Compilers - Part 1: Outline

Diogo César Ferreira
11/0027931

University of Brasília

August 26, 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 programing 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, 2].

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 programing language. The grammar of the language is presented on the following section.

2 Language grammar

2.1 Formal description

1. program → function_definition
   | declaration
2. function_definition → type_specifier IDENTIFIER **(** parameter_list **)** compound_statement
   | type_specifier IDENTIFIER **( )** compound_statement
3. parameter_list → parameter_declaration
   | parameter_list **,** parameter_declaration
4. parameter_declaration → type_specifier declarator
5. compound_statement → **{ }**
   | **{** statement_list **}**
   | **{** declaration_list **}**
   | **{** declaration_list statement_list **}**
6. declaration_list → declaration
   | declaration_list declaration

7. declaration → type_specifier init_declarator_list **;**

8. init_declarator_list → init_declarator
   | init_declarator_list **,** init_declarator

9. init_declarator → declarator
   | declarator **=** initializer

10. declarator → IDENTIFIER
    | **(** declarator **)**
    | declarator **[** logical_or_expression **]**
    | declarator **[ ]**
    | declarator **(** identifier_list **)**
    | declarator **(** parameter_list **)**
    | declarator **( )**

11. statement_list → statement
    | statement_list statement

12. statement → compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement

13. selection_statement → IF **(** expression **)** statement
    | IF **(** expression **)** statement ELSE statement

14. iteration_statement → WHILE **(** expression **)** statement
    | DO statement WHILE **(** expression **) ;**

15. jump_statement → RETURN **;**
    | RETURN expression **;**

16. expression_statement → **;**
    | expression **;**

17. identifier_list → IDENTIFIER
    | identifier_list **,** IDENTIFIER

18. initializer → assignment_expression
    | **{** initializer_list **}**
    | **{** initializer_list **, }**

19. initializer_list → initializer
    | initializer_list **,** initializer

20. expression → assignment_expression
    | expression **,** assignment_expression

21. argument_expression_list → assignment_expression
    | argument_expression_list **,** assignment_expression

22. assignment_expression → logical_or_expression
    | postfix_expression **=** assignment_expression

23. logical_or_expression → logical_and_expression
    | logical_or_expression OR_OP logical_and_expression

24. logical_and_expression → equality_expression
    | logical_and_expression AND_OP equality_expression

25. equality_expression → relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression

26. relational_expression → additive_expression
    | relational_expression < additive_expression
    | relational_expression > additive_expression
    | relational_expression LE_OP additive_expression
    | relational_expression GE_OP additive_expression

27. additive_expression → multiplicative_expression
    | additive_expression + multiplicative_expression
    | additive_expression - multiplicative_expression

28. multiplicative_expression → postfix_expression
    | multiplicative_expression * postfix_expression
    | multiplicative_expression / postfix_expression
    | multiplicative_expression % postfix_expression

29. postfix_expression → primary_expression
    | postfix_expression [expression]
    | postfix_expression ( )
    | postfix_expression (argument_expression_list)
    | postfix_expression INC_OP
    | postfix_expression DEC_OP

30. primary_expression → IDENTIFIER
    | CONSTANT
    | STRING_LITERAL
    | (expression)

31. type_specifier → VOID
    | CHAR
    | INT
    | FLOAT

2.2 Remarks

The language specified in this work should contain:

Keywords *if else do while return*

Data types *void char int float*

Character and string delimiters *' "*

Other symbols *( ) [ ] { } ; ,*

Arithmetical operators *+ - * / % ++ --*

Logical operators *&& ||*

Comparison operators *< > <= >= ==*

Comments */* */ //*

No special syntactic structure has been designed for polymorphic function declaration. This feature will be implemented either through changes in how the symbol table is built or through parse tree annotations at later stages in the compilation process.

3 Semântica

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; }
        }
}
```

References

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

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