九 尾 狐 の 語

K Language

Polymorphism in a C-like programming language

Diogo César Ferreira
11/0027931

University of Brasília

November 18, 2019

1 The language

1.1 Introduction

Programming languages are languages that can be converted into sets of instructions to be executed by a computer. This conversion process is called translation (or compilation) and is done by software known as compilers. As programming languages evolved, we observe an increasing variety of abstractions to accommodate for new programming paradigms and techniques, which also bring them closer to the domain of the problems they are set out to solve and away from the specificities of architecture set implementations [1].

One such abstraction is polymorphism. In typed languages, symbols (named identifiers for entities such as variables or functions) are constrained by their data type. If polymorphism is available, we are able to interact with different data types through a single interface, for instance, by allowing multiple types to be assigned to a given symbol. Polymorphism improves code readability and helps keeping the namespace clean by allowing functions to be identified by their expected behavior rather than contrived by the types of their arguments. Polymorphism is also one of the key features in object-oriented programming [1, 4].

Aside from the possibility to perform arithmetical operations over a few different data types (integers, floating point and pointers), the C programming language does not support polymorphism. In this work we seek to provide polymorphism in the form of function overloading - where polymorphic functions may have multiple definitions according to their argument types - to a simplified subset of the C programming language. The grammar of the language is presented in the following subsection.

1.2 Lexical features

The following table summarizes what word patterns are part of the language we are designing. These patterns should be recognized by the scanner, and therefore are provided as an input file for the FLEX [5] scanner generator (see attached file “language.l”):

Data types	void	char	int	float			
Keywords	if	else	do	while	return		
Arithmetical operators	+	−	*	/	%	++	−−
Comparison operators	<	>	<=	>=	==	!=	
Logical operators	&&	||	!				
Assignment operator	=						
Character/string delimiters	'	"					
Array indexers	[	]					
Scope delims./init lists	{	}					
End of statement mark	;						
Function args/calls	(	)					
List element separator	,						
Comment delimiters	/*	*/	//				

Character constants	text delimited by single quotation marks (')
String literals	text delimited by double quotation marks (")
Integer decimal constants	e.g. 1, 2, 100, -5, +7
Integer hexadecimal constants	e.g. 0x01, -0xA, +0x123
Floating-point constants	g.g. 0.0, -1.234, +3.14
Comments and white space	are ignored.

1.3 Language syntax

Our language's grammar contains elements taken from the one presented by Harbison III and Steele [3], which is a compilation from the many versions of the C grammar specified by the ISO C Standard along the years. Our grammar follows the general structure of the C language's grammar. We aim to provide a simplified version of the C language with support for polymorphic function definitions, that is, which allows the programmer to define multiple function with the same name but different argument types. The following grammar is also provided as an input file for the GNU Bison [2] parser generator (see attached file "language.y"). Striken-through rules are the ones that have been removed or changed from the previous version of our grammar.

1. start → declaration-list

2. declaration-list → declaration
   | declaration-list declaration

3. declaration → ~~function-declarator ;~~
   | ~~function-declarator compound-statement~~
   | function-definition
   | init-declarator **;**

4. init-declarator → ~~declarator~~
   | ~~declarator = initializer~~
   | type **IDENTIFIER**
   | type **IDENTIFIER** = assignment-expression
   | type **IDENTIFIER** [assignment-expression]

| type **IDENTIFIER** [ ] = **{** initializer-list **}**
| type **IDENTIFIER** [ ] = **{** initializer-list **,** **}**
| type **IDENTIFIER** [ ] = **STRING-LITERAL**

5. ~~declarator~~ → ~~type **IDENTIFIER**~~
| ~~type **IDENTIFIER** [ ]~~
| ~~type **IDENTIFIER** [assignment-expression]~~

6. ~~initializer~~ → ~~assignment-expression~~
| ~~**{** initializer-list **}**~~
| ~~**{** initializer-list **,** **}**~~

7. initializer-list → ~~initializer~~
| ~~initializer-list **,** initializer~~
| assignment-expression
| initializer-list **,** assignment-expression

8. ~~function-declarator~~ → ~~type **IDENTIFIER** (parameter-list)~~
| ~~type **IDENTIFIER** ( )~~
| ~~type **IDENTIFIER** (**VOID**)~~

9. function-definition → type **IDENTIFIER** **(** argument-list **)** **{** statement-list **}**
| type **IDENTIFIER** **(** **VOID** **)** **{** statement-list **}**
| type **IDENTIFIER** **(** **)** **{** statement-list **}**
| type **IDENTIFIER** **(** argument-list **)** **{** **}**
| type **IDENTIFIER** **(** **VOID** **)** **{** **}**
| type **IDENTIFIER** **(** **)** **{** **}**

10. ~~parameter-list~~ → ~~declarator~~
| ~~parameter-list **,** declarator~~

11. argument-list → argument
| argument-list **,** argument

12. argument → type **IDENTIFIER**
| type **IDENTIFIER** [ ]

13. compound-statement → **{** **}**
| **{** statement-list **}**

14. statement-list → statement
| statement-list statement

15. statement → **;**
| init-declarator **;**
| assignment-expression **;**

| conditional-statement
| iteration-statement
| compound-statement
| return-statement **;**

16. conditional-statement → **IF** **(** assignment-expression **)** compound-statement
| **IF** **(** assignment-expression **)** compound-statement **ELSE** compound-statement

17. iteration-statement → **WHILE** **(** assignment-expression **)** compound-statement
| **DO** compound-statement **WHILE** **(** assignment-expression **)** **;**

18. return-statement → **RETURN**
| **RETURN** assignment-expression

19. assignment-expression → logical-or-expression
| postfix-expression = logical-or-expression

20. logical-or-expression → logical-and-expression
| logical-or-expression || logical-and-expression

21. logical-and-expression → equality-expression
| logical-and-expression **&&** equality-expression

22. equality-expression → relational-expression
| equality-expression == relational-expression
| equality-expression **!=** relational-expression

23. relational-expression → additive-expression
| relational-expression < additive-expression
| relational-expression > additive-expression
| relational-expression <= additive-expression
| relational-expression >= additive-expression

24. additive-expression → multiplicative-expression
| additive-expression + multiplicative-expression
| additive-expression − multiplicative-expression

25. multiplicative-expression → ~~postfix-expression~~ unary-expression
| multiplicative-expression * ~~postfix-expression~~ unary-expression
| multiplicative-expression / ~~postfix-expression~~ unary-expression
| multiplicative-expression % ~~postfix-expression~~ unary-expression

26. unary-expression → postfix-expression
| **!** unary-expression

| − unary-expression
| − − unary-expression
| ++ unary-expression

27. postfix-expression → primary-expression
| ~~postfix-expression~~ **IDENTIFIER** [assignment-expression]
| ~~postfix-expression~~ **IDENTIFIER** **(** **)**
| ~~postfix-expression~~ **IDENTIFIER** **(** argument-call-list **)**
| ~~postfix-expression ++~~
| ~~postfix-expression − −~~

28. primary-expression → **IDENTIFIER**
| **CONSTANT-INT**
| **CONSTANT-HEX**
| **CONSTANT-FLOAT**
| **CONSTANT-CHAR**
| **STRING-LITERAL**
| **(** assignment-expression **)**

29. argument-call-list → assignment-expression
| argument-call-list **,** assignment-expression

30. type → **VOID**
| **INT**
| **FLOAT**
| **CHAR**

1.3.1 Grammar changes

Rule 3: the removed productions were replaced by the *function-definition* variable. Function declaration without definition has been removed in this version due to implementation issues, but might be added once again in a future version.

Rules 4, 5 and 6: the *declarator* and *initializer* variables have been removed, but their productions (rules 5 and 6) are now part of rule 4. This was done because of changes in the underlying data structures used to hold token data.

Rules 8 and 9: *function-declarator* (rule 8) was removed and replaced by *function-definition* (rule 9). The previous version used the variable *compound-statement* for the function body, but this resulted in difficulties to implement scope resolution.

Rules 10, 11 and 12: the variable *parameter-list* (rule 10) has been refactored into *argument-list* (rule 11) and *argument* (rule 12). This part of the grammar is now handled by a specific data structure used to hold a function's argument list.

Rules 25 and 26: unary operators were introduced, mainly to allow the negation of a value.

Rule 27: postfix increment and decrement operators were moved to rule 26 as unary prefix operators. The other productions in this rule were changed to allow only identifiers on their left-hand side, since these productions represent either array indexing or function calls, and are

now handled by specific functions and no longer require the *postfix-expression* variable to be resolved beforehand (an expression instead of an identifier here would be invalid anyway).

1.3.2 Input / output

Input and output operations are not specified in our language's grammar. However, these will be provided in the form of special polymorphic functions available to the user, akin to a small built-in `stdio.h` library. The available functions will have the following signatures:

```
void write(int i);    //
void write(char c);   // output values to the standard output
void write(char[] s); //
void write(float f);  //

void read(int i);    //
void read(char c);   // receives values from the standard input
void read(char[] s); //
void read(float f);  //
```

1.4 Semantics

1.4.1 Scope

Except in the case of initializer lists, scopes are enclosed by curly braces ('{' and '}'). Every program should have at least one scope, which is the global scope, where the programmer is only allowed to declare and initialize (but not assign) global variables and to define functions. Functions can be defined only in the global scope.

The body of a function is treated as a compound statement itself and compound statements can be nested. Within compound statements, there can be variable declaration, initialization and assignment, arithmetic and logical expressions, flow control statements (conditional and loops) and jump statements (return). Since variable assignment can only be done within compound statements and these are not allowed in the global scope, then a program must have at least one function for variables to be able to be assigned. Variables and functions must always be declared, initialized or defined before they can be used.

Declaration of variables with the same name in a given scope is not allowed. However, a variable declaration is allowed to shadow another variable in a parent scope. When a variable must be used, a declaration of that variable will be first searched in the in the current scope. If not found, it will be searched in the current scope's parent scope and so forth up to the global scope.

Unlike in the C programming language, definition of functions with the same name is allowed, provided that the functions have different argument types. Definition of functions with the same name and same argument types is not allowed, even if the return type differs. The argument variable names pertain to the outermost scope enclosed by the function's body.

1.4.2 Implicit type conversion

Our language is comprised of five basic types: `void` (no type), `char` (characters), `int` (integers), `float` (floating point numbers), and pointers in the form of arrays of one of the basic types: `char[]` (strings), `int[]` (arrays of integers), `float[]` (arrays of floating point numbers). The rules for conversions are as follows:

- **Same type** when values are of the same type, no conversion is needed;
- **`void`** cannot be implicitly converted to another type;
- **`char`** can be implicitly converted to **`int`** or **`float`**;
- **`int`** can only be implicitly converted to **`float`**;
- **`float`** cannot be implicitly converted to another type;
- **Arrays** cannot be implicitly converted to another type.

Essentially, a value can be implicitly converted to another type with larger storage size. For function calls, the same rules should apply after the function's return value has been resolved. However, types are not implicitly converted on the arguments of a function call.

1.4.3 Explicit type conversion

Since there are no casts in the language, there can be no explicit type conversion.

1.4.4 Expressions

Most of the language's operators are left-associative with the exception of unary operators and the assignment operator, which are right-associative. Expressions are type checked for incompatibilities and if possible, statically evaluated. However static evaluation of expressions has not been implemented in this version.

1.5 Use-cases

We present some use cases in which polymorphic functions would be useful. Functions with the same behavior, but different argument types:

```
int   i = -1;
float f = -1.0;
abs(i); // return the absolute value of i
abs(f); // return the absolute value of f
        // In C, abs does not accept a floating-point argument
        // we must use fabs instead
```

In this simple example, the abs function could compute the absolute of any numeric value passed as argument. The type constrains in C, however, do not allow the abs function to be called with a floating-point argument.

Functions with different behavior, but the same "intuitive meaning" for the programmer

```
int  i[3] = { 9, 2, 5 };
char w[][9] = { "word", "vocable", "locution" };

sort(i); // The user implements a sorting function for integer vectors
sort(w); // The user implements a lexicographic or alphabetic
         // sorting function for string vectors
```

The implementation of a sorting function for integers and strings would be considerably distinct. However, intuitively a programmer might expect some notion of ordering to be present in arrays of strings. This can also be applied to functions with different number of arguments:

```
// Should return the lowest of either x or y
int min(int x, int y) {
    if (x <= y) { return x; }
    else { return y; }
}

// Should also return the lowest of either x, y or z
int min(int x, int y, int z) {
    if (x <= y) {
        if (x <= z) { return x; }
        else { return z; }
    } else {
        if (y <= z) { return y; }
        else { return z; }
    }
}
```

2 The compiler

In this section we will describe the compiler's implementation for the K Programming Language, which has the responsibility to read the input text and ultimately convert it into an executable file. The first step is to define the patterns which are to be recognized by the scanner. Then, the Fast Lexical Analyzer (FLEX) [5] software is used to generate the lexical analyzer's main functionality. Moreover we provide the language's grammar to the GNU Bison parser generator [2] which generates the syntax analyzer. Additional routines for input, output, error handling and data structures to build the syntax tree and symbol table were also implemented and will be explained in detail later.

2.1 Program usage

The program takes one argument which should be a path to the input file.

```
$ ./a.out <input_file>
```

The program will then output the symbol table along with the annotated syntax tree generated from the input file, if it is part of the language. Should there be errors, the program will instead list them along with their positions in the text (line and column numbers, where tabs count as 4 columns).

2.1.1 Compilation instructions

1. (Optional) Generate the lexical analyzer .c source code through FLEX

   ```
   $ flex language.l
   ```

2. (Optional) Generate the syntax analyzer .c source code through GNU Bison

   ```
   $ bison -Wall language.y
   ```

3. Compile the program

   ```
   $ gcc -std=c18 -m64 -O3 src/main.c src/node.c src/node-list.c src/table.c src/table-stack.
       c src/arg-list.c src/parser.c src/scanner.c
   ```

4. Alternatively, a makefile has also been provided which will run all commands

   ```
   $ make
   OR
   $ make flex   # for step 1
   $ make bison  # for step 2
   $ make rel    # for step 3
   ```

Technical specifications

Operating system: Debian GNU/Linux bullseye/sid (testing release)

Compiler version: gcc (Debian 9.2.1-17) 9.2.1 20191102

FLEX version: flex 2.6.4

GNU Bison version: bison (GNU Bison) 3.4.2

2.1.2 Attached files

```
a.out       - Precompiled x64 binary
language.l  - FLEX definitions file
language.y  - BISON definitions file
makefile    -
src/        - Source code folder
tests/      - Test input files
```

2.2 The symbol table

We use a single hash table with linked lists as buckets, defined in `table.h` and `table.c`, to implement the symbol table. If performance becomes an issue due to high load factor, the table can be rehashed to decrease its load factor. Each entry in the symbol table is comprised of a key and its attributes. The following attributes are currently being stored:

- **Symbol type**: indicates if the symbol is a variable, a function or some other abstract structure such as a scope enclosed by a compound statement;
- **Return type**: stores a function's return type or a variable's data type;
- **Value**: an union that stores a symbol's current value, if applicable;
- **Argument list**: a structure that stores a function's argument list, if there is one;
- **Pointer to node**: a pointer to a node in the syntax tree that represents a statement's body
- **Scope**: a pointer to this symbol's enclosing scope, which itself is an entry in some other symbol table.

As per current implementation, the data structure used for a symbol table is the same data structure used for a symbol itself. Therefore, each entry in the symbol table is also a symbol table. The reason for this is to make the implementation of some auxiliary data structures simpler: there would be a pointer to a new symbol table in the attributes of a symbol anyway, since some entries such as functions and compound statements have their own child scope.

2.2.1 Polymorphic functions

Whereas variables are included into the table based solely on their names, the insertion of a function in the table uses a two-step hashing scheme: first, the function is hashed by its name and inserted into the table. This symbol is now a new symbol table itself. A key that represents the function's argument types is then constructed from its argument list, hashed and inserted into this new symbol table. Again, this symbol is another symbol table, which now holds the scope of the function's body, where the argument variable names are added to. For example, the following code:

```
int min(int x, int y) {}
int min(int x, int y, int z) {}
```

Would generate the following entries in the symbol table:

```
min  <s_type=FUNCTION,r_type=int,args=void>                    # function name
   int,int  <s_type=FUNCTION,r_type=int,args=int,int>         # arg types for min(int,int)
      x  <s_type=VARIABLE,r_type=int,args=void>               # arg 1 of min(int,int)
      y  <s_type=VARIABLE,r_type=int,args=void>               # arg 2 of min(int,int)
   int,int,int  <s_type=FUNCTION,r_type=int,args=int,int,int> # arg types for min(int,int,int)
      x  <s_type=VARIABLE,r_type=int,args=void>               # arg 1 of min(int,int,int)
      y  <s_type=VARIABLE,r_type=int,args=void>               # arg 2 of min(int,int,int)
      z  <s_type=VARIABLE,r_type=int,args=void>               # arg 3 of min(int,int,int)
```

Thus, a function call would look up for a function in the current scope's symbol table, then builds the argument types key from the types resolved by the type checker and looks up for such an entry in the function's symbol table. In the present version, the type checker is unable to determine the types of function calls. However, insertions and redefinition errors are being correctly handled.

2.2.2 Scope resolution

The symbol table is also responsible for scope resolution issues. Whenever a variable declaration is found, the analyzer looks up for its name in the current scope. If it is found, a symbol re-declaration error is thrown, otherwise, the variable is added to the current scope's table. Whenever a variable is used, the analyzer looks up for its name in the current scope. If it is not found, the analyzer searches the parent scope and then its parent all the way up to the global scope or until a declaration is found. If none is found, an undeclared variable error is thrown.

2.2.3 Scope management

Scopes are stored withing a stack structures which elements are symbol tables. The stack is initialized with a single element that corresponds to the program's global scope. Whenever the parser finds a rule that would begin a new scope, then a new symbol, with an appropriately given key is inserted into the current context (top of the stack) and then pushed onto the stack, becoming the new element on the top, thus the new current context. When the parser finds the terminal token that ends a scope the parser pops the context from the stack, leaving its parent on the top. These actions are handled by the parser and are defined in the GNU Bison input file in rules such as this:

```
compound_statement
   : ...
   | '{'    { begin(NULL); }    statement_list '}'    { $$ = $3  ; finish(); }
```

Here, `begin(NULL)` is a mid-rule action whose only purpose is to create a new context and push it to the stack and `finish()` pops the stack when the context is closed.

2.3 Syntax tree

The syntax tree is build using the data structures defined in `node.h` and `node.c`. Most of the non-terminals in the grammar are set as pointer to our tree node type, to allow for the tree to be built as the parser proceeds. Identifiers, types and argument lists are interpreted respectively as strings, integers and lists. They are currently being handled in such a way that they need not be included in the tree.

```
%union {
```

```
    struct node * node;
    int ival;
    const char * sval;
    struct arg_list * al;
}
```

In the current version, the scanner no longer creates tree nodes. The scanner now appropriately sets `yylval` and only the parser handles the management of the nodes. The parser is instructed to either promote or to build new nodes for most grammar productions, building the syntax tree (Listings 1 and 2).

```
declaration
    : function_definition                   { $$ = $1; }
    | init_declarator ';'                   { $$ = $1; }
```

Listing 1: Example of parser rules where nodes are promoted. This procedure is preformed for productions in which there is no need to create a new node, since no new information would be gained indoing so.

```
additive_expression
    : ...
    | additive_expression '-' multiplicative_expression
    {
        $$ = nl_push(node_list, node_init(OP_SUB, "-", $1, $3, ENDARG)); // create a new node
        assign_context($$);
        typecheck_lazy($$);
    }
```

Listing 2: Example of a parser rule where a new nodes is created. Here, we will need to type check and evaluate both operands of the expression, before we know what attributes will be assigned to the head of the rule.

2.3.1 Syntax tree annotations

As nodes are created, they may be assigned to an entry in the symbol table. Some of these entries are actual symbols derived from identifiers, while others are surrogate symbols that represent each node's current state. Currently, these symbols are only being used to store the annotations in the tree, but in the future they will used as actually temporary symbols for the production of the 3-address code. Most of the annotations are currently being done by the type checker, as will be shown in Section 2.4.

2.4 Type checker

When a node in the tree is created, sometimes it is also type-checked (See Listing 2 for example). Type-checking is done by the functions

```
Symbol * typecheck_lazy(Node * node);
```

and

```
void resolve_types(Symbol * tgt, Symbol * src);
```

The function `typecheck_lazy` is a recursive function that lazily evaluates a node's type. It is lazy because, if the node's type has already been set, then it does nothing. If, however the node's type is still not defined, then it recursively evaluates its children nodes. If the node has more than one child, `resolve_types` is called to perform implicit type conversion or otherwise

throw semantic errors. When the type checker returns, it links the node it just analyzed to an entry in the symbol table, setting its attributes accordingly.

We can use a lazy type checker here because it is being called in the same pass as the parsing, and because the tree is built bottom-up. In the current version of the compiler, however, the type checker has not yet been fully implemented as only expressions are being type checked. The type checking of some statement nodes, such as `if` and `while` nodes will require implementing additional auxiliary routines. This is the reason why the tree is not yet fully annotated.

Although the lazy type checker was implemented in such a way that it can also be used from the root of the tree, top-down, it does not currently addresses handling of nodes with different semantic meaning, which caused some annotations to be incorrectly assigned when we tried to perform the type checking on a second pass. Moreover, since the nodes do not remember the line and column in the input text file where they were created, using two passes also caused the semantic error messages related to types to display an incorrect position in the source file. Therefore, we decided for this phase of the project that it would be better that the annotations in the tree should remain incomplete, though correct.

2.5 Error handling

As the program reads the input file, the parser requests tokens from the scanner, who splits the input text into tokens and relays them back to the parser. If an error is found, a message is sent to `stderr` indicating the position (line and column) in the text where it was found, and whether it was detected by the scanner, the parser or the semantic analyzer.

2.5.1 Lexical errors

If the scanner finds an error, it outputs a message and returns a special error token to the parser. Following suggestions, identifiers with more than 32 characters are no longer considered an error and only output a warning if they are found, which does not block compilation. The scanner recognizes the following errors (Table 1).

Table 1: Lexical errors and their associated tokens. The tokens are returned to the parser and the program continues analyzing the rest of the text.

Error type	Token type
Unrecognized token	`UNRECOGNIZED_TOKEN`
Malformed character constant	`INVALID_CHAR_CONST`

2.5.2 Syntax errors

Syntax errors are thrown from the parser when it is unable to complete a shift or reduce operation. To allow the parser to continue reading the input, the built in `error` token is included in some of the grammar's rules. Particularly, we have included such rules in order to capture:

- Malformed blocks of code (the parser continues after the next semicolon or enclosing curly brace)

```
declaration
    : ...
    | error ';'
```

```
    | error compound_statement
    ;
```

- Malformed statements (the parser continues after the next semicolon)

```
statement
    : ...
    | error ';'
```

- Malformed expressions (the parser continues after the next semicolon or enclosing parenthesis)

```
primary_expression
    : ...
    | error
```

- Malformed argument lists (the parser continues after the next enclosing parenthesis)

```
function_definition
    : ...
    | type IDENTIFIER '(' error ')'
```

2.5.3 Semantic errors

Semantic errors are thrown from the semantic analyzer when there are constructs in the input text that violate the language's semantics as described in the Section 1.4. These errors do not abort the compilation process before the parser has finished, as most of the semantic analysis is performed simultaneously with the parsing. Semantic errors are handled by the functions defined in the file "`actions.c`".

Re-declaration of variable: This error is issued if a variable is declared or initialized more than once with the same name in a given scope. It is **not** an error to declare of initialize a variable with the same name of one that exists in a parent scope, but the new variable will shadow the old one, that is, the variable in the parent scope will be inaccessible.

Redefinition of function: This error is issued if a function is defined more than once with the same name **and** the same argument types. This language aims to implement polymorphic functions, therefore, defining functions with the same name and different argument types is allowed.

Variable use without declaration: This error is issued if a variable is used before being declared or initialized either in the current scope or any of its parent scopes up to the global scope. Variable declaration or initialization must always precede its use.

Function call without definition: This error is issued if a function is called but no function with that name has been defined. Functions must always be defined before they are used.

Function call with invalid argument types: This error is issued if a previously defined function is called with arguments types that are not present in any of the previous definitions of that function.

Incompatible types: This error is issued if there is an expression with operands of incompatible types, as described on Section 1.4.2.

2.5.4 Remarks about error handling

These strategies allows us to display a reasonable amount of errors before the program is unable to proceed and terminates. In fact the parser should be able to finish (albeit not correctly) the syntax tree even when errors are found in several situations. The error messages are displayed as they are found, and since the scanner returns a token that is never used by the parser, lexical errors also generate syntax errors. To avoid encumbering the output when errors are found, only error messages and the symbol table are displayed on invalid input.

2.6 Tests

Eight sample test files are provided in the folder ./tests.

2.6.1 Valid input files

valid1.c is valid, tests for most recognized keywords and punctuation;

valid2.c is valid, tests for numerical values and loops;

valid3.c is valid, tests for character and string literals;

valid4.c is valid, tests for a slightly more complex program containing several language features;

valid5.c is valid, tests mostly for multiple function declarations and scope resolution.

2.6.2 Invalid input files

error1.c is invalid and contain the following errors:
Line 1, column 25: syntax error, unexpected '*', expecting IDENTIFIER
Line 4, column 22: lexical error, unrecognized token: '?'
Line 4, column 26: lexical error, unrecognized token: ':'
Line 4, column 28: lexical error, invalid character constant.

error2.c is invalid and contain the following errors:
Line 1, column 1: lexical error, unrecognized token: '#'
Line 1, column 1: syntax error, unexpected UNRECOGNIZED_TOKEN, ...
Line 1, column 16: lexical error, unrecognized token: '.'
Line 6, column 9: warning (lexical), identifier exceeds 32 characters.
Line 6, column 52: warning (lexical), identifier exceeds 32 characters.
Line 6, column 92: semantic error: undeclared symbol 'a234567890b234567890...
Line 7, column 22: semantic error: undeclared symbol 'abcde'
Line 10, column 17: warning (lexical), identifier exceeds 32 characters.
Line 10, column 57: semantic error: redefinition of 'abcde'
Line 13, column 14: lexical error, invalid character constant.
Line 13, column 14: syntax error, unexpected INVALID_CHAR_CONST

error3.c is invalid and contain the following errors:
Line 3, column 14: semantic error: redefinition of 'x'
Line 4, column 14: semantic error: redefinition of 'y'
Line 16, column 7: semantic error: undeclared symbol 'a'

Line 22, column 5: semantic error: redefinition of 'min' with parameters 'int,int'
Line 23, column 7: semantic error: undeclared symbol 'c'
Line 25, column 20: semantic error: undeclared symbol 'max'

Lines and columns are 1-indexed and the columns should point to where the error begins. Note, however, that tabs (\t) count as if they had a width of four characters.

References

[1] V Aho Alfred et al. Compilers, Principles, Techniques & Tools. Second Edition. Pearson Education, 2007.

[2] The Free Software Foundation. GNU Bison. URL: `https://www.gnu.org/software/bison/` (visited on 10/19/2019).

[3] S.P. Harbison III and G.L. Steele. C, A Reference Manual. Fifth Edition. Prentice-Hall, 2002.

[4] C Strachey. "Fundamental concepts in programming languages." In: 13 (2000), pp. 11–49.

[5] Paxson Vern et al. The Fast Lexical Analyzer. URL: `https://github.com/westes/flex` (visited on 10/19/2019).