# Compilers Preparation 2

1511186
Liang Chen

February 25, 2020

**Abstract**

Abstract

**Keywords**

## 1 C Syntax Description

### 1.1 Outline

Generally speaking, A C program is made up of many **declarations** and **function definition**, you can only implement all the function you want inside a definition of a function, where you write **statements**. Most statements inform the compiler how you want to control this program, such as iteration, selection or directly jump. And there's another important statement called expression statement which is mainly consisted of an **expression**. An expression is a minimal function element of a program, it can call a function, modify memory units, or just simply compute a value.

There is an exact standard definition of C syntax, referencing that, here I only discuss the parts that **may** be concerned in the later assignment.

### 1.2 Program

A C program is in fact a series of declarations and function definitions.

$$\begin{aligned} Program \rightarrow\ & Declaration \quad Program \\ |\ & Function\_Definition \quad Program \\ |\ & \boldsymbol{\epsilon} \end{aligned}$$

## 1.3 Declaration

Declarations are used to introduce the identifiers used in a program and to specify their important attributes, such as type, storage class, and identifier name. A declaration that also causes storage to be reserved for an object or that includes the body of a function, is called a definition. Formally, the syntax of the declaration is:

$$Declaration \rightarrow Declaration\_Specifier \quad Init\_Declarator\_List$$

### 1.3.1 Declaration Specifier

Declaration specifier specific this declaration, such as its data type and how to store and some more details. Formally:

$$\begin{aligned} Declaration_Specifier \rightarrow\ & Storage\_Class\_Specifier \quad Declaration\_Specifier \\ |\ & Type\_Specifier \quad Declaration\_Specifier \\ |\ & Storage\_Class\_Specifier \\ |\ & Type\_Specifier \end{aligned}$$

$$\begin{aligned} Storage\_Class\_Specifier \rightarrow\ & \textbf{\textit{auto}} \\ |\ & \textbf{\textit{static}} \\ |\ & \textbf{\textit{extern}} \\ |\ & \boldsymbol{\epsilon} \end{aligned}$$

$$\begin{aligned} Type_Specifier \rightarrow\ & \textbf{\textit{Data\_Type}} \\ |\ & \boldsymbol{\epsilon} \end{aligned}$$

Data_type is basically all the types the compiler accepts:

$$\begin{aligned} Data\_Type \rightarrow\ & \textbf{\textit{bool}} \\ |\ & \textbf{\textit{float}} \\ |\ & \textbf{\textit{void}} \\ |\ & \textbf{\textit{int}} \end{aligned}$$

Notice that each type of the declaration specifier is optional and can only occur once, but except for function declaration where at least one such specifier

or qualifier must be present. And I can see now how syntax itself is really not enough for entire compiling, you have to define many semantic actions to control the intermediate data structure for all the semantic check. Usually you can put the hard work down to the semantic action instead of making up a really enormous syntax rule.

### 1.3.2 Initial Declarator

Initial declarator list indicates some identifiers to be declared. Formally:

$$\begin{aligned} Init\_Declarator\_List &\rightarrow Init\_Declarator\_List\ \mathbf{,}\ Init_{D}ecarator \\ &\mid \epsilon \end{aligned}$$

$$\begin{aligned} Init\_Declarator &\rightarrow Declarator \\ &\mid Declarator\ =\ Initializer \end{aligned}$$

The declarator indicates the object or function being declared. It can be as simple as a single identifier, or can be a complex construction declaring an array, structure, pointer, union, or function. Mainly:

$$\begin{aligned} Declarator &\rightarrow Pointer\ Direct\_Declarator \\ &\mid Direct_{D}eclarator \end{aligned}$$

Non-terminal Pointer describes the prefix about pointer, while the Direct_Declarator is a object, formally:

$$\begin{aligned} Pointer &\rightarrow *\ Pointer \\ &\mid * \end{aligned}$$

$$\begin{aligned} Direct\_Declarator &\rightarrow Direct\_Declarator\ [\ Const\_Expression\ ] \\ &\mid Direct\_Declarator\ (\ ) \\ &\mid Direct\_Declarator\ (\ Parameter\_List\ ) \\ &\mid (\ Declarator\ ) \\ &\mid \boldsymbol{Identifier} \end{aligned}$$

This kind of syntax mainly means the declaration of function, array, pointer and their combinations. The Parameter_List describes the parameterss of a function here. Formally:

$$\begin{aligned} \mathit{Parameter\_List} &\rightarrow \mathit{Parameter\_List}\ \mathbf{,}\ \mathit{Parameter\_Declaration} \\ &\mid \mathit{Parameter\_Declaration} \end{aligned}$$

The Parameter_Declaration here is similar to the one above, except that names of params here can be omitted. Formally:

$$\begin{aligned} \mathit{Parameter\_Declaration} &\rightarrow \mathit{Declaration\_Specifiers} \\ &\mid \mathit{Declaration\_Specifiers}\quad \mathit{Declarator} \\ &\mid \mathit{Declaration\_Specifiers}\quad \mathit{Abstract\_Declarator} \end{aligned}$$

The initializer is basically a set of values to assign to the declarator. Formally:

$$\begin{aligned} \mathit{Initializer} &\rightarrow \mathit{Assignmen\_Expression} \\ &\mid \mathbf{\{}\ \mathit{Initializer\_List}\ \mathbf{\}} \end{aligned}$$

$$\begin{aligned} \mathit{Initializer\_List} &\rightarrow \mathit{Initializer\_List}\ \mathbf{,}\ \mathit{Initializer} \\ &\mid \mathit{Initializer} \end{aligned}$$

This syntax accepts initial assignments of basic data types and arrays, but not the structs.

## 1.4 Function Definition

A function definition includes the code of the function. Function definitions can appear in any order, and in one source file or several, although a function cannot be split between files. Function definitions cannot be nested. Formally:

$$\mathit{Function\_Definition} \rightarrow \mathit{Declaration\_Specifier}\ \mathit{Declarator}\ \mathit{Compound\_Statement}$$

We have already defined most of non-terminals above, but not yet the compound statement, which is a type of statement. The statement is another huge topic to discuss.

## 1.5 Statement

And focusing on the syntax of statements, we can see that they are classified into several types:

1. Labeled Statement
2. Compound Statement
3. Expression Statement
4. Selection Statement
5. Iteration Statement
6. Jump Statement

Therefore, we can now give a syntax about statement:

$$\begin{aligned} \textit{Statement} \rightarrow\ & \textit{Labeled\_Statement} \\ |\ & \textit{Compound\_Statement} \\ |\ & \textit{Expression\_Statement} \\ |\ & \textit{Selection\_Statement} \\ |\ & \textit{Iteration\_Statement} \\ |\ & \textit{Jump\_Statement} \end{aligned}$$

Now we dig on each type of statement.

### 1.5.1 Labeled Statement

Labeled statements are preceded by a colon, it is used to flag a location in a program which is usually a target of 'goto'. It has three types: The first one is just an identifier followed by a colon and another statement, the other two, 'case' and 'default', is used in switch statement. Formally:

$$\begin{aligned} \textit{Labeled\_Statement} \rightarrow\ & \boldsymbol{Identifier}\ \textbf{:}\ \textit{Statement} \\ |\ & \boldsymbol{case}\quad \textit{Constant\_Expression}\ \textbf{:}\ \textit{Statement} \\ |\ & \boldsymbol{default}\ \textbf{:}\ \textit{Statement} \end{aligned}$$

### 1.5.2 Compound Statement

A compound statement, also called a block, consisted of a series of optional declarations and optional statements within braces. Formally:

$$\textit{Compound\_Statement} \rightarrow \{\ \textit{Block\_Content}\ \}$$

$$\begin{aligned} \textit{Block\_Content} \rightarrow\ & \textit{Declaration}\quad \textit{Block\_Content} \\ |\ & \textit{Statement}\quad \textit{Block\_Content} \\ |\ & \textit{Declaration}\quad \textit{Block\_Content} \\ |\ & \boldsymbol{\epsilon} \end{aligned}$$

### 1.5.3 Expression Statement

A valid expression followed by a semicolon can be a statement.Formally:

$$Expression\_Statement \rightarrow Expression\ \mathbf{;}$$

Notice that expression here is another big thing to discuss, but let's just save it for later.

### 1.5.4 Null Statement

A null statement is used to provide a null operation in situations where the grammar of the language requires a statement, but the program requires no work to be done. The null statement consists of a semicolon. Formally:

$$Null\_Statement \rightarrow \mathbf{;}$$

In fact I think we can merge null statement and expression statement by assigning a $\boldsymbol{\epsilon}$ expression.

### 1.5.5 Selection Statement

I think it is a well-known statement. Two types: if-statement and switch-statement. Remember that we have already defined case statement and default statement for switch before. Formally:

$$\begin{aligned} Selection\_Statement &\rightarrow If\_Statement \\ &\mid Switch\_Statement \end{aligned}$$

$$\begin{aligned} If\_Statement &\rightarrow \boldsymbol{if}\ (\ Expression\ )\ Statement \\ &\mid \boldsymbol{if}\ (\ Expression\ )\ Statement \quad \boldsymbol{else} \quad Statement \end{aligned}$$

$$Switch\_Statement \rightarrow \boldsymbol{switch}\ (\ Expression)\ Statement$$

Notice here that instead of making a complex syntax to accept case statement only inside a switch statement, this syntax is fairly loose, this means you have to check the semantic of each case statement to make sure it's valid.

### 1.5.6 Iteration Statement

This is also pretty well-known, it has 3 types: while-statement, do-statement and for-statement, formally:

$$
\begin{aligned}
\mathit{Iteration\_Statement} &\rightarrow \mathit{While\_Statement} \\
&\mid \mathit{Do\_Statement} \\
&\mid \mathit{For\_Statement}
\end{aligned}
$$

$$
\mathit{While\_Statement} \rightarrow \textbf{\textit{while}}\ (\ \mathit{Expression}\ )\ \mathit{Statement}
$$

$$
\mathit{Do\_Statement} \rightarrow \textbf{\textit{do}}\ \mathit{Statement}\ \textbf{\textit{while}}\ (\ \mathit{Expression}\ )\ ;
$$

$$
\begin{aligned}
\mathit{For\_Statement} &\rightarrow \textbf{\textit{for}}\ (\ \mathit{Expression\_Statement}\ ;\ \mathit{Expression\_Statement}\ ;\ \mathit{Expression}\ )\ \mathit{Statement} \\
&\mid \textbf{\textit{for}}\ (\ \mathit{Expression\_Statement}\ \mathit{Expression\_Statement}\ )\ \mathit{Statement} \\
&\mid \textbf{\textit{for}}\ (\ \mathit{Declaration}\ \mathit{Expression\_Statement}\ \mathit{Expression}\ )\ \mathit{Statement} \\
&\mid \textbf{\textit{for}}\ (\ \mathit{Declaration}\ \mathit{Expression\_Statement}\ )\ \mathit{Statement}
\end{aligned}
$$

This syntax of for statement accepts a declaration inside the iteration and cases without terminal condition.

### 1.5.7 Jump Statement

Actually this is familiar, too, namely all instructions that transfer control. It has four types: goto-statement, continue-statement, break-statement and return-statement. Formally:

$$
\begin{aligned}
\mathit{Jump\_Statement} &\rightarrow \mathit{Goto\_Statement} \\
&\mid \mathit{Continue\_Statement} \\
&\mid \mathit{Break\_Statement} \\
&\mid \mathit{Return\_Statement}
\end{aligned}
$$

$$
\mathit{Goto\_Statement} \rightarrow \textbf{\textit{goto}}\ \textbf{\textit{Identifier}}\ ;
$$

$$
\mathit{Continue\_Statement} \rightarrow \textbf{\textit{continue}}\ ;
$$

$$
\mathit{Break\_Statement} \rightarrow \textbf{\textit{break}}\ ;
$$

$$
\begin{aligned}
\mathit{Return\_Statement} &\rightarrow \textbf{\textit{return}}\ \mathit{Expression}\ ; \\
&\mid \textbf{\textit{return}}\ ;
\end{aligned}
$$

## 1.6 Expression

So far we have seen many derivations stops at forms of expression. Expressions are sequences of operators and operands that are used for one or more of these purposes:

1. Computing a value from the operands.
2. Designating objects or functions.
3. Generating side effects, which are any actions other than the evaluation of the expression for example, modifying the value of an object.

The ANSI C standard defines the priority and combination rules of operators only by CFG syntax, where the atomic unit of an expression is called a primary expression, it can be either an identifier, a constant, a string literal or an expression within a pair of parenthesis, formally:

$$\begin{aligned} \mathit{Primary\_Expression} \rightarrow\ & \boldsymbol{Identifier} \\ \mid\ & \boldsymbol{Constant} \\ \mid\ & \boldsymbol{String\_Literal} \\ \mid\ & (\ \mathit{Expression}\ ) \end{aligned}$$

Let's not discuss about the 'Expression' here now. Up on primary expressions, postfixes can be added. The postfix expression is primary expressions with postfixes, which can indicate an access of a pointer, an array or a struct member; A function call; A self-increment or decrement. Formally:

$$\begin{aligned} \mathit{Postfix\_Expression} \rightarrow\ & \mathit{Postfix\_Expression} -> \boldsymbol{Identifier} \\ \mid\ & \mathit{Postfix\_Expression}\ [\ \mathit{Expression}\ ] \\ \mid\ & \mathit{Postfix\_Expression}\ \boldsymbol{.Identifier} \\ \mid\ & \mathit{Postfix\_Expression}\ (\ ) \\ \mid\ & \mathit{Postfix\_Expression}\ (\ \mathit{Argument\_Expression\_List}\ ) \\ \mid\ & \mathit{Postfix\_Expression}\ ++ \\ \mid\ & \mathit{Postfix\_Expression}\ -- \\ \mid\ & \mathit{Primary\_Expression} \end{aligned}$$

The argument expression list here describes a set of parameters to be passed to a function, formally:

$$\begin{aligned} \mathit{Argument\_Expression\_List} \rightarrow\ & \mathit{Argument\_Expression\_List}\ ,\ \mathit{Assignment\_Expression} \\ \mid\ & \mathit{Assignment\_Expression} \end{aligned}$$

The syntax of argument expression will be concerned later. Upon postfix expressions, the most prior operator is the unary operator, formally:

$$\begin{aligned} Unary\_Expression &\rightarrow Unary\_Operator \quad Unary\_Expression \\ &\mid Postfix\_Expression \end{aligned}$$

$$\begin{aligned} Unary\_Operator &\rightarrow ++ \\ &\mid -- \\ &\mid \& \\ &\mid * \\ &\mid + \\ &\mid - \\ &\mid \sim \\ &\mid ! \end{aligned}$$

And the next prior operators are the multiplicative operators, formally:

$$\begin{aligned} Multiplicative\_Expression &\rightarrow Multiplicative\_Expression \quad Multiplicative\_Operator \quad Unary\_Expression \\ &\mid Unary\_Expression \end{aligned}$$

$$\begin{aligned} Multiplicative\_Operator &\rightarrow * \\ &\mid / \\ &\mid \% \end{aligned}$$

We can give the syntax of the complex expression with many operators in a priority list from high to low:

1. Postfix operators
2. Unary operators
3. Multiplicative operators
4. Additive operators(+, -)
5. Relational operators(>, <, >=, <=)
6. Equality operators(==, !=)
7. And operator(&)
8. Exclusive or operator(^)
9. Inclusive or operator(|)

10. Logical and operator(&&)
11. Logical or operator(||)
12. Assignment operators(=, +=, -=,...)

Similar to the syntax above, we can define the syntax of the topmost assignment expression, which is the superset of all the lower expressions. And now we can give the actual syntax of the general expression. Expressions are a series of assignment expression divided by ',', formally:

$$\begin{aligned} \mathit{Expression} \rightarrow\ & \mathit{Expression}\ \mathbf{,}\ \mathit{Assignment\_Expression} \\ |\ & \mathit{Assignment\_Expression} \end{aligned}$$

## 1.7 Summing-up

So far we have given the outline of the C syntax, and from doing this I realize that there is a strong relationship between the syntax and semantics. The syntax precedes the semantic check, it firstly rules that what kind of input is syntax-valid. And each piece of syntax corresponds a semantic action, check the semantic level of the syntax-valid input. The input is accepted by the compiler when it is valid at both syntax and semantic level.

Therefore, there will be a trade-off between the syntax and the semantic level: You can either make a complex syntax, so the semantic check will be much easier, almost every input that is valid in syntax will also be valid at semantic level; Or else, you can make the syntax very simple, but on the other hand, the semantic check will be more complicated. The goal of the design is to compromise between these two.

Here is an example: case statement. The C language requires that case statements can only occur inside a switch statement. One way to accomplish is to define the behavior into the syntax, and that will split the case statement from the general statement, which seems a little weird; Or like the ANSI C standard syntax putting this into semantic actions, the syntax will be much more clear and well-organized. But one thing important to keep in mind: Consider what it will take at the semantic level when you are making the syntax definition.

# 2 Assembly Example

## 2.1 Environment

I've wrote two ASM programs for GNU's assembler[1], which is in AT&T syntax, using function 'scanf' and 'printf' of C standard library in Linux[2]. So they can be directly accepted by **GCC** in **Linux**, but **not available** in Windows, because of the difference of lib function name and ABI[3] calling conventions, which will be mentioned later.

## 2.2 Introduction

The first example is a very simple program calculating the factorial of an integer, with its source code named 'Factorial.c' and ASM translation named 'Factorial.s'; The second example is a recursion program calculating Fibonacci sequence, with its source code named 'Fibonacci.c' and ASM translation named 'Fibonacci.s'.

The annotations of the codes include more details, but this report will mainly discuss one topic that the compiler must involve: Calling convention.

## 2.3 Calling Conventions

Calling conventions stipulate the behavior of the stack during the function call, such as how to prepare the stack for the callee[4]; How the parameters and results are passed; How the registers should be carefully used.

## 2.4 Stack Preparation

The callee should firstly save the caller's stack frame in manner, then change the base pointer to the new stack base:

```
73      pushq   %rbp
74      movq    %rsp, %rbp
```

After this, the callee allocates the stack for itself, by subtract the stack pointer with an offset:

```
79      subq    $16, %rsp
```

[1] Abbreviated to GAS
[2] Remember last report introduces how the linker find and invoke the lib function automatically.
[3] Application Binary Interface
[4] The function being called

There are two reasons for this instruction: First, it will reserve a stack space for the upcoming local variables which will be base addressing with BP; Second, if there will be another function called by this callee, this space may also be used to pass parameters which will be base addressing with SP.

Meanwhile, it is a standard of X64 SYSTEM V ABI that, the SP must be divided by 16 or 32 bytes when execute a function call. This may need some explanation: When a 'CALL' instruction is executed, the SP is 16 bytes aligned, then the 'CALL' instruction pushes the RIP before the callee starts and pushes RBP into the stack. The RSP still 16 bytes aligned after these two 'PUSHQ'. Therefore, the offset to subtract should be 16 bytes aligned to keep this manner.

## 2.5 Passing Parameters

Back to those days of X86, we only have few registers to use, so the stack is used to pass parameters. This results an argument that who is in charge to sweep the parameters out of stack. Conventions like CDECL stipulate that the caller is in charge, and this provides some features like variable parameters[5]; Some other conventions like FASTCALL stipulate oppositely, and this may be faste even though it can't support variable parameters.

Because of the increment of registers, modern CPU and OS mainly follow the X86-64 conventions, which are divided into Microsoft X64 conventions and System V AMD64 conventions[6]. Windows is using the first one while Linux is using the second, and they're not compatible. But the common decision is to use registers to pass several parameters[7], and the others will still be passed by stack. They individually stipulate different registers to pass corresponding parameters. This example is in Linux so the DI and SI are used.

```
104     leaq    -4(%rbp), %rsi
105     movl    $string1, %edi
106     call    __isoc99_scanf
```

## 2.6 Register Protection

Another problem is that during the function call, contents in registers may be volatile and lost, so it is rather necessary to save the registers that will still be used after the call. In X86-64 conventions, Registers like RBXRBPR12R13R14R15 are protected by the caller, that means the caller should push them into stack before the call if they are in use, however, the callee can use them without any manner; Registers such as R10, R11 are protected by the callee, that means the it is reliable to the caller while the callee should care for them before using.

[5] 'printf' for example
[6] Even though this architecture is designed by AMD. MS just loves being different.
[7] 4 in MS and 6 in AMD

```
50      pushq   %r11
51      subq    $8, %rsp
52      call    fb
53      addq    $8, %rsp
54      popq    %r11
```

# 3 Compilers Implement

Here are some guesses and thoughts about the implement of compilers. At the syntax level, since the statements are arranged linearly and basically individual, the statements can be translated individually by actions on the AST; At the semantic level, there should be a list or dictionary if the efficiency is concerned, recording all the symbols and their attributes such as variables and their types, labels, and functions and their signatures. Once the compiler recognizes a reference of a symbol, it checks whether this syntax action is valid in semantic.

# 4 References

References are in the attachment.