

ZON is a simple programming language for studying the development process and tools of creating a programming language. This is ZON v1, I think the previous v0 was too complex for ChatGPT helping me accomplish it, so let's make the v1 more simpler.

Conclusion

This is the 2nd attempt to create the ZON language with ChatGPT. The progress of this version, v1, is ChatGPT helped me to create the parser and lexer for the ZON language, but there is no output from the terminal and the program is keep hanging when running the `hello.zon`

Here is the commands to run the ZON v1:

```
$ flex scanner.l
$ bison -d parser.y
parser.y: warning: 25 shift/reduce conflicts [-Wconflicts-sr]
parser.y: note: rerun with option '-Wcounterexamples' to generate
conflict counterexamples
$ gcc lex.yy.c parser.tab.c -o zon
$ ./zon hello.zon
^C
$
```

Oop! LLM model service, like ChatGPT, cannot `remember` chat history during the conversation, so when I asking for `hello.zon` file which I provided in the beginning of our chat, ChatGPT has no idea about it. The `memory` is limited by the token length.

The token limitation of GPT

- GPT-3 is `2049` , around 1500 words, 3 pages
- GPT-3.5 is `4096` , around 3000 words, 6 pages
- GPT-4-8K is `8129` , around 6000 words, 12 pages
- GPT-4-32K is `32768` , around 24000 words, 50 pages

ChatGPT is based on GPT3.5 at the moment, so it can only reference around 3000 words in the history of our conversation which means **the ZON language needs to be done in 3000 words, is that feasible?**

Development and Discussions with ChatGPT

Q: ZON is a simple programming language for briefly understand programming language development process and tools.

you can reference to links below to show me the steps of creating the ZON lanague

- <https://dev.to/evantypanski/writing-a-simple-programming-language-from-scratch-part-1-54a2>
- <https://dev.to/evantypanski/writing-a-simple-programming-language-from-scratch-part-2-2522>
- <https://dev.to/evantypanski/writing-a-simple-programming-language-from-scratch-part-3-1d7l>

here are some requirements:

1. ZON is a strongly typed lanauge
2. support only data types of `int`, `float`, `string`, `bool`
3. support only Arithmetic operators, no bitwise operations, no logical operators
4. support only the `if-else` control flow

Sure, I can guide you through the steps of creating the ZON programming language as outlined in the three-part tutorial series by Evan Typanski on dev.to.

Part 1: Setting up the Project

In the first part of the tutorial, you will learn how to set up the project and define the basic structure of the language.

1. Set up the project directory with the following files:

- `lexer.py` for the lexer (also known as the tokenizer)
- `parser.py` for the parser
- `ast.py` for the abstract syntax tree
- `evaluator.py` for the evaluator

2. Define the basic structure of the language, including the supported data types (`int`, `float`, `string`, `bool`), operators (+, -, *, /), and statements (assignment, if-else).

3. Define the lexer, which will convert the input string into a stream of tokens. Each token represents a single unit of meaning in the language, such as a keyword, identifier, or operator.
4. Test the lexer by running it on some sample code and printing out the resulting tokens.

Part 2: Parsing the Code

In the second part of the tutorial, you will learn how to parse the code and build an abstract syntax tree (AST) from the tokens generated by the lexer.

1. Define the grammar of the language using a context-free grammar (CFG) or a variant of Backus-Naur Form (BNF).
2. Use a parser generator such as PLY (Python Lex-Yacc) to generate a parser from the grammar.
3. Implement the parser and use it to generate an abstract syntax tree (AST) from the input code. The AST represents the code as a tree of nodes, with each node representing a different part of the code (such as a statement or an expression).
4. Test the parser by running it on some sample code and printing out the resulting AST.

Part 3: Evaluating the AST

In the third and final part of the tutorial, you will learn how to evaluate the AST and execute the code.

1. Define the evaluation rules for the language, which determine how the AST is traversed and evaluated.
2. Implement an evaluator that takes an AST and applies the evaluation rules to produce a result.
3. Test the evaluator by running it on some sample code and printing out the resulting output.
4. Add support for the `if-else` control flow statement by modifying the evaluation rules and updating the parser and lexer accordingly.

Once you have completed these steps, you will have a functional programming language that supports the basic data types and arithmetic

operations, as well as the `if-else` control flow statement.

Q: this is an example of ZON source code, called `hello.zon`:

```
// declare add function
fn add(x: int, y: int) int{
    return x + y;
}

// declare test function
fn test(){
    // declare an array with 10 elements
    var x = []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for(i in x){
        if((i % 2) == 0){
            if(i <= 5){
                print("i = {}\n", i);
            }
        }else{
            if(i >= 5){
                print("b = {}\n", i);
            }
        }
    }
    print("hello\n");
}

fn main(){
    // x is an integer with a value of 123
    var x: int = 123;
    // y is a float point with a value of 3.14
    var y: float = 3.14;
    // z is a boolean with a value of true
    var z: bool = true;
    // str is a string with a value of 'hello world!'
    var str: string = "hello world!";

    // arr is an array with 5 integer elements
```

```

var arr = [5]int{1, 2, 3, 4, 5};

// arithmetic operations
var a = (x + 256) * 2;
var b = (y - 1.0) / 2;
var c = a % 2;

print("1 + 2 = {}\n", add(1,2));

}

```

is it good for writing the grammar in EBNF? is it missing something?

Yes, you can write the ZON grammar in Extended Backus-Naur Form (EBNF) to define the language syntax more precisely. Here's an example of how you could write the grammar for the ZON programming language in EBNF:

```

program          = { function_def | statement } ;
function_def     = "fn", identifier, "(", [ parameters ], ")", type,
block ;
parameters       = parameter, { ",", parameter } ;
parameter        = identifier, ":", type ;
type             = "int" | "float" | "string" | "bool" ;
block            = "{", { statement }, "}" ;
statement        = declaration | assignment | if_statement |
for_statement | print_statement ;
declaration      = "var", identifier, ":", type, [ "=", expression
], ";" ;
assignment       = identifier, "=", expression, ";" ;
if_statement     = "if", "(", expression, ")", block, [ "else",
block ] ;
for_statement    = "for", "(", identifier, "in", expression, ")",
block ;
print_statement  = "print", "(", expression, { ",", expression },
")", ";" ;
expression       = and_expression ;
and_expression   = equality_expression, { "&&", equality_expression
} ;

```

```

equality_expression = relational_expression, { ("==" | "!="),
relational_expression } ;
relational_expression = additive_expression, { ("<" | ">" | "<=" |
">="), additive_expression } ;
additive_expression = multiplicative_expression, { ("+" | "-"),
multiplicative_expression } ;
multiplicative_expression = primary_expression, { ("*" | "/" |
"%"), primary_expression } ;
primary_expression = literal | identifier | function_call | "(",
expression, ")" ;
literal          = int_literal | float_literal | string_literal |
bool_literal | array_literal ;
int_literal      = digit, { digit } ;
float_literal    = digit, { digit }, ".", { digit } ;
string_literal   = "'", { character }, "'" ;
bool_literal     = "true" | "false" ;
array_literal    = "[", [ expression, { ",", expression } ], "]",
"int" | "float" | "string" | "bool" ;
function_call    = identifier, "(", [ expression, { ",", expression
} ], ")" ;
identifier       = letter, { letter | digit | "_" } ;
digit            = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
"8" | "9" ;
letter           = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" ;
character        = letter | digit | " " | "!" | "@" | "#" | "$" |
"%" | "^" | "&" | "*" | "(" | ")" | "-" | "+" | "=" | "[" | "]" |
"{" | "}" | ";" | ":" | "," | "." | "/" | "?" | "<" | ">" | "|" ;

```

This grammar should cover most of the syntax in the `hello.zon` example code you provided, including the function definition, variable declaration, arithmetic operations, and control flow statements such as `if` and `for`. However, it's possible that some features of ZON are missing from this grammar, as the language could potentially support additional features beyond what is shown in the example code. It's always a good idea to refer to the language specification or documentation to ensure that your grammar is complete and accurate.

Q: The part 1 of tutorial series by Evan Typanski is about creating `scanner.lex` and `parser.y` files. please give me the steps of creating `scanner.lex` and `parser.y` without example code.

Sure, here are the basic steps for creating `scanner.lex` and `parser.y` files:

1. Determine the grammar of your programming language: The first step is to determine the grammar of your programming language. This will define the syntax and structure of the language. You can use EBNF or a similar notation to define the grammar.
2. Create the `scanner.lex` file: The `scanner.lex` file defines the lexical rules for your programming language. This includes things like keywords, identifiers, literals, and operators. You will use a tool like Flex or JFlex to generate a C or Java program from your `scanner.lex` file.
3. Create the `parser.y` file: The `parser.y` file defines the syntax rules for your programming language. This includes the grammar rules defined in EBNF, as well as any additional semantic actions that need to be performed during parsing. You will use a tool like Bison or JavaCUP to generate a C or Java program from your `parser.y` file.
4. Define the AST: The AST (Abstract Syntax Tree) is a data structure that represents the syntax of your programming language. You will need to define the nodes of the AST and how they relate to each other based on the grammar of your language.
5. Integrate the scanner and parser: Once you have the scanner and parser generated from their respective files, you will need to integrate them into your compiler. This typically involves setting up communication between the scanner and parser, as well as defining how the AST is constructed during parsing.
6. Test and debug: Finally, you will need to test and debug your compiler to ensure that it is generating correct output for the input programs. You can use a variety of test cases to ensure that your compiler is working correctly.

Q: now, we have the grammar in EBNF which is based on `hello.zon`, let's call it `zon-grammar` show me the `zon-grammar`, please.

Sure, based on the `hello.zon` example code you provided, here is a possible `zon-grammar` in EBNF notation:

```
program ::= function_decl* main_function_decl

function_decl ::= "fn" function_name "(" param_decl ")"
               return_type_decl block_statement

param_decl ::= param_decl_list | empty
param_decl_list ::= param_decl_item ("," param_decl_item)*
param_decl_item ::= variable_name ":" type_name

return_type_decl ::= type_name | empty

main_function_decl ::= "fn" "main" "(" ")" block_statement

block_statement ::= "{" statement* "}"

statement ::= variable_decl | assignment_statement |
              function_call_statement | if_statement | for_statement |
              print_statement

variable_decl ::= "var" variable_name ":" type_name "=" expression
               ";"

assignment_statement ::= variable_name "=" expression ";"

function_call_statement ::= function_name "(" arg_list ")" ";"
arg_list ::= arg_item ("," arg_item)*
arg_item ::= expression

if_statement ::= "if" "(" expression ")" block_statement ("else"
block_statement)?

for_statement ::= "for" "(" variable_decl ";" expression ";"
assignment_statement ")" block_statement

print_statement ::= "print" "(" string_expression (","
```



```
expression)* ")" ";"
```

```
expression ::= literal | variable_name | arithmetic_expression
```

```
arithmetic_expression ::= expression (arithmetic_operator  
expression)*
```

```
literal ::= int_literal | float_literal | string_literal |  
bool_literal
```

```
int_literal ::= digit+
```

```
float_literal ::= digit+ "." digit+
```

```
string_literal ::= "\"" char* "\""
```

```
bool_literal ::= "true" | "false"
```

```
variable_name ::= letter (letter | digit | "_")*
```

```
type_name ::= "int" | "float" | "string" | "bool"
```

```
function_name ::= letter (letter | digit | "_")*
```

```
letter ::= "a" | "b" | "c" | ... | "z" | "A" | "B" | "C" | ... |  
"Z"
```

```
digit ::= "0" | "1" | "2" | ... | "9"
```

```
arithmetic_operator ::= "+" | "-" | "*" | "/" | "%"
```

```
string_expression ::= string_literal | variable_name
```

Note that this is just one possible implementation of `zon-grammar` and there can be variations based on the specific requirements of the language.

Q: give me another example of `zon-grammar` based on the requirements and `hello.zon`.

Sure, here is another possible implementation of `zon-grammar` based on the requirements and the `hello.zon` example code:

```
program ::= statement*
```

```
statement ::= variable_decl | assignment_statement |  
function_call_statement | if_statement | for_statement |  
print_statement
```

```
variable_decl ::= "var" variable_name ":" type_name "=" expression  
";"
```

```
assignment_statement ::= variable_name "=" expression ";"

function_call_statement ::= function_name "(" arg_list ")" ";"
arg_list ::= (expression ("," expression)*)?

if_statement ::= "if" "(" expression ")" "{" statement* "}"
("else" "{" statement* "}")?

for_statement ::= "for" "(" variable_name "in" expression ")" "{"
statement* "}"

print_statement ::= "print" "(" expression ")" ";"

expression ::= term (("+" | "-") term)*

term ::= factor (("*" | "/" | "%") factor)*

factor ::= literal | variable_name | "(" expression ")"

literal ::= int_literal | float_literal | string_literal |
bool_literal

int_literal ::= digit+

float_literal ::= digit+ "." digit+

string_literal ::= "\"" char* "\""

bool_literal ::= "true" | "false"

variable_name ::= letter (letter | digit)*

type_name ::= "int" | "float" | "string" | "bool"

function_name ::= letter (letter | digit)*

letter ::= "a" | "b" | "c" | ... | "z" | "A" | "B" | "C" | ... |
```

```
"Z"
```

```
digit ::= "0" | "1" | "2" | ... | "9"
```

This grammar is similar to the previous one, but with some differences in the syntax of the expressions and statements. For example, instead of using an `arithmetic_expression` non-terminal for arithmetic operations, we use a `term` non-terminal that handles multiplication, division, and modulo operations in addition to the `literal` and `variable_name` non-terminals. The `if` statement also uses curly braces to enclose the statement block, and the `for` statement uses the `in` keyword to iterate over a collection instead of the C-style syntax. The `arg_list` is made optional in the function call statement.

Q: how to validate the `zon-grammar`?

There are a few ways to validate the `zon-grammar`:

1. **Manually review the grammar:** You can manually review the grammar to ensure that it follows the syntax and structure rules of EBNF. You can check for things like consistency in naming conventions, correct use of symbols and operators, and proper use of terminals and non-terminals.
2. **Use a parser generator:** You can use a parser generator like Bison or ANTLR to automatically generate a parser from your grammar. These tools can automatically generate code for you that implements your grammar, and they will provide you with error messages if your grammar contains any ambiguities or conflicts.
3. **Create a sample program:** You can create a sample program in the ZON language and use it to test your grammar. This approach will help you identify any errors or ambiguities in the grammar, and will also give you a sense of how easy or difficult it is to write programs in the ZON language. You can use a text editor or integrated development environment (IDE) to write your sample program, and then test it against your grammar to ensure that it is valid.

Q: What are Frontend and Backend in the compiler manner?

In the context of compilers, the frontend and backend refer to different stages of the compilation process.

The **frontend** of a compiler is responsible for analyzing and parsing the source code written in a programming language and creating an intermediate representation of the program. This intermediate representation is typically in the form of an abstract syntax tree (AST) or a series of intermediate code instructions. The frontend also performs semantic analysis, which checks the program for syntax and semantic errors and ensures that it adheres to the rules of the programming language.

The **backend** of a compiler takes the intermediate representation generated by the frontend and translates it into machine code that can be executed on a particular platform or architecture. The backend typically consists of several stages, including code optimization, code generation, and assembly. The goal of the backend is to generate efficient machine code that can be executed by the target platform.

In summary, the frontend is responsible for parsing, analyzing, and translating the source code into an intermediate representation, while the backend takes that intermediate representation and generates efficient machine code that can be executed by a specific platform.

Q: how to use Flex and Bison to create the ZON programming language based on the `hello.zon` file? please show me the steps.

Sure! Here are the steps to use Flex and Bison to create the ZON programming language based on the `hello.zon` file:

1. Define the grammar: First, you need to define the grammar for the ZON programming language in the form of a context-free grammar. You can use the EBNF grammar that you have already created as a starting point.
2. Write the Lex file: Next, you need to create a file called `scanner.l` (or any name you prefer) that defines the lexical analyzer using Flex. In this file, you need to specify the regular expressions that define the tokens of the language. For example, you might define a regular expression for the `int` keyword or for integer literals.
3. Write the Bison file: After creating the Lex file, you need to create a file called `parser.y` (or any name you prefer) that defines the syntax analyzer using Bison. In this file, you need to specify the grammar rules for the language using Bison's syntax. You also need to specify

the actions to be taken when each rule is matched. For example, you might define a rule for an `if` statement and specify the action to be taken when the statement is executed.

4. Generate the scanner and parser: Once you have created the `scanner.l` and `parser.y` files, you need to use Flex and Bison to generate the scanner and parser code. You can do this by running the following commands in your terminal:

```
flex scanner.l  
bison -d parser.y
```

5. The `flex` command will generate a C program for the lexical analyzer, while the `bison` command will generate a C program for the syntax analyzer.
6. Write the driver program: Finally, you need to write a driver program that ties together the scanner and parser. This program will read in the source code, pass it to the scanner to generate tokens, and then pass the tokens to the parser to generate an abstract syntax tree. You can then use this abstract syntax tree to perform semantic analysis and generate machine code if desired.

That's it! With these steps, you can use Flex and Bison to create a compiler for the ZON programming language based on the `hello.zon` file.

Q: as previous discussion, we can use a parser generator like Bison to automatically generate a parser from the `zon-grammar` file. show me how to do that, please.

Sure, here are the steps to generate a parser using Bison:

1. Install Bison: If you haven't already, you'll need to install Bison. You can download it from the official Bison website or install it using a package manager like Homebrew on macOS or apt-get on Linux.
2. Create the Bison input file: Create a file called `parser.y` and copy the `zon-grammar` into it. This file will contain the Bison grammar rules.
3. Define the tokens: In the `parser.y` file, you need to define the tokens that your lexer will generate. These tokens should match the token names used in your Flex input file. You can define the tokens using the `%token` directive in the Bison file.

4. Define the grammar rules: After defining the tokens, you need to define the grammar rules. You can do this using Bison's grammar notation. Each grammar rule should be defined using the `%rule` directive.
5. Add semantic actions: You can add semantic actions to the grammar rules using C code. These actions will be executed when the corresponding rule is recognized. You can access the parsed tokens using special variables generated by Bison.
6. Generate the parser: Once you've defined the grammar rules and semantic actions, you can use Bison to generate a parser. To generate the parser, run the following command in your terminal:

```
bison -d parser.y
```

This will generate two files: `parser.tab.c` and `parser.tab.h`.

7. Include the generated files: In your main program, you need to include the generated `parser.tab.h` file. This file contains the declarations for the parser functions.
8. Write the lexer: Before you can use the parser, you need to write a lexer using Flex. The lexer should generate tokens that match the token names defined in the `parser.y` file.
9. Compile the code: Once you've written the lexer and included the generated files, you can compile the code. You'll need to link the lexer, parser, and any other dependencies together.
10. Test the parser: Finally, you can test the parser by passing input files to it. If the input files are valid according to your grammar, the parser should parse them correctly.

Note that this is a simplified overview of the process. There are many details involved in each step, and you may need to modify the steps based on your specific requirements.

Q: per our discussion, here are steps for creating the lexer and parser:

1. create `parser.y` file based on `hello.zon` in Bison grammar rules
2. generate `scanner.l` which is a lexer from the output of Bison, `parser.tab.c` and `parser.tab.h`

3. compile lexer and parser files and test the parser
is it correct?

Yes, that is correct!

Q: how to create `parser.y` file based on `hello.zon`?

To create the `parser.y` file based on `hello.zon`, you will need to follow the grammar rules of Bison. Here is an example of how you can create the `parser.y` file based on `hello.zon`:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Declare any additional functions and global variables here */

%}

/* Declare tokens here */
%token INT FLOAT STRING BOOL TRUE FALSE VAR FN RETURN FOR IN IF
ELSE

%%

/* Define grammar rules here */

program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    variable_declaration
```

```
| assignment_statement  
| function_call_statement  
| control_flow_statement  
| return_statement  
| expression_statement  
;
```

variable_declaration:

```
VAR identifier ':' data_type '=' expression ';'
;
```

assignment_statement:

```
identifier '=' expression ';'
;
```

function_call_statement:

```
identifier '(' arg_list ')' ';'
;
```

control_flow_statement:

```
if_statement  
| for_statement  
;
```

if_statement:

```
IF '(' expression ')' '{' statement_list '}' %prec  
LOWER_THAN_ELSE  
| IF '(' expression ')' '{' statement_list '}' ELSE '{'  
statement_list '}'  
;
```

for_statement:

```
FOR '(' variable_declaration ';' expression ';' expression ')' '{'  
statement_list '}'  
;
```

return_statement:

```
RETURN expression ';'
;
```



```
    ;

expression_statement:
    expression ';'
    ;

arg_list:
    /* empty */
    | expression
    | arg_list ',' expression
    ;

expression:
    '(' expression ')'
    | expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | identifier
    | constant
    ;

data_type:
    INT
    | FLOAT
    | STRING
    | BOOL
    ;

identifier:
    /* define identifier here */
    ;

constant:
    INT_CONSTANT
    | FLOAT_CONSTANT
    | STRING_CONSTANT
```

```

        | BOOL_CONSTANT
    ;

%%

/* Define any additional functions and global variables here */

/* Define the main function for testing */

int main(int argc, char *argv[]) {
    /* Declare any necessary variables and data structures here */

    yyparse();

    /* Clean up any dynamically allocated memory and exit */

    return 0;
}

```

Note that this is just an example of how you can create the `parser.y` file based on `hello.zon`. You may need to modify it to suit your specific needs.

Q: I created the `parser.y` and run `bison -d parser.y`, the output is show below, how to fixed it?

```

$ bison -d parser.y
parser.y:98.5-16: error: symbol 'INT_CONSTANT' is used, but is not
defined as a token and has no rules
  98 |         INT_CONSTANT
    |         ^~~~~~
parser.y:99.7-20: error: symbol 'FLOAT_CONSTANT' is used, but is
not defined as a token and has no rules
  99 |         | FLOAT_CONSTANT
    |         ^~~~~~
parser.y:100.7-21: error: symbol 'STRING_CONSTANT' is used, but is
not defined as a token and has no rules
 100 |         | STRING_CONSTANT
    |         ^~~~~~
parser.y:101.7-19: error: symbol 'BOOL_CONSTANT' is used, but is
not defined as a token and has no rules

```

```
101 |      | BOOL_CONSTANT  
    |      | ^~~~~~
```

The error message indicates that the tokens `INT_CONSTANT`, `FLOAT_CONSTANT`, `STRING_CONSTANT`, and `BOOL_CONSTANT` are used in your parser, but they are not defined in your lexer file `scanner.l`.

To fix this error, you need to add rules for these tokens in your lexer file. In `scanner.l`, you can add regular expressions to match these token types and return their corresponding token values to the parser. For example, to match integer constants, you can add a rule like this:

```
[0-9]+      { yylval.int_val = atoi(yytext); return INT_CONSTANT;  
}
```

This rule matches a sequence of one or more digits (`[0-9]+`) and sets the value of `yylval.int_val` to the integer representation of the matched text using `atoi()`. It then returns the token type `INT_CONSTANT` to the parser.

You can add similar rules for the other token types in your `scanner.l` file. After adding these rules, you need to regenerate the lexer and parser files using the following commands:

```
flex scanner.l  
bison -d parser.y
```

This should generate the lexer and parser files with the necessary token definitions.

Q: do I need to create `scanner.l` and `parser.y` based on the `hello.zon` at the same time? show me the `scanner.l` based on the `hello.zon`.

Yes, you need to create `scanner.l` and `parser.y` based on the `zon-grammar` at the same time.

Here's an example `scanner.l` based on the `hello.zon` file and the `zon-grammar` we created earlier:

```
%{  
#include "parser.tab.h"  
%}
```

DIGIT	[0-9]
LETTER	[A-Za-z]
WHITESPACE	[\t\n]

%%

"fn"	{ return FN; }
"var"	{ return VAR; }
"if"	{ return IF; }
"else"	{ return ELSE; }
"for"	{ return FOR; }
"return"	{ return RETURN; }

"int"	{ return INT_TYPE; }
"float"	{ return FLOAT_TYPE; }
"string"	{ return STRING_TYPE; }
"bool"	{ return BOOL_TYPE; }

```
{DIGIT}+      { yylval.integer = atoi(yytext); return  
INT_CONSTANT; }
```

```
{DIGIT}+\.{DIGIT}+ { yylval.real = atof(yytext); return  
FLOAT_CONSTANT; }
```

```
\("[^\\n]|(\\.\\.)*\\" { yylval.string = yytext; return  
STRING_CONSTANT; }
```

```
"true"|"false" { yylval.boolean = strcmp(yytext, "true") == 0 ? 1  
: 0; return BOOL_CONSTANT; }
```

"=="	{ return EQ; }
"!="	{ return NEQ; }
"<"	{ return LT; }
">"	{ return GT; }
"<="	{ return LTE; }
">="	{ return GTE; }

"+"	{ return PLUS; }
"-"	{ return MINUS; }
"*"	{ return MUL; }

```

"/"      { return DIV; }
%"      { return MOD; }

"="      { return ASSIGN; }
";"      { return SEMICOLON; }
","      { return COMMA; }
":"      { return COLON; }
"("      { return LPAREN; }
")"      { return RPAREN; }
"{"      { return LBRACE; }
"}"      { return RBRACE; }

{LETTER}({LETTER}|{DIGIT})* { yylval.identifier = strdup(yytext);
return IDENTIFIER; }

{WHITESPACE}+ /* ignore whitespace */
.             { printf("Invalid token: %s\n", yytext); }

%%

int yywrap(void) {
    return 1;
}

```

In this `scanner.l`, we define regular expressions for tokens such as keywords, identifiers, and constants. When a token is recognized, the corresponding `return` statement is used to pass the token type and any associated values to the parser. If an invalid token is encountered, an error message is printed.

Note that in order to use the token names defined in `parser.y`, we include `parser.tab.h` at the top of the file.

Q: what are output files of `flex scanner.l`? and output files of `bison -d parser.y`?

The `flex scanner.l` command generates a C source file named `lex.yy.c` that contains the implementation of the lexer. The `bison -d parser.y` command generates two files, `parser.tab.c` and

`parser.tab.h`. The `.c` file contains the implementation of the parser, and the `.h` file contains the declarations of various macros, data structures, and functions used by the parser.

Q: now, I got `scanner.l`, `lex.yy.c`, and `parser.y` after running `flex scanner.l` in the terminal. how to run bison to generate `parser.tab.c` and `parser.tab.h` files?

To generate the `parser.tab.c` and `parser.tab.h` files from the `parser.y` file, you can run the following command in the terminal:

```
bison -d parser.y
```

The `-d` option tells Bison to generate a header file (`parser.tab.h`) that declares the parser's symbols and their types. The parser code will be generated in `parser.tab.c`. Once this command is executed, you should have the `parser.tab.c` and `parser.tab.h` files in your working directory.

Q: I got the same errors by running `bison -d parser.y`

```
$ bison -d parser.y
parser.y:98.5-16: error: symbol 'INT_CONSTANT' is used, but is not
defined as a token and has no rules
  98 |         INT_CONSTANT
      |         ^~~~~~
parser.y:99.7-20: error: symbol 'FLOAT_CONSTANT' is used, but is
not defined as a token and has no rules
  99 |         | FLOAT_CONSTANT
      |         ^~~~~~
parser.y:100.7-21: error: symbol 'STRING_CONSTANT' is used, but is
not defined as a token and has no rules
 100 |         | STRING_CONSTANT
      |         ^~~~~~
parser.y:101.7-19: error: symbol 'BOOL_CONSTANT' is used, but is
not defined as a token and has no rules
 101 |         | BOOL_CONSTANT
      |         ^~~~~~
```

These errors indicate that the token names used in the parser rules are not defined as tokens in the lexer. You need to define these tokens in the lexer (`scanner.l`) using regular expressions or rules.

For example, to define `INT_CONSTANT` token, you can add the following rule to scanner.l:

```
[0-9]+      { yylval.ival = atoi(yytext); return INT_CONSTANT; }
```

Similarly, you can define `FLOAT_CONSTANT`, `STRING_CONSTANT`, and `BOOL_CONSTANT` tokens using appropriate regular expressions or rules.

After defining the tokens in the lexer, you need to regenerate the lexer using `flex scanner.l`. Then, you can run `bison -d parser.y` to generate `parser.tab.c` and `parser.tab.h` files.

Q: I have defined the `INT_CONSTANT` in the `sscanner.l` show below

```
DIGIT [0-9]
{DIGIT}+ { yylval.integer = atoi(yytext); return INT_CONSTANT; }
```

but it did not work.

It seems like you defined the `INT_CONSTANT` in the lexer (`scanner.l`), but it needs to be defined in the parser (`parser.y`) as a token. To do this, you can add the following code in the `%token` section of `parser.y`:

```
%token INT_CONSTANT
```

This will define `INT_CONSTANT` as a token in the parser and resolve the error you encountered.

So I added in `parser.y`

```
%token INT_CONSTANT FLOAT_CONSTANT STRING_CONSTANT BOOL_CONSTANT
```

Q: fixed the error but I got the following warning

```
parser.y:56.5-70: warning: token for %prec is not defined:
LOWER_THAN_ELSE [-Wother]
    56 |      IF '(' expression ')' '{' statement_list '}' %prec
      |      LOWER_THAN_ELSE
      |
      |
      | ^~~~~~
parser.y: warning: 33 shift/reduce conflicts [-Wconflicts-sr]
```

```
parser.y: warning: 2 reduce/reduce conflicts [-Wconflicts-rr]
parser.y: note: rerun with option '-Wcounterexamples' to generate
conflict counterexamples
```

can I ignore it?

Warnings should not be ignored as they may indicate potential issues in the grammar rules. The first warning indicates that the token `%prec LOWER_THAN_ELSE` used in the grammar rule for `if-else` statements is not defined. You need to define it as a precedence token in the `parser.y` file.

The second and third warnings indicate that there are shift/reduce and reduce/reduce conflicts in the grammar, which can lead to ambiguous parsing. It's recommended to resolve these conflicts to ensure the grammar is unambiguous. You can use the `-Wcounterexamples` option with Bison to generate conflict counterexamples and help you identify and resolve the conflicts.

Q: how to fix this error?

```
parser.y:56.5-70: warning: token for %prec is not defined:
LOWER_THAN_ELSE [-Wother]
  56 |      IF '(' expression ')' '{' statement_list '}' %prec
      |      LOWER_THAN_ELSE
      |
      ^~~~~~
```

The warning message indicates that the `%prec` token `LOWER_THAN_ELSE` used in the production rule for `IF` statement does not have a definition. To fix this, you need to define `LOWER_THAN_ELSE` as a token in your `scanner.l` file and make sure it is consistent with the definition in your `parser.y` file.

For example, in your `scanner.l` file, you could add the following definition for `LOWER_THAN_ELSE`:

```
"lower_than_else" { return LOWER_THAN_ELSE; }
```

Then, in your `parser.y` file, you can define the `%token` for `LOWER_THAN_ELSE`:


```
%token LOWER_THAN_ELSE
```

Make sure that the spelling and capitalization of the token name are consistent between the two files. After making these changes, regenerate the parser and lexer files and check if the warning has been resolved.

let's make the `if` condition support only `else` statement, so I remove the `LOWER_THAN_ELSE` from the `parser.y` file.

```
if_statement:
    IF '(' expression ')' '{' statement_list '}' %prec
    LOWER_THAN_ELSE
    | IF '(' expression ')' '{' statement_list '}' ELSE '{'
    statement_list '}'
    ;
```

to

```
if_statement:
    IF '(' expression ')' '{' statement_list '}' ELSE '{'
    statement_list '}'
    ;
```

And ignore warnings in `parser.y` to simplify the implementation

Q: nice, I have got `hello.zon`, `lex.yy.c`, `parser.tab.c`, `parser.tab.h`, `parser.y`, and `scanner.l` files in the project. as per our discussion on Step 9: compile the code, how to compile the code?

To compile the code, you need to first compile `parser.tab.c` and `lex.yy.c` using a C compiler, such as `gcc`, and link them together to create an executable program. Here are the steps to compile the code:

1. Compile `parser.tab.c` and `lex.yy.c` files using `gcc`:

```
gcc -c parser.tab.c
gcc -c lex.yy.c
```

2. Link the object files together to create an executable program:

```
gcc parser.tab.o lex.yy.o -o zon
```

3. Run the program:

```
./zon hello.zon
```

This will compile the `parser.tab.c` and `lex.yy.c` files into object files and then link them together to create an executable program called `zon`. You can then run the program on the `hello.zon` file using the command `./zon hello.zon`.

Q: how to fix the following errors when running `gcc -c parser.tab.c`

```
parser.tab.c: In function 'yyparse':
parser.tab.c:1096:16: warning: implicit declaration of function
'yylex' [-Wimplicit-function-declaration]
1096 |         yychar = yylex ();
      |                   ^~~~~
parser.tab.c:1231:7: warning: implicit declaration of function
'yerror'; did you mean 'yyerrok'? [-Wimplicit-function-
declaration]
1231 |         yerror (YY_("syntax error"));
      |         ^~~~~~
      |         yyerrok
```

The first warning about `yylex()` suggests that the compiler does not recognize the function. This is likely because we have not included the header file generated by Flex `scanner.h`. We need to include it in our `parser.tab.c` file, so add `#include "scanner.h"` at the top of `parser.tab.c`.

The second warning is related to the function `yerror()`, which is used to handle syntax errors. This function is not generated by Bison, so we need to define it ourselves. We can define a simple implementation that prints an error message to `stderr` and returns:

```
void yerror (const char *msg)
{
    fprintf (stderr, "%s\n", msg);
}
```

Add this implementation to the top of `parser.y`, before the first rule. Then re-run `bison -d parser.y` to generate a new `parser.tab.c` file, and run `gcc -c parser.tab.c` again.

Q: the `flex scanner.l` only create a `lex.yy.c` file, where does the `scanner.h` from?

The `scanner.h` file is typically generated by including the `%option header-file="scanner.h"` directive at the top of your `scanner.l` file. This directive tells `flex` to generate a header file named `scanner.h` that declares all of the functions, variables, and macros used in the lexer.

If you did not include the `%option header-file="scanner.h"` directive in your `scanner.l` file, you can manually create a header file and declare any necessary functions, variables, or macros used in the lexer. However, it's recommended to use the `%option header-file` directive to ensure that the header file stays in sync with the lexer.

I did the following steps to fix those errors:

1. added the a line at the top of `scanner.l` file and re-run `flex scanner.l` to generate the `scanner.h` file

```
%option header-file="scanner.h"
```

2. added the `yyerror` declaration at line number of 8 in `parser.y`

```
/* fixed: implicit declaration of function 'yyerror'; did you mean
'yyerrok'? [-Wimplicit-function-declaration] */
void yyerror (const char *msg)
{
    fprintf (stderr, "%s\n", msg);
}
```

3. re-run `bison -d parser.y` then added the `scanner.h` header at line number of 76 in `parser.tab.c` file

```
/* First part of user prologue. */
#line 1 "parser.y"

#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

#include "scanner.h"
```

Now, I can compile the `parser.tab.c` and get an object file called `parser.tab.o` but got errors on `lex.yy.c`

```
$ gcc -c parser.tab.c
$ gcc -c lex.yy.c
scanner.l: In function 'yylex':
scanner.l:20:10: error: 'INT_TYPE' undeclared (first use in this
function)
  20 | "int"          { return INT_TYPE; }
    |             ^~~~~~
scanner.l:20:10: note: each undeclared identifier is reported only
once for each function it appears in
scanner.l:21:10: error: 'FLOAT_TYPE' undeclared (first use in this
function)
  21 | "float"        { return FLOAT_TYPE; }
    |             ^~~~~~
scanner.l:22:10: error: 'STRING_TYPE' undeclared (first use in
this function)
  22 | "string"       { return STRING_TYPE; }
    |             ^~~~~~
scanner.l:23:10: error: 'BOOL_TYPE' undeclared (first use in this
function)
  23 | "bool"         { return BOOL_TYPE; }
    |             ^~~~~~
scanner.l:25:9: error: request for member 'integer' in something
not a structure or union
  25 | {DIGIT}+      { yylval.integer = atoi(yytext); return
INT_CONSTANT; }
    |             ^
scanner.l:26:9: error: request for member 'real' in something not
a structure or union
  26 | {DIGIT}+\\. {DIGIT}+ { yylval.real = atof(yytext); return
FLOAT_CONSTANT; }
    |             ^
scanner.l:27:9: error: request for member 'string' in something
not a structure or union
  27 | \"([^\n]|\\.)*\" { yylval.string = yytext; return
STRING_CONSTANT; }
```

```

      |           ^
scanner.l:28:9: error: request for member 'boolean' in something
not a structure or union
    28 | "true"|"false" { yylval.boolean = strcmp(yytext, "true")
== 0 ? 1 : 0; return BOOL_CONSTANT; }
      |           ^
scanner.l:30:10: error: 'EQ' undeclared (first use in this
function)
    30 | "=="          { return EQ; }
      |           ^~
scanner.l:31:10: error: 'NEQ' undeclared (first use in this
function)
    31 | "!="          { return NEQ; }
      |           ^~~
scanner.l:32:10: error: 'LT' undeclared (first use in this
function)
    32 | "<"          { return LT; }
      |           ^~
scanner.l:33:10: error: 'GT' undeclared (first use in this
function)
    33 | ">"          { return GT; }
      |           ^~
scanner.l:34:10: error: 'LTE' undeclared (first use in this
function)
    34 | "<="        { return LTE; }
      |           ^~~
scanner.l:35:10: error: 'GTE' undeclared (first use in this
function)
    35 | ">="        { return GTE; }
      |           ^~~
scanner.l:37:10: error: 'PLUS' undeclared (first use in this
function)
    37 | "+"          { return PLUS; }
      |           ^~~~
scanner.l:38:10: error: 'MINUS' undeclared (first use in this
function)
    38 | "-"          { return MINUS; }
      |           ^~~~~
scanner.l:39:10: error: 'MUL' undeclared (first use in this
function)
    39 | "*"          { return MUL; }
      |           ^~~
scanner.l:40:10: error: 'DIV' undeclared (first use in this
function)

```

```

40 | "/"                { return DIV; }
    |                ^~~
scanner.l:41:10: error: 'MOD' undeclared (first use in this
function)
41 | "%"                { return MOD; }
    |                ^~~
scanner.l:43:10: error: 'ASSIGN' undeclared (first use in this
function)
43 | "="                { return ASSIGN; }
    |                ^~~~~~
scanner.l:44:10: error: 'SEMICOLON' undeclared (first use in this
function)
44 | ";"                { return SEMICOLON; }
    |                ^~~~~~
scanner.l:45:10: error: 'COMMA' undeclared (first use in this
function)
45 | ","                { return COMMA; }
    |                ^~~~~
scanner.l:46:10: error: 'COLON' undeclared (first use in this
function)
46 | ":"                { return COLON; }
    |                ^~~~~
scanner.l:47:10: error: 'LPAREN' undeclared (first use in this
function)
47 | "("                { return LPAREN; }
    |                ^~~~~~
scanner.l:48:10: error: 'RPAREN' undeclared (first use in this
function)
48 | ")"                { return RPAREN; }
    |                ^~~~~~
scanner.l:49:10: error: 'LBRACE' undeclared (first use in this
function)
49 | "{"                { return LBRACE; }
    |                ^~~~~~
scanner.l:50:10: error: 'RBRACE' undeclared (first use in this
function)
50 | "}"                { return RBRACE; }
    |                ^~~~~~
scanner.l:52:9: error: request for member 'identifier' in
something not a structure or union
52 | {LETTER}({LETTER}|{DIGIT})* { yylval.identifier =
strdup(ytext); return IDENTIFIER; }
    |                ^
scanner.l:52:46: error: 'IDENTIFIER' undeclared (first use in this

```

```
function)
    52 | {LETTER}({LETTER}|{DIGIT})* { yylval.identifier =
      | strdup(yytext); return IDENTIFIER; }
      |
      | ^~~~~~
```

Q: I compiled `parser.tab.c` and got a `parser.tab.o` file but got a lot errors when running `gcc -c lex.yy.c`. how to fix errors shown below:

```
scanner.l: In function 'yylex':
scanner.l:20:10: error: 'INT_TYPE' undeclared (first use in this
function)
    20 | "int"          { return INT_TYPE; }
      |          ^~~~~~
scanner.l:20:10: note: each undeclared identifier is reported only
once for each function it appears in
scanner.l:21:10: error: 'FLOAT_TYPE' undeclared (first use in this
function)
    21 | "float"        { return FLOAT_TYPE; }
      |        ^~~~~~
scanner.l:22:10: error: 'STRING_TYPE' undeclared (first use in
this function)
    22 | "string"       { return STRING_TYPE; }
      |       ^~~~~~
scanner.l:23:10: error: 'BOOL_TYPE' undeclared (first use in this
function)
    23 | "bool"         { return BOOL_TYPE; }
      |         ^~~~~~
```

The errors indicate that the token types `INT_TYPE`, `FLOAT_TYPE`, `STRING_TYPE`, and `BOOL_TYPE` are not defined in the `scanner.l` file. You should define these token types in the `parser.y` file as follows:

```
%token <integer> INT_CONSTANT
%token <float> FLOAT_CONSTANT
%token <string> STRING_CONSTANT
%token <integer> BOOL_CONSTANT
%token INT_TYPE FLOAT_TYPE STRING_TYPE BOOL_TYPE
```

And then use them in the `scanner.l` file as follows:

```
"int"          { return INT_TYPE; }
"float"        { return FLOAT_TYPE; }
"string"       { return STRING_TYPE; }
"bool"         { return BOOL_TYPE; }
```

Make sure to re-run `flex scanner.l` and `bison -d parser.y` after making the changes, and then compile all source files with `gcc -c parser.tab.c lex.yy.c`.

Q: and how about these errors:

```
scanner.l:25:9: error: request for member 'integer' in something
not a structure or union
  25 | {DIGIT}+          { yylval.integer = atoi(yytext); return
INT_CONSTANT; }
      |               ^
scanner.l:26:9: error: request for member 'real' in something not
a structure or union
  26 | {DIGIT}+\.{DIGIT}+ { yylval.real = atof(yytext); return
FLOAT_CONSTANT; }
      |               ^
scanner.l:27:9: error: request for member 'string' in something
not a structure or union
  27 | \"([^\n] | (\\.))*\" { yylval.string = yytext; return
STRING_CONSTANT; }
      |               ^
scanner.l:28:9: error: request for member 'boolean' in something
not a structure or union
  28 | "true"|"false" { yylval.boolean = strcmp(yytext, "true")
== 0 ? 1 : 0; return BOOL_CONSTANT; }
      |               ^
```

These errors are occurring because the `yylval` union members are not defined in the scanner. You need to include the `parser.tab.h` file in the scanner to get access to the `yylval` union.

To fix these errors, add the following line to the top of `scanner.l`:

```
%{
#include "parser.tab.h"
%}
```


This will include the `parser.tab.h` file and make the `yylval` union members available to the scanner.

Q: this is the `scanner.l` file

```
%option header-file="scanner.h"

%{
#include "parser.tab.h"
%}

DIGIT      [0-9]
LETTER     [A-Za-z]
WHITESPACE [ \t\n]

%%

"fn"       { return FN; }
"var"      { return VAR; }
"if"       { return IF; }
"else"     { return ELSE; }
"for"      { return FOR; }
"return"   { return RETURN; }

"int"      { return INT_TYPE; }
"float"    { return FLOAT_TYPE; }
"string"   { return STRING_TYPE; }
"bool"     { return BOOL_TYPE; }

{DIGIT}+   { yylval.integer = atoi(yytext); return
INT_CONSTANT; }
{DIGIT}+\.{DIGIT}+ { yylval.real = atof(yytext); return
FLOAT_CONSTANT; }
\"([^\n\\]|(\\.\\.))*\" { yylval.string = yytext; return
STRING_CONSTANT; }
"true"|"false" { yylval.boolean = strcmp(yytext, "true") == 0 ? 1
: 0; return BOOL_CONSTANT; }

"=="      { return EQ; }
```

```

"!="      { return NEQ; }
"<"      { return LT; }
">"      { return GT; }
"<="     { return LTE; }
">="     { return GTE; }

"+"      { return PLUS; }
"-"      { return MINUS; }
"*"      { return MUL; }
"/"      { return DIV; }
%"       { return MOD; }

"="      { return ASSIGN; }
";"      { return SEMICOLON; }
","      { return COMMA; }
":"      { return COLON; }
"("      { return LPAREN; }
")"      { return RPAREN; }
"{"      { return LBRACE; }
"}"      { return RBRACE; }

{LETTER}({LETTER}|{DIGIT})* { yylval.identifier = strdup(yytext);
return IDENTIFIER; }

{WHITESPACE}+ /* ignore whitespace */
.             { printf("Invalid token: %s\n", yytext); }

%%

int yywrap(void) {
    return 1;
}

```

And this is the `parser.y` file

```

%{
#include <stdio.h>
#include <stdlib.h>

```

```

#include <string.h>

/* Declare any additional functions and global variables here */

/* fixed: implicit declaration of function 'yyerror'; did you mean
'yyerrok'? [-Wimplicit-function-declaration] */
void yyerror (const char *msg)
{
    fprintf (stderr, "%s\n", msg);
}

%}

/* Declare tokens here */
%token INT FLOAT STRING BOOL TRUE FALSE VAR FN RETURN FOR IN IF
ELSE

/* FIX: error: symbol 'INT_CONSTANT' is used, but is not defined
as a token and has no rules */
%token INT_CONSTANT FLOAT_CONSTANT STRING_CONSTANT BOOL_CONSTANT

%%

/* Define grammar rules here */

program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    variable_declaration
    | assignment_statement

```

```
| function_call_statement  
| control_flow_statement  
| return_statement  
| expression_statement  
;
```

variable_declaration:

```
VAR identifier ':' data_type '=' expression ';'
;
```

assignment_statement:

```
identifier '=' expression ';'
;
```

function_call_statement:

```
identifier '(' arg_list ')' ';'
;
```

control_flow_statement:

```
if_statement  
| for_statement  
;
```

if_statement:

```
IF '(' expression ')' '{' statement_list '}' ELSE '{'  
statement_list '}'  
;
```

for_statement:

```
FOR '(' variable_declaration ';' expression ';' expression ')'  
'{' statement_list '}'  
;
```

return_statement:

```
RETURN expression ';'
;
```

expression_statement:

```
    expression ';'
;

arg_list:
    /* empty */
    | expression
    | arg_list ',' expression
;

expression:
    '(' expression ')'
    | expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | identifier
    | constant
;

data_type:
    INT
    | FLOAT
    | STRING
    | BOOL
;

identifier:
    /* define identifier here */
;

constant:
    INT_CONSTANT
    | FLOAT_CONSTANT
    | STRING_CONSTANT
    | BOOL_CONSTANT
;
```

```
%%
```

```
/* Define any additional functions and global variables here */
```

```
/* Define the main function for testing */
```

```
int main(int argc, char *argv[]) {
```

```
    /* Declare any necessary variables and data structures here */
```

```
    yyparse();
```

```
    /* Clean up any dynamically allocated memory and exit */
```

```
    return 0;
```

```
}
```

is there any problem?

There seem to be a couple of issues with the code:

1. In `scanner.l`, the regular expression for string constants is not properly capturing the matched string. It should be `\"([^\\"\\n]|(\\\.))*\"` { `yylval.string = strdup(yytext); return STRING_CONSTANT; }` to allocate memory for the string value and assign it to `yylval.string`.
2. In `parser.y`, the `identifier` rule is empty and needs to be defined. It should be `identifier: IDENTIFIER ;`.
3. In `parser.y`, the `INT_CONSTANT`, `FLOAT_CONSTANT`, `STRING_CONSTANT`, and `BOOL_CONSTANT` tokens are used, but they are not declared as tokens in the `%token` section. They need to be added: `%token INT_CONSTANT FLOAT_CONSTANT STRING_CONSTANT BOOL_CONSTANT`.
4. In `parser.y`, the `yyerror` function is declared, but it is not defined. It needs to be implemented to handle syntax errors in the input.

Q: give me the correct `scanner.l` and `parser.y` in separately response. starting from `scanner.l`, I will ask for `parser.y` later.

Sure, here's the corrected `scanner.l` file:

```

%option header-file="scanner.h"

%{
#include "parser.tab.h"
#include <string.h>
%}

DIGIT          [0-9]
LETTER         [A-Za-z]
WHITESPACE     [ \t\n]

%%

"fn"           { return FN; }
"var"          { return VAR; }
"if"           { return IF; }
"else"         { return ELSE; }
"for"          { return FOR; }
"return"       { return RETURN; }

"int"          { return INT_TYPE; }
"float"        { return FLOAT_TYPE; }
"string"       { return STRING_TYPE; }
"bool"         { return BOOL_TYPE; }

{DIGIT}+      { yylval.integer = atoi(yytext); return
INT_CONSTANT; }
{DIGIT}+\. {DIGIT}+ { yylval.real = atof(yytext); return
FLOAT_CONSTANT; }
\"([^\n]|(\\\.))*\" { yylval.string = strdup(yytext); return
STRING_CONSTANT; }
"true"|"false" { yylval.boolean = strcmp(yytext, "true") == 0 ? 1
: 0; return BOOL_CONSTANT; }

"=="          { return EQ; }
"!="          { return NEQ; }
"<"          { return LT; }
">"          { return GT; }

```

```

"<="      { return LTE; }
">="      { return GTE; }

"+"      { return PLUS; }
"-"      { return MINUS; }
"*"      { return MUL; }
"/"      { return DIV; }
%"      { return MOD; }

"="      { return ASSIGN; }
";"      { return SEMICOLON; }
","      { return COMMA; }
":"      { return COLON; }
"("      { return LPAREN; }
")"      { return RPAREN; }
"{"      { return LBRACE; }
"}"      { return RBRACE; }

{LETTER}({LETTER}|{DIGIT})* { yylval.identifier = strdup(yytext);
return IDENTIFIER; }

{WHITESPACE}+ /* ignore whitespace */
.            { printf("Invalid token: %s\n", yytext); }

%%

int yywrap(void) {
    return 1;
}

```

And here's the corrected `parser.y` file:

```

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Declare any additional functions and global variables here */

```



```

void yyerror(const char *msg)
{
    fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}

%}

/* Declare tokens here */
%token INT_TYPE FLOAT_TYPE STRING_TYPE BOOL_TYPE
%token <integer> INT_CONSTANT
%token <real> FLOAT_CONSTANT
%token <string> STRING_CONSTANT
%token <boolean> BOOL_CONSTANT
%token IDENTIFIER
%token VAR FN RETURN FOR IN IF ELSE
%token EQ NEQ LT GT LTE GTE PLUS MINUS MUL DIV MOD ASSIGN
%token SEMICOLON COMMA COLON LPAREN RPAREN LBRACE RBRACE

%%

/* Define grammar rules here */

program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    variable_declaration
    | assignment_statement
    | function_call_statement
    | control_flow_statement

```

```

    | return_statement
    | expression_statement
    ;

variable_declaration:
    VAR identifier COLON data_type ASSIGN expression SEMICOLON
    ;

assignment_statement:
    identifier ASSIGN expression SEMICOLON
    ;

function_call_statement:

```

Q: show me the corrected `parser.y` file again, please

Sure, here is the corrected `parser.y` file:

```

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Declare any additional functions and global variables here */

/* fixed: implicit declaration of function 'yyerror'; did you mean
'yyerrok'? [-Wimplicit-function-declaration] */
void yyerror (const char *msg)
{
    fprintf (stderr, "%s\n", msg);
}

%}

/* Declare tokens here */
%token INT FLOAT STRING BOOL TRUE FALSE VAR FN RETURN FOR IN IF
ELSE
%token <integer> INT_CONSTANT

```

```
%token <real> FLOAT_CONSTANT
%token <string> STRING_CONSTANT
%token <boolean> BOOL_CONSTANT

%%

/* Define grammar rules here */

program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    variable_declaration
    | assignment_statement
    | function_call_statement
    | control_flow_statement
    | return_statement
    | expression_statement
    ;

variable_declaration:
    VAR identifier ':' data_type '=' expression ';'
    ;

assignment_statement:
    identifier '=' expression ';'
    ;

function_call_statement:
    identifier '(' arg_list ')' ';'
    ;
```

control_flow_statement:

- if_statement
- | for_statement

;

if_statement:

IF '(' expression ')' '{' statement_list '}' ELSE '{' statement_list '}'

;

for_statement:

FOR '(' variable_declaration ';' expression ';' expression ')' '{' statement_list '}'

;

return_statement:

RETURN expression ';' ;

expression_statement:

expression ';' ;

arg_list:

- /* empty */
- | expression
- | arg_list ',' expression

;

expression:

- '(' expression ')'
- | expression '+' expression
- | expression '-' expression
- | expression '*' expression
- | expression '/' expression
- | expression '%' expression
- | identifier
- | constant

```

;

data_type:
    INT
    | FLOAT
    | STRING
    | BOOL
;

identifier:
    /* define identifier here */
;

constant:
    INT_CONSTANT
    | FLOAT_CONSTANT
    | STRING_CONSTANT
    | BOOL_CONSTANT
;

%%

/* Define any additional functions and global variables here */

/* Define the main function for testing */

int main(int argc, char *argv[]) {
    /* Declare any necessary variables and data structures here */

    yyparse();

    /* Clean up any dynamically allocated memory and exit */

    return 0;
}

```

Note that the `%token` directives now include type information for each constant token. Also, the `identifier` non-terminal rule is still empty and

needs to be defined.

Re-generate files via Flex and Bison:

1. run flex and bison

```
$ rm lex.yy.c parser.tab.* scanner.h
$ ls
hello.zon  parser.y  scanner.l
$ flex scanner.l
$ bison -d parser.y
parser.y: warning: 33 shift/reduce conflicts [-Wconflicts-sr]
parser.y: warning: 2 reduce/reduce conflicts [-Wconflicts-rr]
parser.y: note: rerun with option '-Wcounterexamples' to generate
conflict counterexamples
$ ls
hello.zon  lex.yy.c  parser.tab.c  parser.tab.h  parser.y
scanner.h  scanner.l
```

2. added the `scanner.h` header at line number of 76 in `parser.tab.c` file

3. compile lexer and parser

```
$ gcc -c parser.tab.c
In file included from parser.tab.c:109:
parser.tab.h:81:8: error: unknown type name 'YYSTYPE'
   81 | extern YYSTYPE yylval;
      |           ^~~~~~
parser.tab.c:436:3: error: unknown type name 'YYSTYPE'
   436 | {
      |   ^
parser.tab.c:920:37: error: unknown type name 'YYSTYPE'
   920 | static void
      |                                     ^
parser.tab.c:329:5: error: expected end of line before 'push'
   329 | # else
      |     ^~
parser.tab.c:927:3: note: in expansion of macro
'YY_IGNORE_MAYBE_UNINITIALIZED_BEGIN'
   927 |     YY_SYMBOL_PRINT (yymsg, yykind, yyvaluep, yylocationp);
      |     ^~~~~~
parser.tab.c:330:5: error: expected end of line before 'ignored'
   330 | # define YY_IGNORE_MAYBE_UNINITIALIZED_BEGIN
      | \
```

```

|      ^~~~~~
parser.tab.c:927:3: note: in expansion of macro
‘YY_IGNORE_MAYBE_UNINITIALIZED_BEGIN’
  927 |     YY_SYMBOL_PRINT (yymmsg, yykind, yyvaluep, yylocationp);
|      ^~~~~~
parser.tab.c:331:5: error: expected end of line before ‘ignored’
  331 |     _Pragma ("GCC diagnostic push")
|
\
|      ^~~~~~
parser.tab.c:927:3: note: in expansion of macro
‘YY_IGNORE_MAYBE_UNINITIALIZED_BEGIN’
  927 |     YY_SYMBOL_PRINT (yymmsg, yykind, yyvaluep, yylocationp);
|      ^~~~~~
parser.tab.c:334:5: error: expected end of line before ‘pop’
  334 | # endif
|      ^~~
parser.tab.c:929:3: note: in expansion of macro
‘YY_IGNORE_MAYBE_UNINITIALIZED_END’
  929 |     YY_IGNORE_MAYBE_UNINITIALIZED_BEGIN
|      ^~~~~~
parser.tab.c:937:1: error: unknown type name ‘YYSTYPE’
  937 |
|      ^
parser.tab.c: In function ‘yyparse’:
parser.tab.c:967:5: error: unknown type name ‘YYSTYPE’
  967 |
|      ^
parser.tab.c:968:5: error: unknown type name ‘YYSTYPE’
  968 |     /* The semantic value stack: array, bottom, top. */
|      ^~~~~~
parser.tab.c:969:5: error: unknown type name ‘YYSTYPE’
  969 |     YYSTYPE yyvsa[YYINITDEPTH];
|      ^~~~~~
parser.tab.c:978:3: error: unknown type name ‘YYSTYPE’
  978 |     /* The variables used to return semantic value and
location from the
|      ^~~~~~
parser.tab.c:445:48: error: ‘YYSTYPE’ undeclared (first use in
this function)
  445 |     N elements. */
|
|      ^
parser.tab.c:93:40: note: in definition of macro ‘YY_CAST’
  93 | # define YY_REINTERPRET_CAST(Type, Val)
reinterpret_cast<Type> (Val)

```

```

|                                     ^~~
parser.tab.c:1054:35: note: in expansion of macro 'YY_CAST'
1054 |         union yyallocl *yyptr =
|                                     ^
parser.tab.c:276:21: note: in expansion of macro 'YY_CAST'
276 |         : YY_CAST (YYSIZE_T, -1))
|         ^~~~~~
parser.tab.c:445:38: note: in expansion of macro 'YYSIZEOF'
445 |         N elements. */
|                                     ^
parser.tab.c:1054:54: note: in expansion of macro 'YYSTACK_BYTES'
1054 |         union yyallocl *yyptr =
|                                     ^
parser.tab.c:445:48: note: each undeclared identifier is reported
only once for each function it appears in
445 |         N elements. */
|                                     ^
parser.tab.c:93:40: note: in definition of macro 'YY_CAST'
93 | # define YY_REINTERPRET_CAST(Type, Val)
reinterpret_cast<Type> (Val)
|                                     ^~~
parser.tab.c:1054:35: note: in expansion of macro 'YY_CAST'
1054 |         union yyallocl *yyptr =
|                                     ^
parser.tab.c:276:21: note: in expansion of macro 'YY_CAST'
276 |         : YY_CAST (YYSIZE_T, -1))
|         ^~~~~~
parser.tab.c:445:38: note: in expansion of macro 'YYSIZEOF'
445 |         N elements. */
|                                     ^
parser.tab.c:1054:54: note: in expansion of macro 'YYSTACK_BYTES'
1054 |         union yyallocl *yyptr =
|                                     ^
parser.tab.c:1254:11: warning: implicit declaration of function
'yydestruct' [-Wimplicit-function-declaration]
1254 |         else
|         ^

```

Q: define the `identifier` in the `parser.y`

Sure, you can define `identifier` in the `parser.y` file using the following syntax:


```
identifier:
    IDENTIFIER
    ;
```

You can add this definition to the bottom of the file, after the `constant` rule. The `IDENTIFIER` token will be recognized by the lexer and passed to the parser as a terminal symbol.

Q: got this error

```
$ bison -d parser.y
parser.y:105.5-14: error: symbol 'IDENTIFIER' is used, but is not
defined as a token and has no rules
105 |      IDENTIFIER
    |      ^~~~~~
```

You need to add the `IDENTIFIER` token to the `%token` section in your parser.y file:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Declare any additional functions and global variables here */

/* fixed: implicit declaration of function 'yyerror'; did you mean
'yyerrok'? [-Wimplicit-function-declaration] */
void yyerror (const char *msg)
{
    fprintf (stderr, "%s\n", msg);
}

%}

/* Declare tokens here */
%token INT FLOAT STRING BOOL TRUE FALSE VAR FN RETURN FOR IN IF
ELSE
```

```

%token IDENTIFIER /* Add this line */

/* FIX: error: symbol 'INT_CONSTANT' is used, but is not defined
as a token and has no rules */
%token INT_CONSTANT FLOAT_CONSTANT STRING_CONSTANT BOOL_CONSTANT

%%

/* Define grammar rules here */

program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    variable_declaration
    | assignment_statement
    | function_call_statement
    | control_flow_statement
    | return_statement
    | expression_statement
    ;

variable_declaration:
    VAR identifier ':' data_type '=' expression ';'
    ;

assignment_statement:
    identifier '=' expression ';'
    ;

function_call_statement:
    identifier '(' arg_list ')' ';'

```

```
    ;

control_flow_statement:
    if_statement
    | for_statement
    ;

if_statement:
    IF '(' expression ')' '{' statement_list '}' ELSE '{'
statement_list '}'
    ;

for_statement:
    FOR '(' variable_declaration ';' expression ';' expression ')'
'{' statement_list '}'
    ;

return_statement:
    RETURN expression ';'
    ;

expression_statement:
    expression ';'
    ;

arg_list:
    /* empty */
    | expression
    | arg_list ',' expression
    ;

expression:
    '(' expression ')'
    | expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
```

```

    | identifier
    | constant
    ;

data_type:
    INT
    | FLOAT
    | STRING
    | BOOL
    ;

identifier:
    IDENTIFIER    /* Add this line */
    ;

constant:
    INT_CONSTANT
    | FLOAT_CONSTANT
    | STRING_CONSTANT
    | BOOL_CONSTANT
    ;

%%

/* Define any additional functions and global variables here */

/* Define the main function for testing */

int main(int argc, char *argv[]) {
    /* Declare any necessary variables and data structures here */

    yyparse();

    /* Clean up any dynamically allocated memory and exit */

    return 0;
}

```

Update the `parser.y` as ChatGPT provided

```
$ gcc -c parser.tab.c
parser.tab.c: In function 'yyparse':
parser.tab.c:1098:16: warning: implicit declaration of function
'yylex' [-Wimplicit-function-declaration]
1098 |         yychar = yylex ();
      |                  ^~~~~
```

Added `#include "scanner.h"` to fix the error in `parser.tab.c`:

```
$ bison -d parser.y
parser.y: warning: 25 shift/reduce conflicts [-Wconflicts-sr]
parser.y: note: rerun with option '-Wcounterexamples' to generate
conflict counterexamples
$ gcc -c parser.tab.c
$ gcc -c lex.yy.c
scanner.l: In function 'yylex':
scanner.l:21:10: error: 'INT_TYPE' undeclared (first use in this
function)
  21 | "int"          { return INT_TYPE; }
      |          ^~~~~~
scanner.l:21:10: note: each undeclared identifier is reported only
once for each function it appears in
scanner.l:22:10: error: 'FLOAT_TYPE' undeclared (first use in this
function)
  22 | "float"        { return FLOAT_TYPE; }
      |        ^~~~~~
scanner.l:23:10: error: 'STRING_TYPE' undeclared (first use in
this function)
  23 | "string"       { return STRING_TYPE; }
      |       ^~~~~~
scanner.l:24:10: error: 'BOOL_TYPE' undeclared (first use in this
function)
  24 | "bool"         { return BOOL_TYPE; }
      |         ^~~~~~
```

To fix this, find and replace `INT`, `FLOAT`, `STRING`, `BOOL` in `parser.y`:

```
%token INT_TYPE FLOAT_TYPE STRING_TYPE BOOL_TYPE

data_type:
```

```
INT_TYPE
| FLOAT_TYPE
| STRING_TYPE
| BOOL_TYPE
;
```

But still has some errors.

Q: this is the `parser.y` file

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Declare any additional functions and global variables here */

/* fixed: implicit declaration of function 'yyerror'; did you mean
'yyerrok'? [-Wimplicit-function-declaration] */
void yyerror (const char *msg)
{
    fprintf (stderr, "%s\n", msg);
}

%}

/* Declare tokens here */
%token INT_TYPE FLOAT_TYPE STRING_TYPE BOOL_TYPE TRUE FALSE VAR FN
RETURN FOR IN IF ELSE
%token IDENTIFIER /* Add this line */

/* FIX: error: symbol 'INT_CONSTANT' is used, but is not defined
as a token and has no rules */
%token INT_CONSTANT FLOAT_CONSTANT STRING_CONSTANT BOOL_CONSTANT

%%

/* Define grammar rules here */
```

```
program:
    statement_list
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    variable_declaration
    | assignment_statement
    | function_call_statement
    | control_flow_statement
    | return_statement
    | expression_statement
    ;

variable_declaration:
    VAR identifier ':' data_type '=' expression ';'
    ;

assignment_statement:
    identifier '=' expression ';'
    ;

function_call_statement:
    identifier '(' arg_list ')' ';'
    ;

control_flow_statement:
    if_statement
    | for_statement
    ;

if_statement:
    IF '(' expression ')' '{' statement_list '}' ELSE '{'
```

```
statement_list '}'
```

```
;
```

```
for_statement:
```

```
    FOR '(' variable_declaration ';' expression ';' expression ')'
    '{' statement_list '}'
```

```
;
```

```
return_statement:
```

```
    RETURN expression ';'
;
```

```
expression_statement:
```

```
    expression ';'
;
```

```
arg_list:
```

```
    /* empty */
```

```
    | expression
```

```
    | arg_list ',' expression
```

```
;
```

```
expression:
```

```
    '(' expression ')'
```

```
    | expression '+' expression
```

```
    | expression '-' expression
```

```
    | expression '*' expression
```

```
    | expression '/' expression
```

```
    | expression '%' expression
```

```
    | identifier
```

```
    | constant
```

```
;
```

```
data_type:
```

```
    INT_TYPE
```

```
    | FLOAT_TYPE
```

```
    | STRING_TYPE
```

```
    | BOOL_TYPE
```



```

;

identifier:
    IDENTIFIER    /* Add this line */
;

constant:
    INT_CONSTANT
    | FLOAT_CONSTANT
    | STRING_CONSTANT
    | BOOL_CONSTANT
;

%%

/* Define any additional functions and global variables here */

/* Define the main function for testing */

int main(int argc, char *argv[]) {
    /* Declare any necessary variables and data structures here */

    yyparse();

    /* Clean up any dynamically allocated memory and exit */

    return 0;
}

```

got errors when running `gcc -c lex.yy.c` in the terminal, is the command correct? how to fix these errors?

```

$ gcc -c lex.yy.c
scanner.l: In function 'yylex':
scanner.l:26:9: error: request for member 'integer' in something
not a structure or union
    26 | {DIGIT}+          { yylval.integer = atoi(yytext); return
      | ~~~~~^
INT_CONSTANT; }

```

```
| ^
scanner.l:27:9: error: request for member 'real' in something not
a structure or union
```

```
27 | {DIGIT}+\\. {DIGIT}+ { yylval.real = atof(yytext); return
FLOAT_CONSTANT; }
```

```
| ^
scanner.l:28:9: error: request for member 'string' in something
not a structure or union
```

```
28 | \"([^\n]|\\.)*\" { yylval.string = strdup(yytext);
return STRING_CONSTANT; }
```

```
| ^
scanner.l:29:9: error: request for member 'boolean' in something
not a structure or union
```

```
29 | "true"|"false" { yylval.boolean = strcmp(yytext, "true")
== 0 ? 1 : 0; return BOOL_CONSTANT; }
```

```
| ^
scanner.l:31:10: error: 'EQ' undeclared (first use in this
function)
```

```
31 | "==" { return EQ; }
```

```
| ^~
scanner.l:31:10: note: each undeclared identifier is reported only
once for each function it appears in
```

```
scanner.l:32:10: error: 'NEQ' undeclared (first use in this
function)
```

```
32 | "!=" { return NEQ; }
```

```
| ^~~
scanner.l:33:10: error: 'LT' undeclared (first use in this
function)
```

```
33 | "<" { return LT; }
```

```
| ^~
scanner.l:34:10: error: 'GT' undeclared (first use in this
function)
```

```
34 | ">" { return GT; }
```

```
| ^~
scanner.l:35:10: error: 'LTE' undeclared (first use in this
function)
```

```
35 | "<=" { return LTE; }
```

```
| ^~~
```

scanner.l:36:10: error: 'GTE' undeclared (first use in this function)

```
36 | ">="          { return GTE; }  
    |             ^~~
```

scanner.l:38:10: error: 'PLUS' undeclared (first use in this function)

```
38 | "+"          { return PLUS; }  
    |             ^~~~
```

scanner.l:39:10: error: 'MINUS' undeclared (first use in this function)

```
39 | "-"          { return MINUS; }  
    |             ^~~~~
```

scanner.l:40:10: error: 'MUL' undeclared (first use in this function)

```
40 | "*"          { return MUL; }  
    |             ^~~
```

scanner.l:41:10: error: 'DIV' undeclared (first use in this function)

```
41 | "/"          { return DIV; }  
    |             ^~~
```

scanner.l:42:10: error: 'MOD' undeclared (first use in this function)

```
42 | "%"          { return MOD; }  
    |             ^~~
```

scanner.l:44:10: error: 'ASSIGN' undeclared (first use in this function)

```
44 | "="          { return ASSIGN; }  
    |             ^~~~~~
```

scanner.l:45:10: error: 'SEMICOLON' undeclared (first use in this function)

```
45 | ";"          { return SEMICOLON; }  
    |             ^~~~~~
```

scanner.l:46:10: error: 'COMMA' undeclared (first use in this function)

```
46 | ","          { return COMMA; }  
    |             ^~~~~
```

scanner.l:47:10: error: 'COLON' undeclared (first use in this function)

```

47 | ":"                { return COLON; }
    |                ^~~~~~
scanner.l:48:10: error: 'LPAREN' undeclared (first use in this
function)
48 | "("                { return LPAREN; }
    |                ^~~~~~
scanner.l:49:10: error: 'RPAREN' undeclared (first use in this
function)
49 | ")"                { return RPAREN; }
    |                ^~~~~~
scanner.l:50:10: error: 'LBRACE' undeclared (first use in this
function)
50 | "{"                { return LBRACE; }
    |                ^~~~~~
scanner.l:51:10: error: 'RBRACE' undeclared (first use in this
function)
51 | "}"                { return RBRACE; }
    |                ^~~~~~
scanner.l:53:9: error: request for member 'identifier' in
something not a structure or union
53 | {LETTER}({LETTER}|{DIGIT})* { yylval.identifier =
    |                                     ^
    strdup(yytext); return IDENTIFIER; }

```

The command `gcc -c lex.yy.c` is correct for compiling the Lex-generated scanner. However, it seems that there are errors in the scanner definition file `scanner.l` that need to be fixed before compiling.

The errors indicate that the scanner is trying to access members of `yylval`, which is not defined as a structure or union. This suggests that the `%union` declaration in `parser.y` may be missing or incorrect.

Additionally, there are errors related to undefined tokens, such as `EQ`, `NEQ`, `LT`, `GT`, `LTE`, `GTE`, `PLUS`, `MINUS`, and `MUL`. These tokens should be defined in the parser file using `%token`.

To fix the errors, try adding the following `%union` declaration to `parser.y` before the `%token` declarations:

```

%union {
    int integer;

```

```
float real;
char* string;
int boolean;
}
```

Then, add the missing tokens to the `%token` declarations, like this:

```
%token INT_CONSTANT FLOAT_CONSTANT STRING_CONSTANT BOOL_CONSTANT
EQ NEQ LT GT LTE GTE PLUS MINUS MUL
```

Finally, re-run the command `flex scanner.l` to regenerate the `lex.yy.c` file and then compile it using `gcc -c lex.yy.c`.

To fix this, add missing tokens in `parser.y`:

```
%token DIV MOD ASSIGN SEMICOLON COMMA COLON LPAREN RPAREN LBRACE
RBRACE
```

re-compile

```
$ bison -d parser.y
parser.y: warning: 25 shift/reduce conflicts [-Wconflicts-sr]
parser.y: note: rerun with option '-Wcounterexamples' to generate
conflict counterexamples
# add #include "scanner.h" into parser.tab.c
$ vi parser.tab.c
$ gcc -c parser.tab.c
$ gcc -c lex.yy.c
scanner.l: In function 'yylex':
scanner.l:53:9: error: 'YYSTYPE' has no member named 'identifier'
  53 | {LETTER}{LETTER}{DIGIT}* { yylval.identifier =
    | ^
```

Q: the `parser.y` is

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

/* Declare any additional functions and global variables here */

/* fixed: implicit declaration of function 'yyerror'; did you mean
'yyerrok'? [-Wimplicit-function-declaration] */
void yyerror (const char *msg)
{
    fprintf (stderr, "%s\n", msg);
}

%}

%union {
    int integer;
    float real;
    char* string;
    int boolean;
}

%token EQ NEQ LT GT LTE GTE PLUS MINUS MUL
%token DIV MOD ASSIGN SEMICOLON COMMA COLON LPAREN RPAREN LBRACE
RBRACE

/* Declare tokens here */
%token INT_TYPE FLOAT_TYPE STRING_TYPE BOOL_TYPE TRUE FALSE VAR FN
RETURN FOR IN IF ELSE
%token IDENTIFIER /* Add this line */

/* FIX: error: symbol 'INT_CONSTANT' is used, but is not defined
as a token and has no rules */
%token INT_CONSTANT FLOAT_CONSTANT STRING_CONSTANT BOOL_CONSTANT

%%

/* Define grammar rules here */

program:
    statement_list

```

```
    ;

statement_list:
    statement
    | statement_list statement
    ;

statement:
    variable_declaration
    | assignment_statement
    | function_call_statement
    | control_flow_statement
    | return_statement
    | expression_statement
    ;

variable_declaration:
    VAR identifier ':' data_type '=' expression ';'
    ;

assignment_statement:
    identifier '=' expression ';'
    ;

function_call_statement:
    identifier '(' arg_list ')' ';'
    ;

control_flow_statement:
    if_statement
    | for_statement
    ;

if_statement:
    IF '(' expression ')' '{' statement_list '}' ELSE '{'
statement_list '}'
    ;
```

```
for_statement:
    FOR '(' variable_declaration ';' expression ';' expression ')'
    '{' statement_list '}'
    ;

return_statement:
    RETURN expression ';'
    ;

expression_statement:
    expression ';'
    ;

arg_list:
    /* empty */
    | expression
    | arg_list ',' expression
    ;

expression:
    '(' expression ')'
    | expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | identifier
    | constant
    ;

data_type:
    INT_TYPE
    | FLOAT_TYPE
    | STRING_TYPE
    | BOOL_TYPE
    ;

identifier:
```



```

    IDENTIFIER    /* Add this line */
    ;

constant:
    INT_CONSTANT
    | FLOAT_CONSTANT
    | STRING_CONSTANT
    | BOOL_CONSTANT
    ;

%%

/* Define any additional functions and global variables here */

/* Define the main function for testing */

int main(int argc, char *argv[]) {
    /* Declare any necessary variables and data structures here */

    yyparse();

    /* Clean up any dynamically allocated memory and exit */

    return 0;
}

```

And and `scanner.l` file is

```

%option header-file="scanner.h"

%{
#include "parser.tab.h"
#include <string.h>
%}

DIGIT      [0-9]
LETTER     [A-Za-z]
WHITESPACE [ \t\n]

```

%%

```
"fn"          { return FN; }
"var"         { return VAR; }
"if"          { return IF; }
"else"        { return ELSE; }
"for"         { return FOR; }
"return"      { return RETURN; }
```

```
"int"         { return INT_TYPE; }
"float"       { return FLOAT_TYPE; }
"string"      { return STRING_TYPE; }
"bool"        { return BOOL_TYPE; }
```

```
{DIGIT}+      { yylval.integer = atoi(yytext); return
INT_CONSTANT; }
```

```
{DIGIT}+\. {DIGIT}+ { yylval.real = atof(yytext); return
FLOAT_CONSTANT; }
```

```
\"([^\n]|(\\.))*\" { yylval.string = strdup(yytext); return
STRING_CONSTANT; }
```

```
"true"|"false" { yylval.boolean = strcmp(yytext, "true") == 0 ? 1
: 0; return BOOL_CONSTANT; }
```

```
"=="         { return EQ; }
"!="         { return NEQ; }
"<"          { return LT; }
">"          { return GT; }
"<="         { return LTE; }
">="         { return GTE; }
```

```
"+"          { return PLUS; }
"-"          { return MINUS; }
"*"          { return MUL; }
"/"          { return DIV; }
"%"          { return MOD; }
```

```
"="          { return ASSIGN; }
```

I got an error:

```
scanner.l: In function ‘yylex’:  
scanner.l:53:9: error: ‘YYSTYPE’ has no member named ‘identifier’  
   53 | {LETTER}({LETTER}|{DIGIT})* { yylval.identifier =  
      | ^
```

how to fix it?

Update and rebuild the `parser.tab.c` and `lex.yy.c`

```
hello.zon lex.yy.c lex.yy.o parser.tab.c parser.tab.h  
parser.tab.o parser.y scanner.h scanner.l
```

Q: nice, I got `hello.zon`, `lex.yy.c`, `lex.yy.o`, `parser.tab.c`, `parser.tab.h`, `parser.tab.o`, `parser.y`, `scanner.h`, and `scanner.l` files in the project. the next is compile the `hello.zon` and run the output file, right? show me the command in the terminal to compile the `hello.zon`.

To compile the `hello.zon` file using the `parser.y` and `scanner.l` files, you will need to follow these steps:

1. Generate the lexical analyzer using `flex` by running the following command in the terminal:

```
flex scanner.l
```

This will generate a `lex.yy.c` file.

2. Generate the parser using `bison` by running the following command in the terminal:

```
bison -d parser.y
```

This will generate `parser.tab.c` and `parser.tab.h` files.

3. Compile the generated files using `gcc` by running the following command in the terminal:

```
gcc lex.yy.c parser.tab.c -o hello
```

This will compile the `lex.yy.c` and `parser.tab.c` files and generate an executable named `hello`.

4. Finally, run the `hello` executable by running the following command in the terminal:

```
./hello hello.zon
```

This will run the `hello` executable with the `hello.zon` file as input.

Q: what is the output of `./hello hello.zon`?

The output of `./hello hello.zon` depends on the contents of the `hello.zon` file and the implementation of the `hello` program. Without knowing the specifics of your program and the input file, I cannot accurately predict the output.

However, based on the name of your program, it is likely that it will output some sort of greeting or message to the user. You can try running the command yourself to see what the output is.

Q: the terminal is hang, and didn't output anything, how to debug the program?

```
$ gcc parser.tab.c lex.yy.c -o hello
$ ./hello hello.zon
^C
```

If the terminal is hanging and not producing any output, it's possible that the program is stuck in an infinite loop or waiting for user input. You can try interrupting the program by pressing Ctrl-C in the terminal. This will send a SIGINT signal to the running process and terminate it. If that doesn't work, you can use a debugger such as gdb to step through the program and see where it is getting stuck. To use gdb, first compile your program with debugging symbols using the `-g` flag:

```
$ gcc -g parser.tab.c lex.yy.c -o hello
```

Then run the program with gdb:

```
$ gdb ./hello
```

This will start gdb and load your program. You can then use gdb commands to step through the program and see what's happening. For example, you can use the `run` command to start the program, `break` to set breakpoints, and `step` to step through the program one line at a time. You can also use `print` to print the value of variables and `backtrace` to see where the program is currently executing. Once you've identified where the program is getting stuck, you can use that information to fix the bug in your code.

Q: how to test parser and lexer?

There are different approaches to testing a parser and lexer, but one common method is to use a set of input files that cover various scenarios and edge cases, and check if the output is correct. Here are some steps you can follow:

1. Create a set of input files that represent different cases and edge cases for the language you are parsing. For example, if you are parsing a language that supports arithmetic expressions, you might create input files that test expressions with different operators and operands, different levels of operator precedence, parentheses, etc.
2. Write a test script that runs the parser and lexer on each input file and compares the output against the expected output. The expected output can be hardcoded in the test script or stored in separate files.
3. Use a testing framework, such as `unittest` in Python or `JUnit` in Java, to organize your tests and report the results.
4. Run the test script and analyze the results. If a test fails, debug the parser and lexer by looking at the generated parse trees and tokens, and compare them against the expected output. You can also use debugging tools, such as `gdb` or `lldb`, to step through the code and inspect the values of variables and memory.
5. Once all tests pass, you can be confident that the parser and lexer are working as expected. You can also add more tests as you develop the language or make changes to the parser and lexer.

Q: show me the `hello.zon` file

I'm sorry, as an AI language model, I don't have access to your system files. Could you please provide the content of the `hello.zon` file?