Luckily we found these in error 6-1: [**const var procedure**]. Since declarations is nullable (see error 6-1 for explanation), then the statement non-terminal can be the first token of the block definition, so the follow set includes the first set of statement.

```
block ::= declarations statement
```

Luckily we already found this in error 6-1: [**ident call begin if while read write def return**]. Because statement is also nullable, this means the block non-terminal is also nullable.

```
statement ::= [ … | "def" ident "{" block "}" | … ]
```

So in summary the follow set of the left_curly_brace is [**const var procedure ident call begin if while read write def return }**]

16. **{ must be followed by }** - sometimes non-stopping

```
statement ::= [ … | "def" ident "{" block "}" | … ]
```

Because the token is the last of this statement definition, the follow set is the follow set of statement. We already found this in error 2-4: [**. } ; end**]

17. **missing relational operator** - sometimes non-stopping

```
condition ::= expression ("=="|"!="|"<"|"<="|">"|">=")
expression
```

This error looks a little different because there are multiple valid token types rather than just one. Since the relational operator is followed by the non-terminal expression, the follow set is the first set of expression. We already found this in error 4-2: [**ident number (**].

18. **procedures cannot be used in arithmetic** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in factor It is found when there is an identifier present, but we can't find a variable or a constant in the symbol table with the desired name. We know we have an error if find_symbol(desired name, 1) and find_symbol(desired name, 2) both return -1. The support function would return the index of the desired symbol in the table if one is available; it returns -1 when it can't find anything. You might be wondering, how do we know that we have error 18 and not error 8-5, which is found in the same place? We know that we have error 18 if there **is** a procedure in the symbol table with the desired name. Thus if find_symbol(desired name, kind 1) and find_symbol(desired name, kind 2) returned -1 and find_symbol(desired name, kind 3) returned something other than -1, we have error 18. If find_symbol(desired name, kind 1) and find_symbol(desired name, kind 2) both returned -1 and find_symbol(desired name, kind 3) also returned -1, we have error 8-5.

19. **(must be followed by)** - sometimes non-stopping

```
factor ::= ident | number | "(" expression ")"
```

Since the left parenthesis is the last token of the factor definition, the follow set is the follow set of the factor non-terminal. To find this we find all the occurrences of the factor non-terminal.

```
term ::= factor {("*"|"/") factor}
```

Since factor is also the last token of the term definition, the follow set includes the follow set of the term non-terminal. To find this we find all the occurrences of the term non-terminal.

```
expression ::= term { ("+"|"-") term}
```

Since term is also the last token of the expression definition, the follow set includes the follow set of the expression non-terminal. To find this we find all the occurrences of the expression non-terminal.

```
statement ::= [ … | ident ":=" expression | … ]
statement ::= [ … | "write" expression | … ]
```

In both these cases, expression is the last token of statement, the follow set includes the follow set of statement. We already found this in error 2-4: [. } ; end].

```
condition ::= expression ("=="|"!="|"<"|"<="|">"|">=")
expression
```

Since expression is the last token of condition, the follow set includes the follow set of condition. To find this we find all the occurrences of the condition non-terminal.

```
statement ::= [ … | "if" condition "then" statement | … ]
statement ::= [ … | "while" condition "do" statement | … ]
```

In summary the follow set is [* / + - . } ; end == != < <= > >= then do]

20. **invalid expression** - sometimes non-stopping

```
factor ::= ident | number | "(" expression ")"
```

The meaning of this error is not immediately obvious based on its language, but this error is found when you're in factor and the current token is not an identifier, number, or left parenthesis. For the identifier and number cases, they are the last tokens of the definition, so the follow set includes the follow set of the non-terminal factor. We found this in error 19: [* / + - . } ; end == != < <= > >= then do]. For the left parenthesis case, the follow set is the first set of expression. We already found this in error 4-2: [ident number (]. However you'll notice that this is also the first set of factor, and this error will only occur if the current token is not one of these three types. Therefore the final follow set is [* / + - . } ; end == != < <= > >= then do].

21. **procedure being called has not been defined** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in program (parse if you didn't make program a separate function). For CAL instructions, the M field should be the index in the code array where the procedure starts (aka its address field in the symbol table). However, when we emit

CAL instructions (in STATEMENT), the procedure may not have been defined yet (its address field is -1). So instead, when we emit CAL instructions, we use the index of the procedure in the symbol table for the M field. This way, at the very end once all the procedures are defined and we know their addresses, we can replace the M fields with the proper values. When we go to do this, if the procedure's address field is still -1 (meaning it hasn't been defined), then we have error 21.

22. **procedures can only be defined within the procedure that declares them** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in define statements. When we find the desired procedure (meaning we didn't have errors 8-4 or 14), we need to check that the procedure belongs to the current parent procedure. To do this simply check that the level field of procedure in the symbol table. If it's equal to the global level, we're good. Otherwise, we have error 22.
23. **procedures cannot be defined multiple times** - always non-stopping. Errors that have to do with the use of symbols are always non-stopping, because the desired token type is present, there's just an issue with the symbol itself. This particular error is found in define statements. When we find the desired procedure (meaning we didn't have errors 8-4 or 14) and we know it belongs to the active procedure (meaning we didn't have error 22), we need to check that the procedure has not already been defined. When we define a procedure we set its address field in the symbol table. Before that point, the address is -1. So if the procedure's address is not -1 before we define it, we have error 23.

Please note that we will check for the correct implementation of all of these errors. There is a function in parser.c which will print the error message for you. DO NOT ALTER THE ERROR LIST. All errors should checked for at least once, some may have checks in multiple locations.

Appendix C: Examples

Input Program (example.txt): *The formatting and spacing is a little weird on this test case to ensure that it has fewer than 500 characters.*

```
var a; const b:=3; procedure c; var d; procedure e; const
f:=-1;
begin
     read a;
     d:=a+b*f-(4/2);
     if a==d then write 3;
     if a+2!=d then write 3+1;
     if a<d*(3+8) then write 3-2;
     if a-1<=d+4 then write 3*4/6;
     while 3>7 do write 3*(a+d);
     if a>=d then
          begin
               call e;
               return
          end;
     def e{
          var b; const a:=4;
          begin
               b:=a+b;
               a:=b-3;
               call c;
               return;
          end};
     def c{
          procedure a; procedure f;
          begin
               def a{};
               return;
               call a;
               def f{call f}
          end};
     return;
end.
```

Output (example_out.txt): *Because the program doesn't have any errors and it contains a read instruction, when you run it, it will wait for you to input a value. This doesn't impact the parser output so it doesn't strictly matter, but example_out.txt was created with input "2".*

```
Assembly Code:
Line OP Code   OP Name   L    M
0    7    INC  0    5
1    10   RED  0    2
2    4    STO  0    3
3    3    LOD  0    3
4    1    LIT  0    3
5    1    LIT  0    -1
6    2    MUL  0    3
7    2    ADD  0    1
8    1    LIT  0    4
9    1    LIT  0    2
10   2    DIV  0    4
11   2    SUB  0    2
12   4    STO  0    4
13   3    LOD  0    3
14   3    LOD  0    4
15   2    EQL  0    5
16   9    JPC  0    19
17   1    LIT  0    3
18   10   WRT  0    1
19   3    LOD  0    3
20   1    LIT  0    2
21   2    ADD  0    1
22   3    LOD  0    4
23   2    NEQ  0    6
24   9    JPC  0    29
25   1    LIT  0    3
26   1    LIT  0    1
27   2    ADD  0    1
28   10   WRT  0    1
29   3    LOD  0    3
30   3    LOD  0    4
31   1    LIT  0    3
32   1    LIT  0    8
33   2    ADD  0    1
34   2    MUL  0    3
35   2    LSS  0    7
36   9    JPC  0    41
```

```
37   1    LIT  0    3
38   1    LIT  0    2
39   2    SUB  0    2
40   10   WRT  0    1
41   3    LOD  0    3
42   1    LIT  0    1
43   2    SUB  0    2
44   3    LOD  0    4
45   1    LIT  0    4
46   2    ADD  0    1
47   2    LEQ  0    8
48   9    JPC  0    55
49   1    LIT  0    3
50   1    LIT  0    4
51   2    MUL  0    3
52   1    LIT  0    6
53   2    DIV  0    4
54   10   WRT  0    1
55   1    LIT  0    3
56   1    LIT  0    7
57   2    GTR  0    9
58   9    JPC  0    66
59   1    LIT  0    3
60   3    LOD  0    3
61   3    LOD  0    4
62   2    ADD  0    1
63   2    MUL  0    3
64   10   WRT  0    1
65   8    JMP  0    55
66   3    LOD  0    3
67   3    LOD  0    4
68   2    GEQ  0    10
69   9    JPC  0    72
70   5    CAL  0    73
71   10   HLT  0    3
72   8    JMP  0    84
73   7    INC  0    4
74   1    LIT  0    4
75   3    LOD  0    3
76   2    ADD  0    1
77   4    STO  0    3
78   3    LOD  0    3
79   1    LIT  0    3
```

```
80   2    SUB  0    2
81   4    STO  1    3
82   5    CAL  1    85
83   6    RTN  0    0
84   8    JMP  0    95
85   7    INC  0    3
86   8    JMP  0    89
87   7    INC  0    3
88   6    RTN  0    0
89   6    RTN  0    0
90   5    CAL  0    87
91   8    JMP  0    95
92   7    INC  0    3
93   5    CAL  1    92
94   6    RTN  0    0
95   10   HLT  0    3

Symbol Table:
Kind | Name        | Value | Level | Address | Mark
---------------------------------------------------
   2 |           a |     0 |     0 |       3 |     1
   1 |           b |     3 |     0 |       0 |     1
   3 |           c |     0 |     0 |      85 |     1
   2 |           d |     0 |     0 |       4 |     1
   3 |           e |     0 |     0 |      73 |     1
   1 |           f |    -1 |     0 |       0 |     1
   2 |           b |     0 |     1 |       3 |     1
   1 |           a |     4 |     1 |       0 |     1
   3 |           a |     0 |     1 |      87 |     1
   3 |           f |     0 |     1 |      92 |     1


Input :
Output : 4

Output : 2
```

The following test case contains all of the non-stopping errors you will need to implement.

Input Program (errors.txt)**:**

```
procedure a;
const a  3 const := ;
var ; var a
procedure a procedure ;
```

```
var b; const c := -1; procedure d;
begin
     call a;
     z  z; c := d; d := (1;
     call z; call b; call c; call ;
     if 3 3 ;
     while 4 > 8 ;
     read ; read z; read c; read d;
     write ;
     def {}; def z {}; def b {; def c };
     def d {};
     def d {
          begin
               def d{}
     };
end
```

Output (errors_out.txt):

```
Parser Error 3: identifier is declared multiple times by a
procedure
Parser Error 4: missing := in constant declaration
Parser Error 6: missing ; after constant declaration
Parser Error 2: missing identifier after keyword const
Parser Error 5: missing number in constant declaration
Parser Error 2: missing identifier after keyword var
Parser Error 3: identifier is declared multiple times by a
procedure
Parser Error 6: missing ; after variable declaration
Parser Error 3: identifier is declared multiple times by a
procedure
Parser Error 6: missing ; after procedure declaration
Parser Error 2: missing identifier after keyword procedure
Parser Error 8: undeclared identifier used in assignment
statement
Parser Error 4: missing := in assignment statement
Parser Error 8: undeclared identifier used in arithmetic
expression
Parser Error 7: procedures and constants cannot be assigned
to
Parser Error 18: procedures cannot be used in arithmetic
Parser Error 7: procedures and constants cannot be assigned
to
Parser Error 19: ( must be followed by )
```

```
Parser Error 8: undeclared identifier used in call statement
Parser Error 9: variables and constants cannot be called
Parser Error 9: variables and constants cannot be called
Parser Error 2: missing identifier after keyword call
Parser Error 17: missing relational operator
Parser Error 11: if must be followed by then
Parser Error 12: while must be followed by do
Parser Error 2: missing identifier after keyword read
Parser Error 8: undeclared identifier used in read statement
Parser Error 13: procedures and constants cannot be read
Parser Error 13: procedures and constants cannot be read
Parser Error 20: invalid expression
Parser Error 2: missing identifier after keyword def
Parser Error 8: undeclared identifier used in define
statement
Parser Error 14: variables and constants cannot be defined
Parser Error 16: { must be followed by }
Parser Error 14: variables and constants cannot be defined
Parser Error 15: missing {
Parser Error 23: procedures cannot be defined multiple times
Parser Error 22: procedures can only be defined within the
procedure that declares them
Parser Error 10: begin must be followed by end
Parser Error 1: missing .
Parser Error 21: procedure being called has not been defined
```

Input Program (stoppers.txt): *The formatting and spacing is a little weird on this test case to ensure that it has fewer than 500 characters. Because of the nature of the test case, we don't provide an output file.*

```
? bc the errors stop you have to fix them to see the next
? replace * or var w/ the missing token
const * := 2; ? 2-1
const b * 2; ? 4-1
const c := *; ? 5
const d := 2 * ? 6-1
var *; ? 2-2
var e * ? 6-2
procedure *; ? 2-3
procedure f * ? 6-3
procedure g;
begin
     e * 3; ? 4-2
     call *; ? 2-4
```

```
     begin * end; ? 10
     write 3 ? 6-4
     write (3 var; ? 19
     if 4>8 var; ? 11
     while 4>8 var; ? 12
     read *; ? 2-5
     def *{}; ? 2-6
     def f*}; ? 15
     def g{*; ? 16
     if 4 var 8 then; ? 17
     write var; ? 20
end.
```