Building a compiler

for an yet unnamed language

Polymorphism in a C-like programming language

Diogo César Ferreira
11/0027931

University of Brasília

October 21, 2019

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 has 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.

1.1 Language 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.

1.2 Formal description

Our language's original grammar was based on that 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. However, some of the changes we had made actually rendered our grammar unsuitable for a decent programming language, resulting in problems like the inability to declare more than one function. Therefore, the following grammar has been heavily refactored and mostly written from scratch, even though some elements such as the arithmetic expressions were kept the same.

Despite the changes in the grammar's structure, most, if not all the features as we originally intended remain in the language. Particularly, only a single feature was intentionally removed: now it is no longer possible to declare and initialize more than one variable at once in a single statement (item 4). This was done in order to solve some shift/reduce conflicts that would otherwise require further restructuring on the grammar. On the other hand, we now included the possibility to declare functions without defining (item 3), as we believe will be useful in the future for better demonstrating the use of polymorphic functions.

The grammar is also provided as an input file for the GNU Bison [2] parser generator (see attached file "language.y").

1. start → declaration-list

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

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

4. init-declarator → declarator
   | declarator = initializer

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

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

9. parameter-list → declarator
   | parameter-list **,** declarator

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

11. statement-list → statement
    | statement-list statement

12. statement → **;**
    | init-declarator **;**
    | assignment-expression **;**
    | conditional-statement
    | iteration-statement
    | return-statement **;**

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

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

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

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

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

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

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

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

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

22. multiplicative-expression → postfix-expression
    | multiplicative-expression * postfix-expression
    | multiplicative-expression / postfix-expression
    | multiplicative-expression % postfix-expression

23. postfix-expression → primary-expression
    | postfix-expression [assignment-expression]
    | postfix-expression ( )
    | postfix-expression (argument-list)
    | postfix-expression ++
    | postfix-expression − −

24. primary-expression → **IDENTIFIER**
    | **CONSTANT**
    | **STRING-LITERAL**
    | (assignment-expression)

25. argument-list → assignment-expression
    | argument-list , assignment-expression

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

1.3 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

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 our yet unnamed 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 a preliminary version of the symbol table along with the syntax tree generated from the input file, if it is part of the language. Should there be lexical or syntax errors, the program will instead list them along with their positions in the text (line and column numbers, where tabs count as 4 columns).

2.2 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 -lfl -std=c11 -m64 -O3 src/main.c src/node.c src/table.c src/parser.c src/scanner.c
   ```

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

   ```
   $ make
   ```

Operating system: Debian GNU/Linux 10 (buster)

Compiler version: gcc (Debian 8.3.0-7) 8.3.0

FLEX version: flex 2.6.4

GNU Bison version: bison (GNU Bison) 3.4.1

2.3 Attached files

```
./a.out       - Precompiled x64 binary
./language.l  - FLEX definitions file
./language.y  - BISON definitions file
./makefile    -
./report.pdf  -

Source files:
./src/main.c     - Program entry point
./src/node.c     - Tree nodes data structures (source)
./src/node.h     - Tree nodes data structures (header)
./src/parser.c   - GNU Bison generated code (source)
./src/parser.h   - GNU Bison generated code (header)
./src/scanner.c  - FLEX generated code (source)
./src/scanner.h  - FLEX generated code (header)
./src/table.c    - Hash table data structures (source)
./src/table.h    - Hash table data structures (header)

Sample test files:
./tests/test_error1.c
./tests/test_error2.c
./tests/test_error3.c
./tests/test_valid1.c
./tests/test_valid2.c
./tests/test_valid3.c
./tests/test_valid4.c
./tests/test_valid5.c
```

2.4 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 or the parser.

If the scanner finds an error, it outputs a message and returns a special error token to the parser. 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
Identifier with more than 32 characters	`INVALID_IDENTIFIER`
Unrecognized token	`UNRECOGNIZED_TOKEN`
Malformed character constant	`INVALID_CHAR_CONST`

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 statements (the parser continues after the next semicolon or enclosing curly brace)

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

- 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_declarator
    : ...
    | type IDENTIFIER '(' error ')'
```

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 are displayed on invalid input.

2.5 Syntax tree

To build the syntax tree, we use the data structures defined in `node.h` and `node.c`. By setting Bison's default data type a pointer to our tree node type, we are now able to start building our syntax tree as the parser proceeds.

```
%union {
    struct node * node;
}
```

The scanner transforms some terminal tokens into leaf nodes for the tree. The structures are allocated then passed to the parser via the `yylval` variable:

```
"int"   {
            update_position();
            int t = INT;
            yylval.node = node_init(node_list, t, yytext, NULL);
            return t;
        }
```

Listing 1: Example lexer rule where a leaf node is created. This procedure is done for all data types, identifiers and constants.

Then, the parser is instructed to connect these nodes forming the syntax tree. Nodes are connected for most (except when the `error` token is present) grammar productions. In some cases, such as when an identifier is found, a symbol is also added to the symbol table.

```
declarator : type IDENTIFIER
    {
        $$ = node_init(node_list, 'D', "declarator-variable", $1, $2, NULL);
        add_symbol_var($$);
    }
```

Listing 2: Example parser rule where nodes are connected. This procedure is done for all productions, except error handlers. Here, a symbol is also added to the symbol table through the function `add_symbol_var`

2.6 Symbol table and polymorphic functions

We use a hash table, defined in `table.h` and `table.c`, to implement the symbol table. At this point, the symbol table only stores variable and function names (and their types). We, however, are already able to distinguish each function by their signature, rather than only their name, by analyzing the syntax tree.

Whereas variables are included to the table based solely on their names, a function uses a combination of both its name the list if its argument's types to form the key for the hash table. To do that, we append additional characters to the symbol key as we traverse the function's argument list from the syntax tree. For example, by parsing the following code (see attached file "tests/test_valid5.c"):

```
int min(int x, int y);
int min(int x, int y, int z);
float min(float x, float y);
float min(float x, float y, float z);
```

The following entries would be generated in the symbol table:

Symbol	Return type
min_ii	int
min_iii	int
min_ff	float
min_fff	float

Each entry still knows their original name, but to be accessed in the table, the new key must be calculated, thus resolving collisions between functions with the same name but different signatures.

2.7 Tests

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

test_valid1.c is valid, tests for most recognized keywords and punctuation;

test_valid2.c is valid, tests for numerical values and loops;

test_valid3.c is valid, tests for character and string literals;

test_valid4.c is valid, tests for a slightly more complex program containing several language features;

test_valid5.c is valid, tests mostly for multiple function declarations;

test_error1.c is invalid, and contain the following errors:

- Line 1, column 25: syntax error, unexpected '*', expecting IDENTIFIER
- Line 2, column 22: lexical error, unrecognized token ('?')
- Line 2, column 22: syntax error, unexpected UNRECOGNIZED_TOKEN
- Line 2, column 26: lexical error, unrecognized token (':')
- Line 2, column 28: lexical error, invalid character constant. (char constant too long)

test_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, expecting VOID or INT or FLOAT or CHAR
- Line 1, column 16: lexical error, unrecognized token ('.')
- Line 4, column 9: lexical error, identifier exceeds 32 characters.
- Line 4, column 52: lexical error, identifier exceeds 32 characters.
- Line 6, column 14: lexical error, invalid character constant.
- Line 6, column 14: syntax error, unexpected INVALID_CHAR_CONST

test_error3.c is invalid, and contain the following errors:

- Line 2, column 12: syntax error, unexpected IDENTIFIER
- Line 3, column 15: syntax error, unexpected ')', expecting IDENTIFIER
- Line 4, column 19: syntax error, unexpected ';', expecting '[' or ',' or ')'
- Line 7, column 1: syntax error, unexpected VOID, expecting ';' or '{'
- Line 12, column 12: syntax error, unexpected ')', expecting IDENTIFIER or STRING_LITERAL or CONSTANT or '('
- Line 41, column 27: syntax error, unexpected ';'

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 length of four characters.

2.8 Additional remarks

Overall, this phase of the project presented itself more challenging than the previous ones, given the more complex nature of the parser, as opposed to the scanner. In particular, a remarkably nasty memory leak accompanied by several segfaults showed up as the syntax tree was being built while parsing input with errors. The solution for that was to reserve ownership of the allocated tree nodes to a vector that stores every node as they are allocated (`Nodelist * node_list` in `main.c`).

References

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

[2] The Free Software Foundation. *GNU Bison*. URL: `https://www.gnu.org/software/bison/` (visited on 04/19/2019).

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

[4] C Strachey. "Fundamental concepts in programming languages." In: *Higher-Order and Symbolic Computation* 13 (2000), pp. 11–49.

[5] Paxson Vern et al. *The Fast Lexical Analyzer*. URL: `https://github.com/westes/flex` (visited on 04/19/2019).