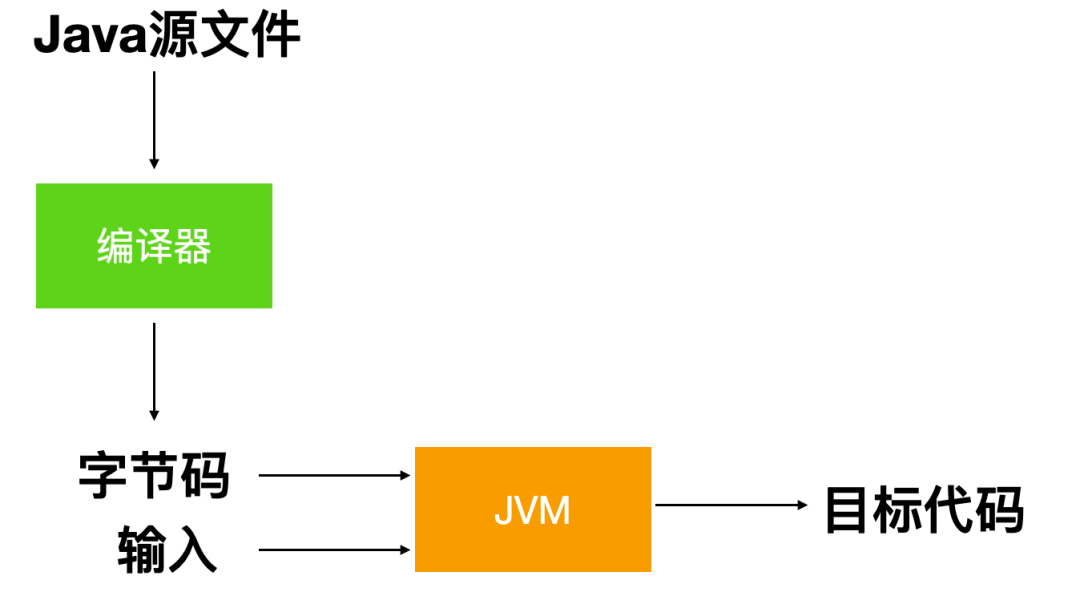


# 编译器的基本概念

除了编译器之外，还有一种称为 解释器(interpreter) 的语言处理器，它不是做翻译工作的，而是把用户提供的输入执行源程序中指定的操作。



我们熟知的 Java 语言，就结合了编译和解释的过程，我们写的 Java 源文件首先被编译成 字节码(bytecode)，字节码是一种中间码，它通常被看成是可执行的二进制文件。然后再由 Java 虚拟机对字节码解释执行。这样，在一台机器上编译的字节码就能够在其他机器上解释执行，这种体现了 Java 语言的平台无关性。



为了提高编译速度，Java 中有一种 just-in-time,JIT 即时编译器会一边编译一边执行。

一个源文件程序可能被划分为多个模块，并存放在多个文件中，还需要把文件链接在一起，所以，除了编译器之外，还需要一种能链接文件的部件参与，预处理器(preprossor) 是做这件事情的。如下图所示



预处理器经过预处理后会作为输入传递给编译器，编译器对源程序进行编译，编译完成后生成汇编代码，作为汇编器的输入传递给汇编器，汇编器进行汇编处理转换为机器代码，注意这个时候还不是目标代码，还要经过链接器与系统库函数进行链接，最后由加载器把目标代码加载到内存中执行。

# 编译器的结构

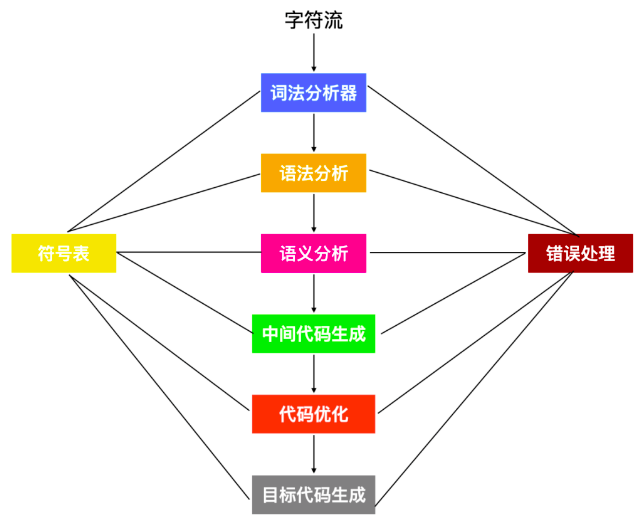
我们上面大概了解了一下语言的处理过程，下面我们就来了解一下编译器的内部结构，编译器内部其实具有两种结构：分析(analysis)部分和 整合(synthesis) 部分。

分析过程相当于是把源程序分成多个结构，每个结构都有特定的语法格式进行校验，在经由每个校验后，如果不满足指定的语法格式则进行提醒，使用户进行修改。分析部分还会收集有关源程序的信息，会把收集到的信息存放在一个被称为 符号表(symbol table) 的数据结构中。符号表和中间表示形式一起传给整合部分。

整合过程是根据分析过程传递的信息来构造用户期待的目标程序。分析和整合统称为 前端(front end) 和 后端(back end) ，哈哈哈哈。

这里你需要知道符号表(Symbol Table) 的概念：符号表是编译器使用和维护的数据结构，由标识符和类型组成。符号表的主要作用是帮助编译器快速定位。

下面是一个编译器的典型结构



# 词法分析

词法分析(Lexical Analyzer)是编译器的第一个步骤，它也被称为 扫描(scanning)。词法分析器通过读入外部的字符流对其进行扫描，并且把它们组成有意义的词素(lexeme)序列，对于每个词素，词法分析器都会产生词法单元(token) 作为输出。这个词法单元会传递给下一个步骤，也就是语法分析。

这里解释一下token、词素和词法分析器的概念

我们常用的编程语言就是具有词素的单词和符号的集合，比如 C 语言中有 ()，-> 等等。关键字 if...while...，变量或函数名称以及数字和字符串常量也被视为词素。并不是所有的字符都属于词素，例如空格和注释就不属于。

词法分析器用来分析词素有两个规则

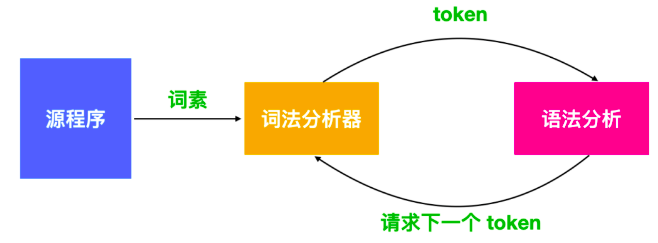
* 跳过不能以字母开头的字符
* 然后找到剩余的最长前缀，也就是词素

这两句话比较抽象，举个例子来说明一下

ifx = 20\*30;

那么第一个词素就是 ifx，为什么不是 if 呢？因为 if 不是最长的前缀。然后后面的词素依次是 =，20，\*，30和；。

词素、词法分析器、token 的关系如下：



词素是 Token 的实例，词法分析器的主要任务就是从源程序中读取字符并产生 token。token 也是有结构的，一般结构如下



在词法分析生成的 token 中，第一个词 token-name 是语法分析期间使用的抽象符号，第二个词 attribute-value 指向的是符号表中关于这个词法单元的条目数。

我们举个例子来看一下词法分析的拆解过程。

比如现在源程序中有一个赋值语句：

income = mainjob + sideline // 收入 = 主业 + 副业

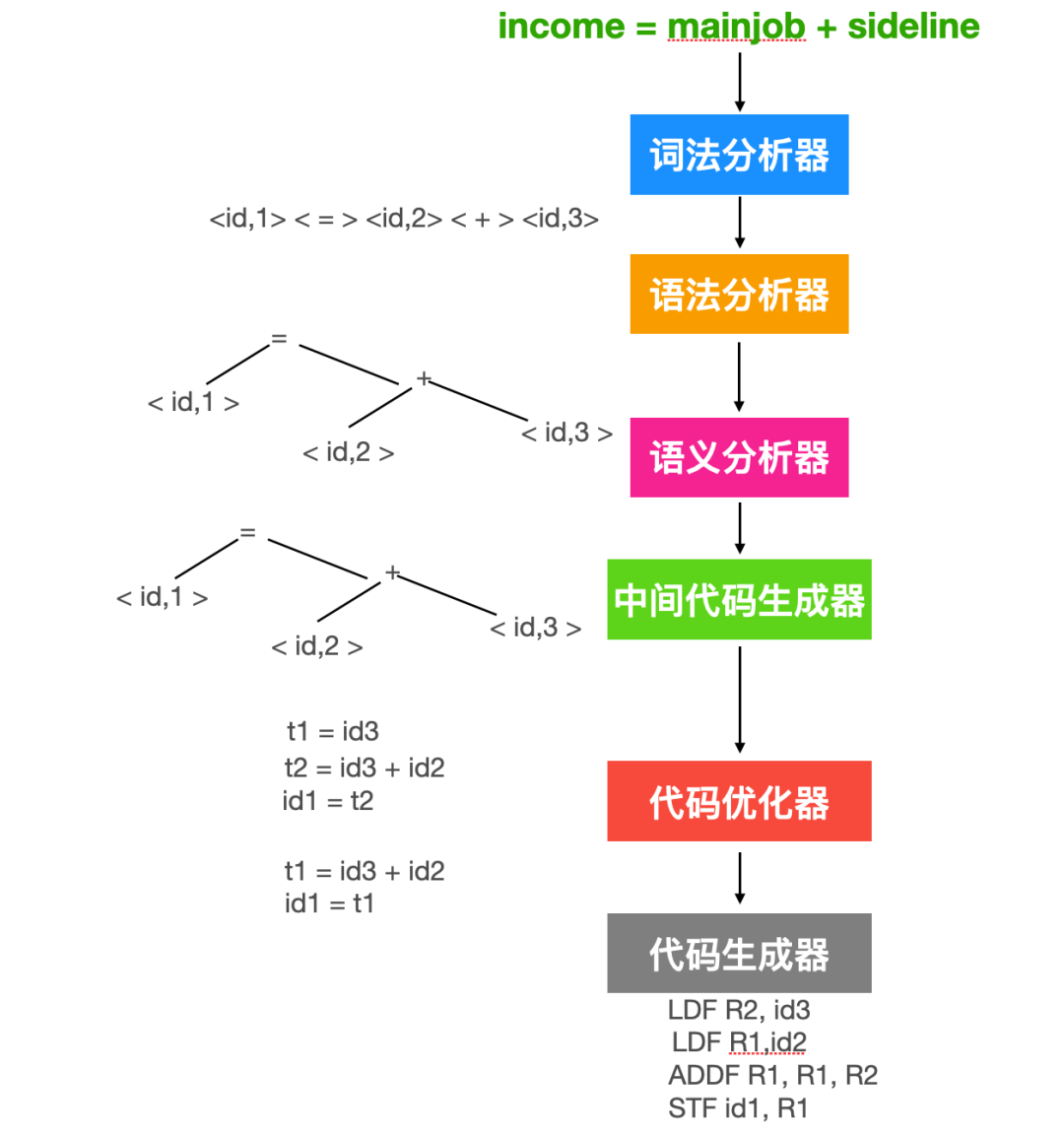
这个赋值语句中的字符可以组合成如下词素，并转换成为 token，并传递给语法分析阶段。

* 首先，income 是一个词素，它会被映射为 <id,1>，其中  id 是表示的 标识符(identifier) 的抽象符号，而 1 指的是符号表中 income 在符号表中的条数。
* 然后是赋值符号 = ，它也是一个词素，被映射称为 token 中的 < = >。这个 token 不需要属性值，所以没有第二个词。
* mainjob 是一个词素，它被映射成为 token 中的 <id,2>，2 是 mainjob 对应的符号表条目
* +也是一个词素，它被映射称为 < + >，没有条目数
* sideline 是一个词素，它被映射称为 token 中的 <id,3>，3 是 sideline 对应的符号表条目

所以，经过词法分析后，上面的源程序会变为

<id,1> < = > <id,2> < + > <id,3>

在上面的表达式中， = 和 + 分别表示赋值和加法运算符的抽象符号。用图来表示的话就是



# 语法分析

编译器的第二个步骤是 语法分析(syntax analysis) 或者称为 解析(parsing)。语法分析器使用由词法分析器生成的各个词法单元的第一个分量来创建树形的中间表示。常用的方法就是 语法树(syntax tree)。编译器的后续步骤都会使用这个语法结构来帮助分析源程序，并生成目标程序。

# 语义分析

语义分析是由 语义分析器(semantic analyzer) 完成的，它使用语法树和符号表中的信息来检查源程序是否和语言定义的语义一致。语义分析器也收集类型信息，并把这些信息放在语法树或者符号表中，以便后续的中间代码生成器使用。

语义分析会进行类型检查(type checking)，这是语义分析器的一个最重要的功能。编译器会检查每个运算符是否具有匹配的运算分量。举个例子比如设计语言要求一个数组的下标是整数，如果你用浮点数作为下标，编译器就会出错。

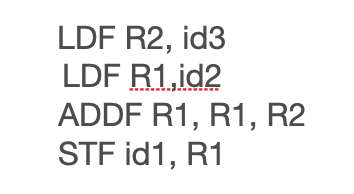
某些程序设计语言比如 Java 会允许自动类型转换(coercion)。如果整数和浮点数进行运算，编译器会把整数转换为浮点数。

# 中间代码生成

在源程序的语法分析和语义分析完成后，很多编译器生成一个明确的低级类机器语言的中间表示。我们可以把中间表示形式看作是抽象，中间形式的代码应该具有两个重要的性质：易于生成，并且能够轻松的被翻译。一般常用的一种是 三地址指令(three-address instructions)的中间表示形式。我们后面会细说。

# 代码生成

代码生成通过中间代码作为输入，并把它映射为目标语言。如果目标语言是机器代码的话，那么必须要为每个变量分配寄存器或内存位置。解释一下上面的运行结果。



每个指令的第一个运算分量指定了一个目标地址，各个指令中的 F 告诉我们它处理的是 浮点数, 上面代码首先把 id3 装载进 R2 寄存器中，然后把 id2 装载进 R1 寄存器中，再对 R1 目标进行 R1 和 R2 寄存器相加的操作。最后把寄存器 R1 的值存放到 id1 的地址中。

# 符号表管理

我们上面提到了符号表的概念，它是一个编译器很重要的功能。符号表能够记录源程序中使用变量的名称，并收集和每个名称相关的属性信息。它相当于一个秘书的作用。符号表还记录了每个变量名字的条目。后面我们会详细的介绍符号表。

# 编译器构造工具

和软件开发一样，写编译器的人可以充分利用现代的软件开发环境进行开发。通常也有 语言编辑器、调试工具、版本管理、测试工具等。除此之外，还需要一些更专业的工具来实现编辑器不同阶段的代码生成。

一些常用的编译器构造工具有

语法分析器生成器：可以根据程序设计语言的语法描述自动生成语法分析器

扫描器生成器：可以根据一个语言的语法单元的正则描述生成词法分析器

语法制导的翻译引擎：用于生成一组遍历分析树并生成中间代码

代码生成器：用于把中间代码转换为目标代码

数据流分析引擎：用于分析输入是如何传递到另一部分的

编译器构造工具：提供用于构造编译器不同阶段的例程