BO 编译器的设计与实现

The Design and Implementation of BO Compiler

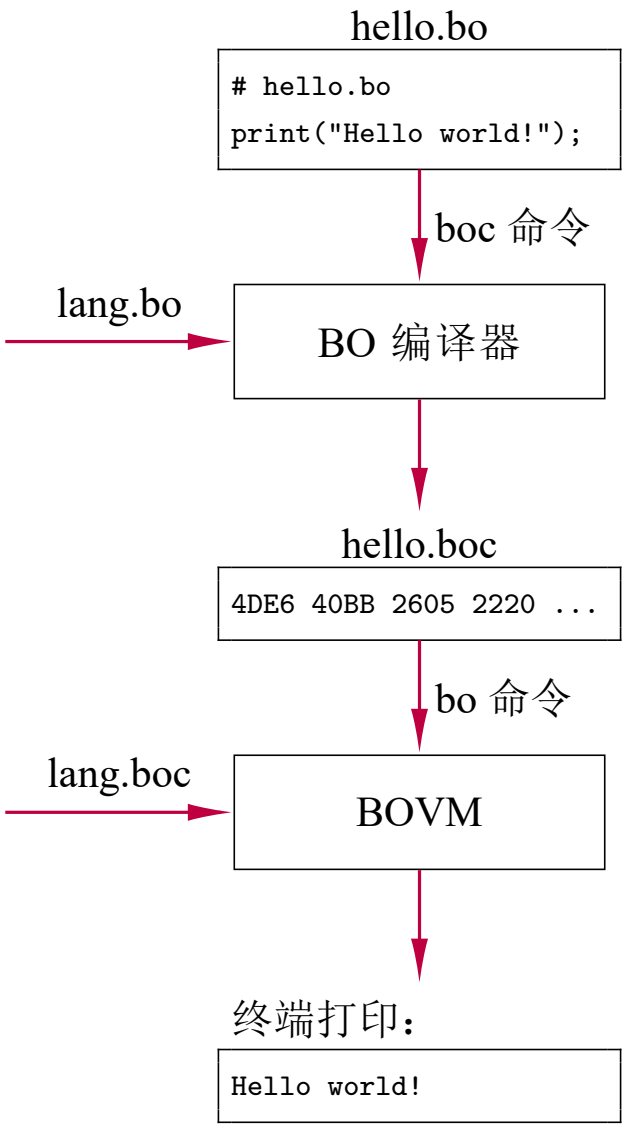
演 讲 人：周博

指导教师：项洁老师

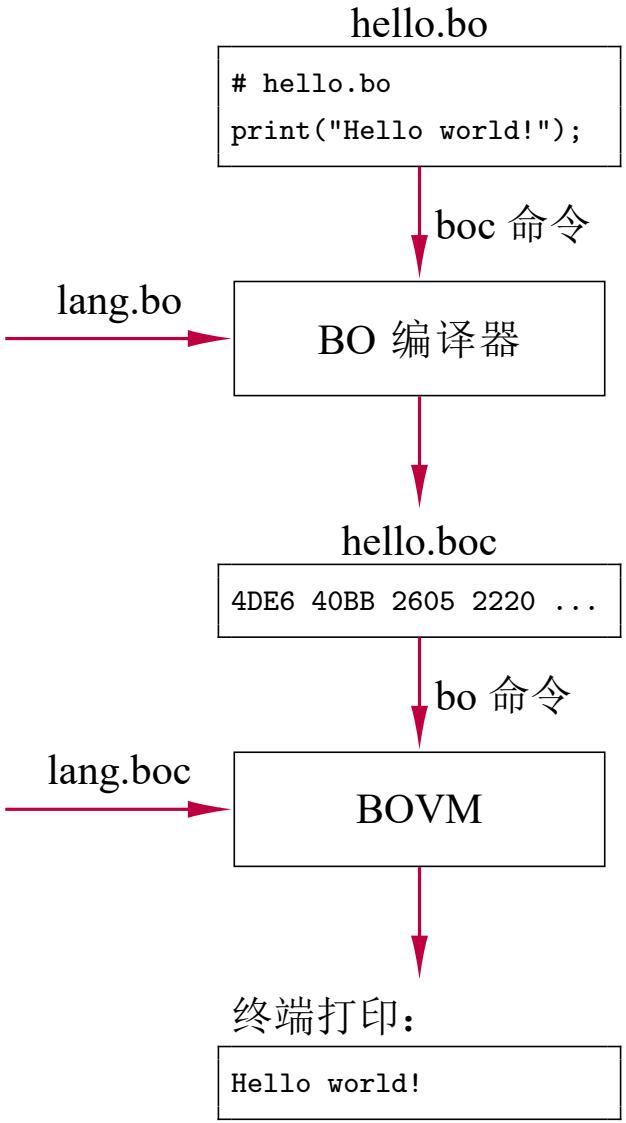
长沙理工大学计算机与通信工程学院

软件工程 18-4 班

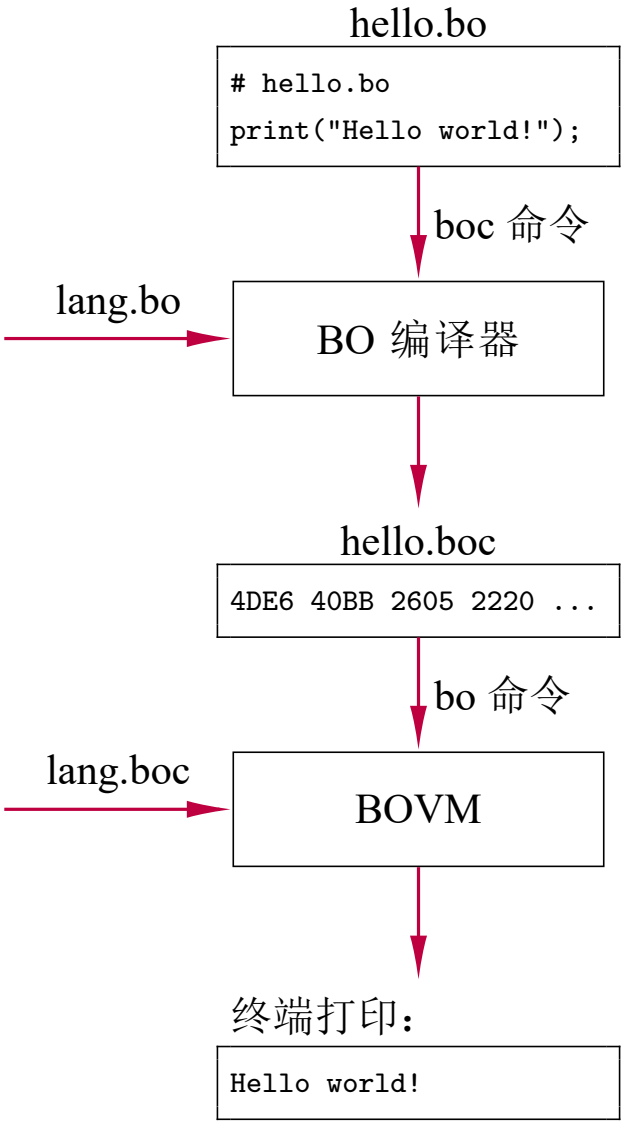
2022 年 6 月 7 日



BO（Binary Object）语言是一种静态类型的编译型语言。

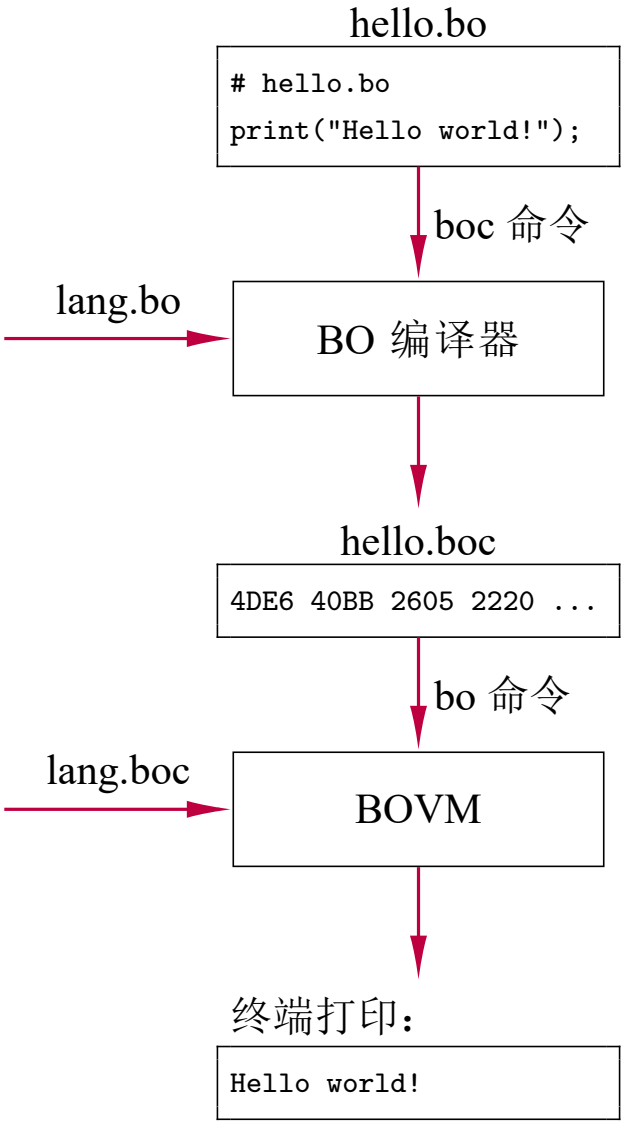


BO（Binary Object）语言是一种静态类型的编译型语言。BO 程序的运行步骤如下：



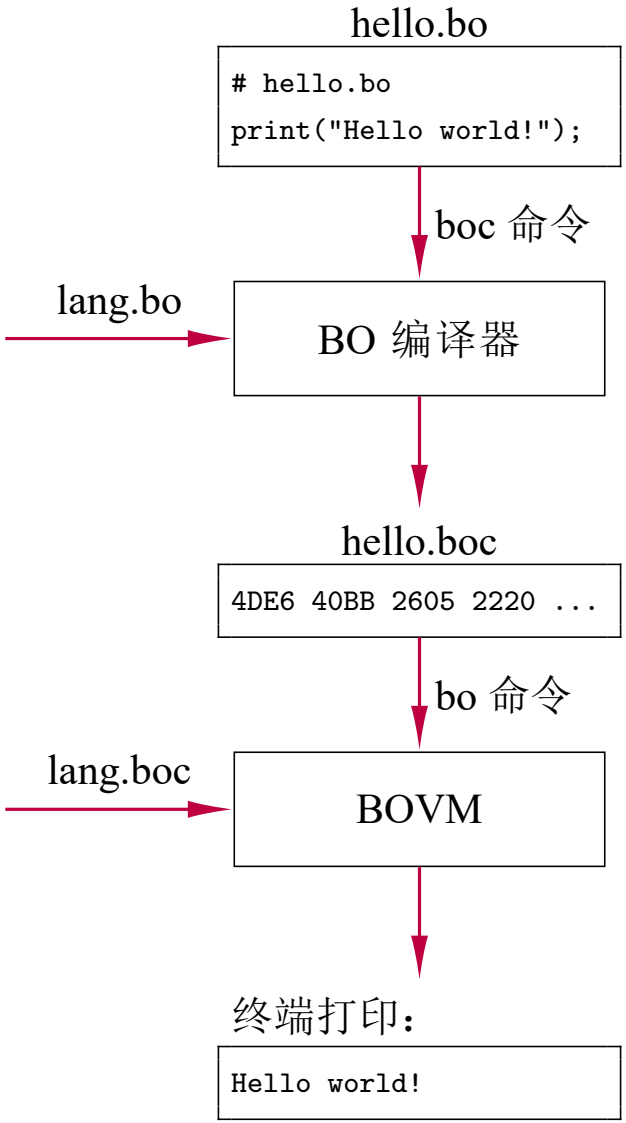
BO（Binary Object）语言是一种静态类型的编译型语言。BO 程序的运行步骤如下：

- 将用 BO 语言写好的源代码以 .bo 格式保存。



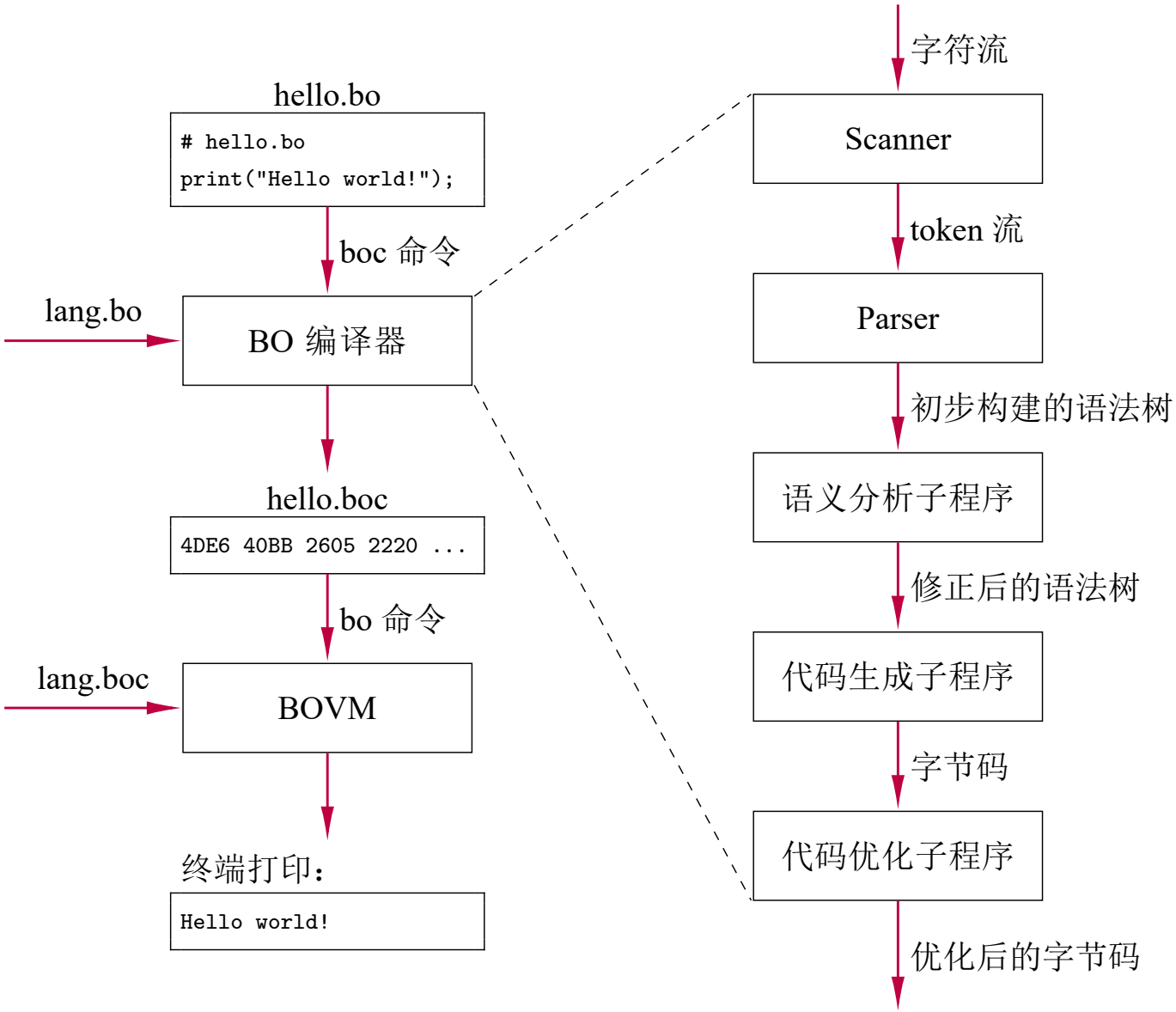
BO（Binary Object）语言是一种静态类型的编译型语言。BO 程序的运行步骤如下：

- 将用 BO 语言写好的源代码以 .bo 格式保存。
- 使用 boc 命令编译 .bo 格式文件，由 BO 编译器生成 .boc 格式的字节码文件。

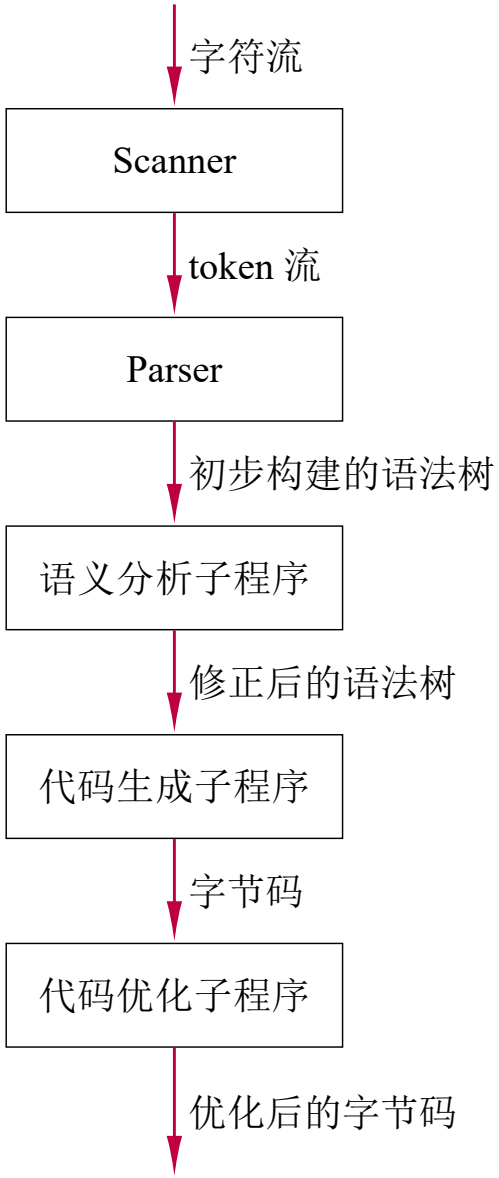


BO（Binary Object）语言是一种静态类型的编译型语言。BO 程序的运行步骤如下：

- 将用 BO 语言写好的源代码以 .bo 格式保存。
- 使用 boc 命令编译 .bo 格式文件，由 BO 编译器生成 .boc 格式的字节码文件。
- 使用 bo 命令执行 .boc 格式的字节码文件，由 BOVM 虚拟机运行得到结果。

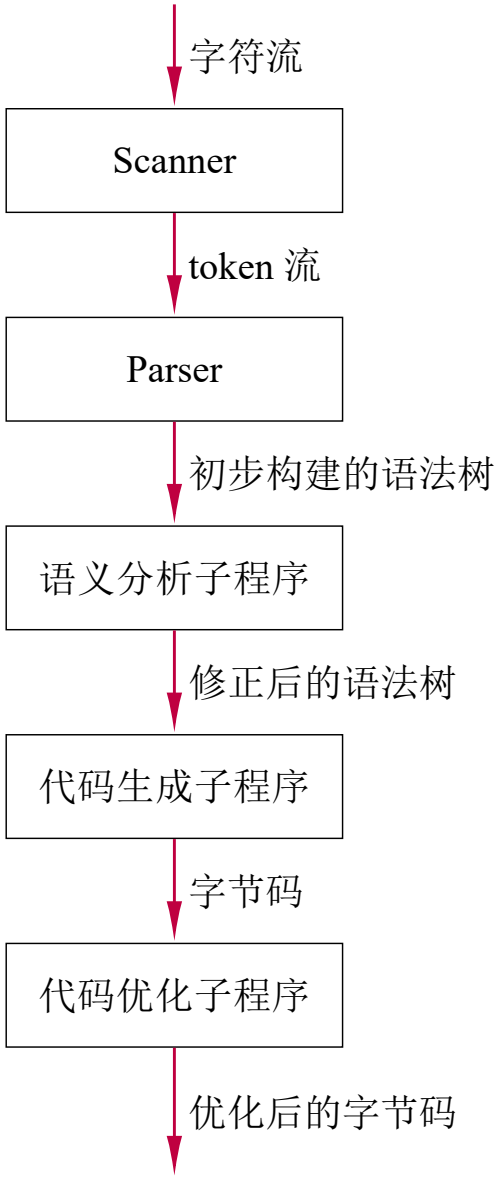

hello.bo
# hello.bo
print("Hello world!");
boc 命令
lang.bo
BO 编译器
hello.boc
4DE6 40BB 2605 2220 ...
bo 命令
lang.boc
BOVM
终端打印：
Hello world!
字符流
Scanner
token 流
Parser
初步构建的语法树
语义分析子程序
修正后的语法树
代码生成子程序
字节码
代码优化子程序
优化后的字节码

本课题的研究重点在于如何实现一个能满足下列要求的 BO 编译器：



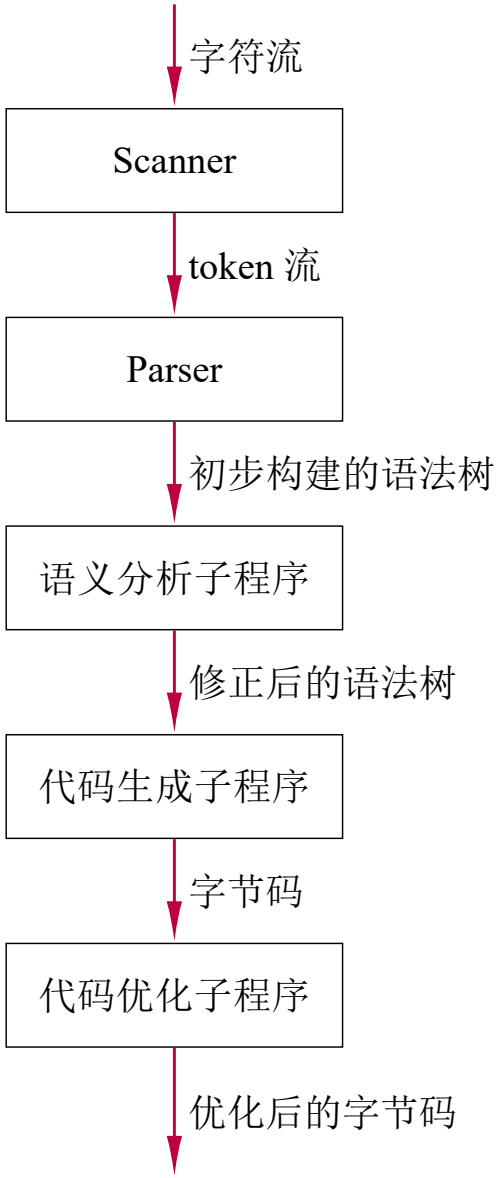
本课题的研究重点在于如何实现一个能满足下列要求的 BO 编译器：

- **正确编译 BO 源程序，生成 `.boc` 格式的字节码文件。**



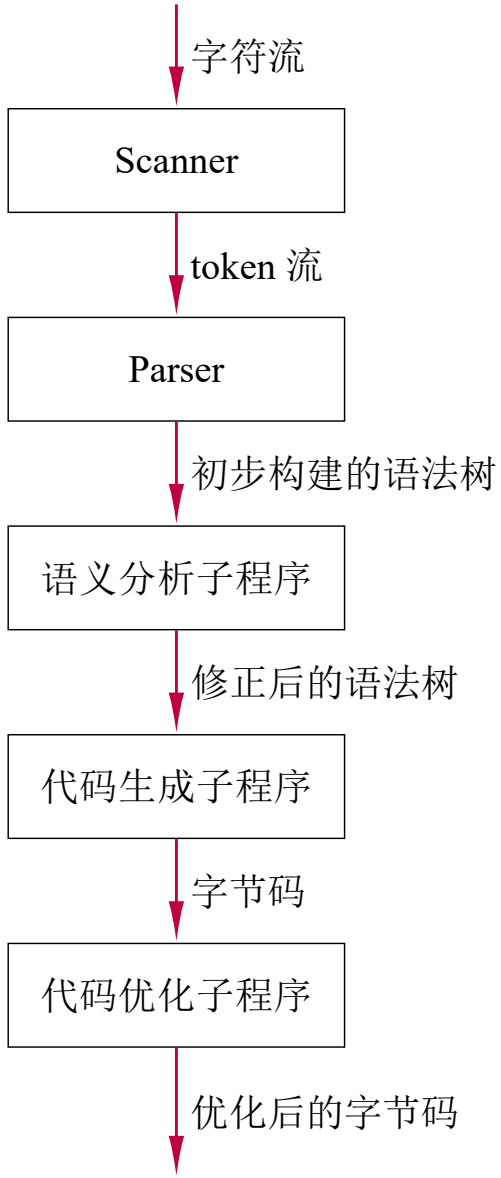
本课题的研究重点在于如何实现一个能满足下列要求的 BO 编译器：

- 正确编译 BO 源程序，生成 .boc 格式的字节码文件。
- 为存在错误的 BO 源程序报告尽可能准确的出错信息。



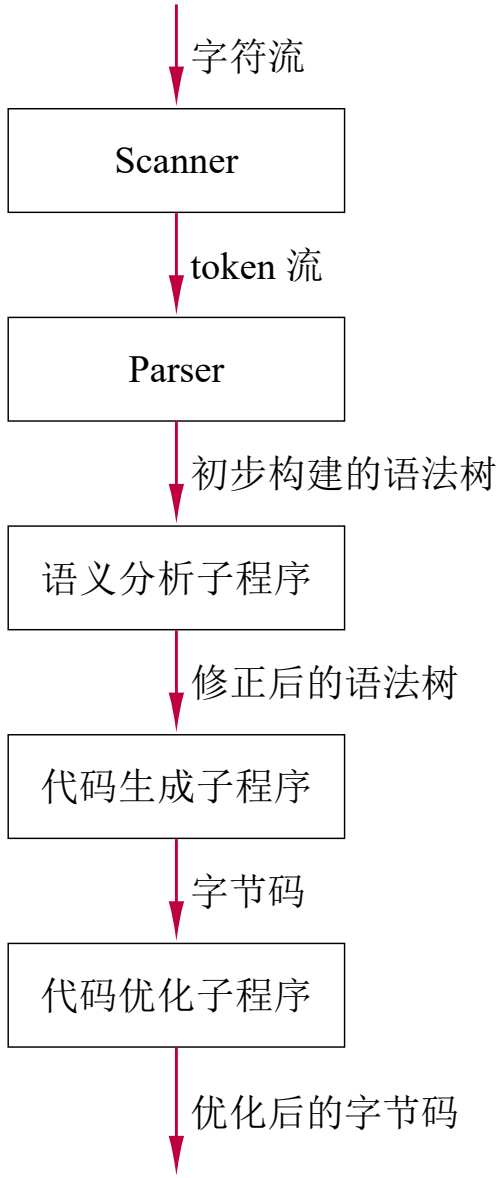
本课题的研究重点在于如何实现一个能满足下列要求的 BO 编译器：

- 正确编译 BO 源程序，生成 .boc 格式的字节码文件。
- 为存在错误的 BO 源程序报告尽可能准确的出错信息。
- 在编译阶段尽可能多地发现 BO 源程序的语义错误。



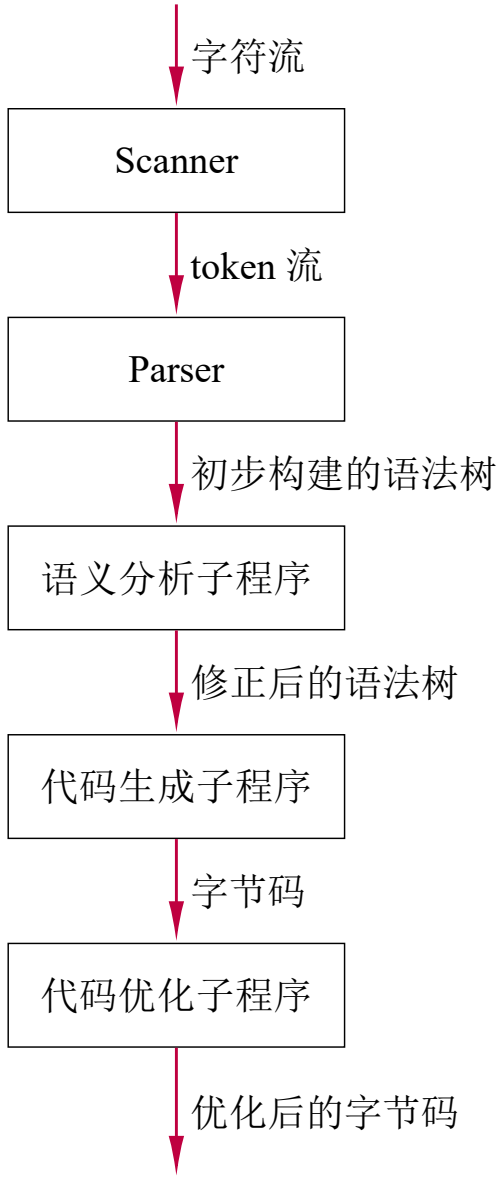
本课题的研究重点在于如何实现一个能满足下列要求的 BO 编译器：

- 正确编译 BO 源程序，生成 .boc 格式的字节码文件。
- 为存在错误的 BO 源程序报告尽可能准确的出错信息。
- 在编译阶段尽可能多地发现 BO 源程序的语义错误。
- 生成尽可能优化的字节码程序。

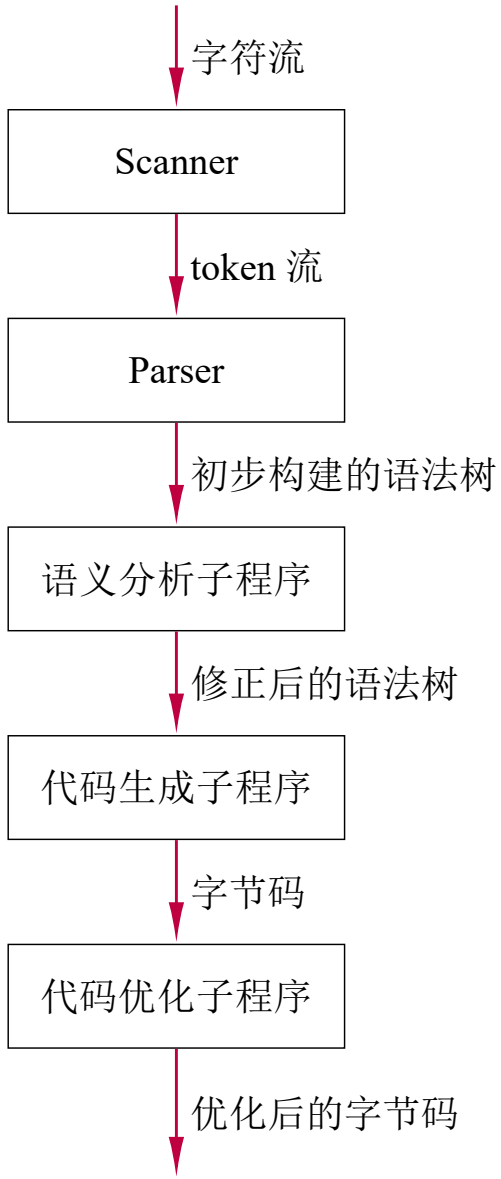


本课题的研究重点在于如何实现一个能满足下列要求的 BO 编译器：

- 正确编译 BO 源程序，生成 `.boc` 格式的字节码文件。
- 为存在错误的 BO 源程序报告尽可能准确的出错信息。
- 在编译阶段尽可能多地发现 BO 源程序的语义错误。
- 生成尽可能优化的字节码程序。

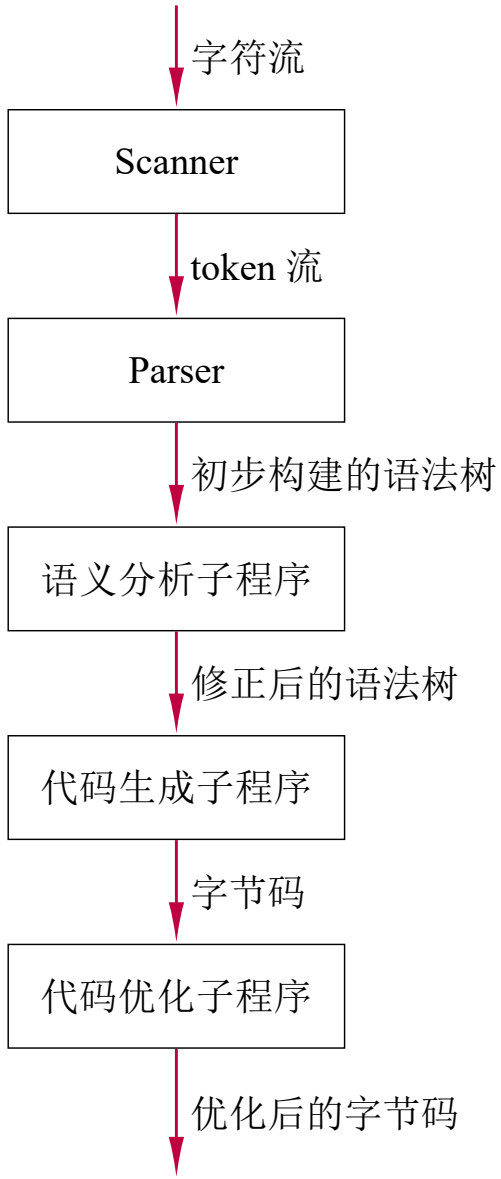


BO 编译器具有如下功能：



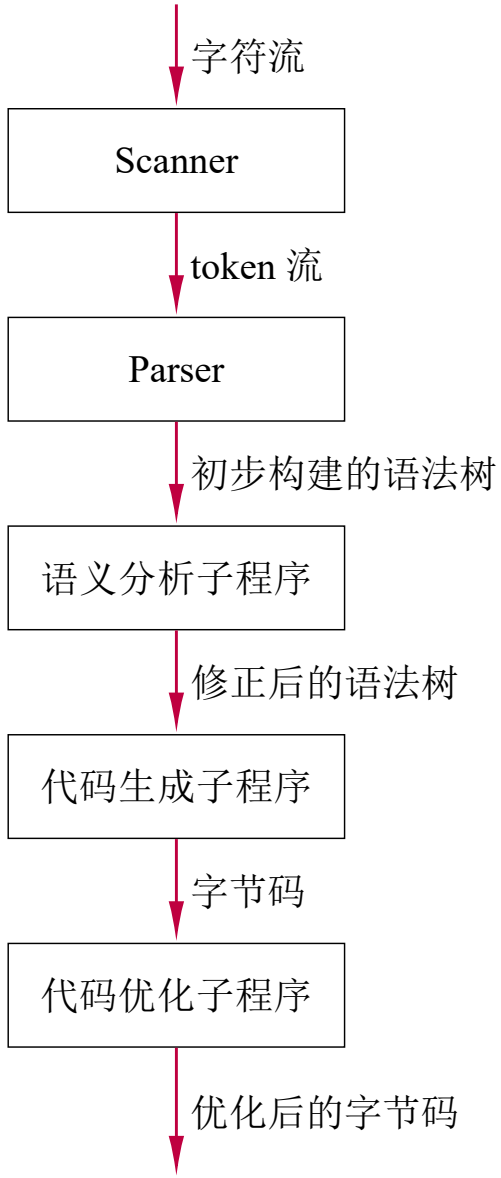
BO 编译器具有如下功能：

- 词法分析



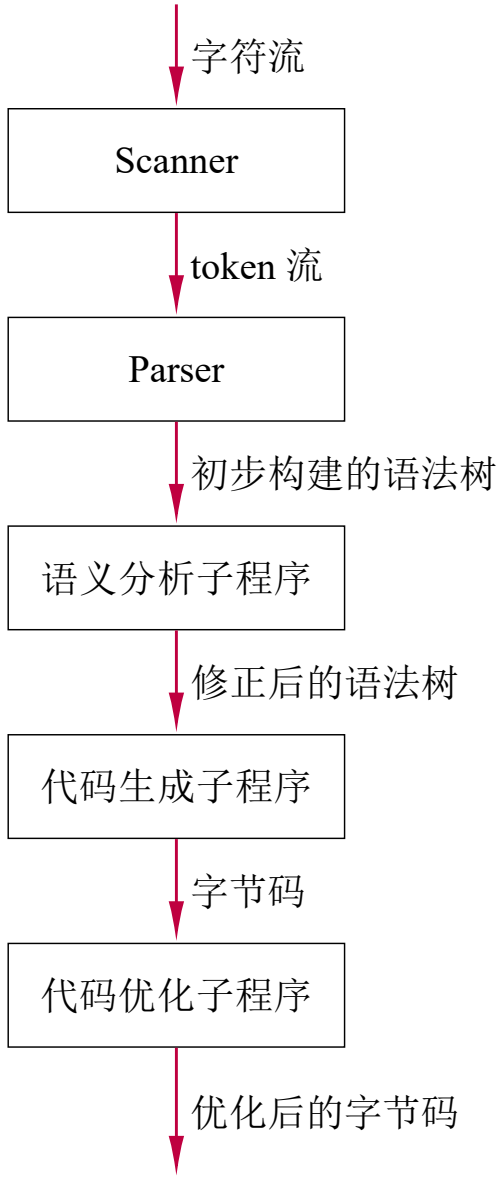
BO 编译器具有如下功能：

- 词法分析
- 语法分析



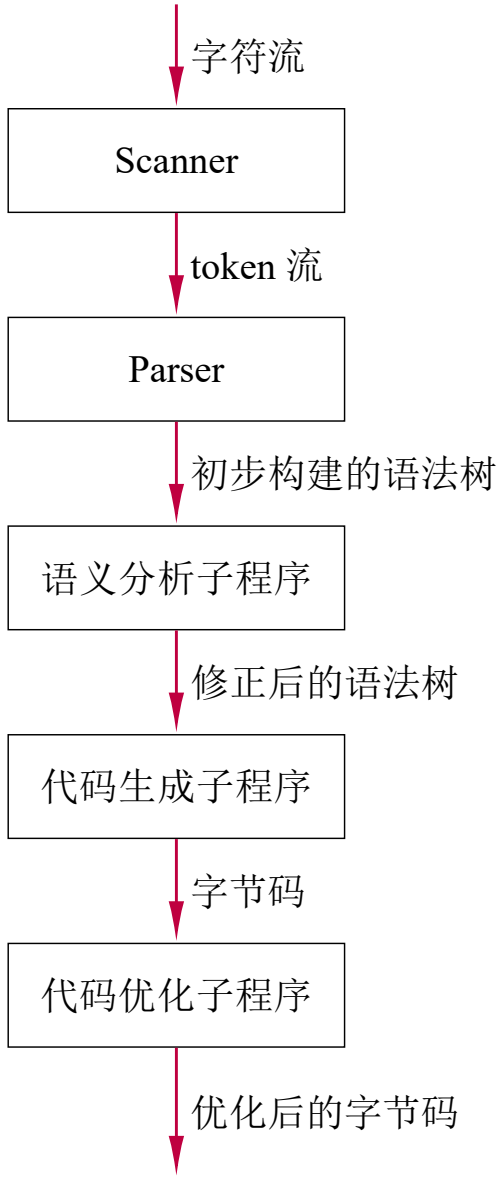
BO 编译器具有如下功能：

- 词法分析
- 语法分析
- 语义分析



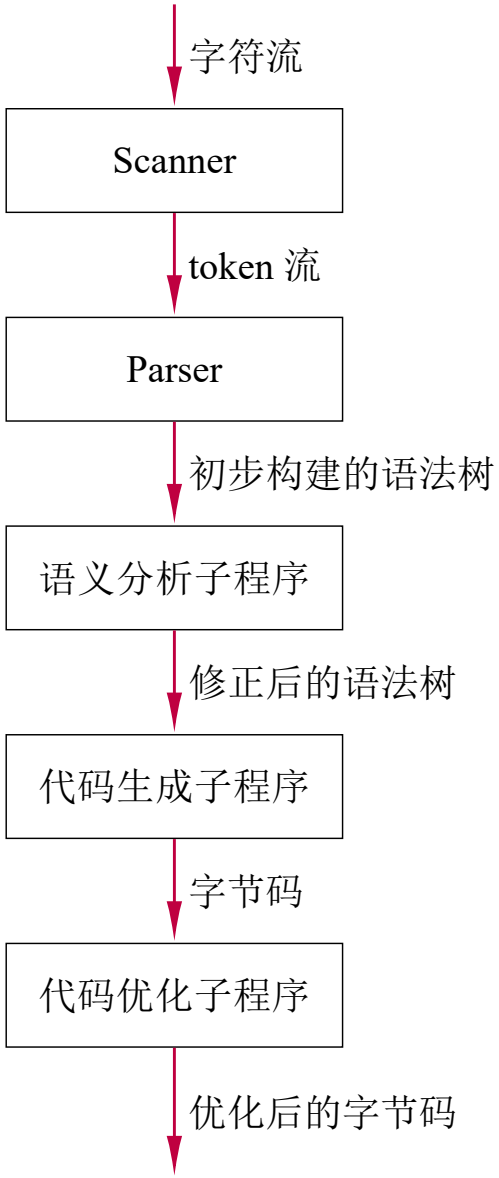
BO 编译器具有如下功能：

- 词法分析
- 语法分析
- 语义分析
- 代码生成



BO 编译器具有如下功能：

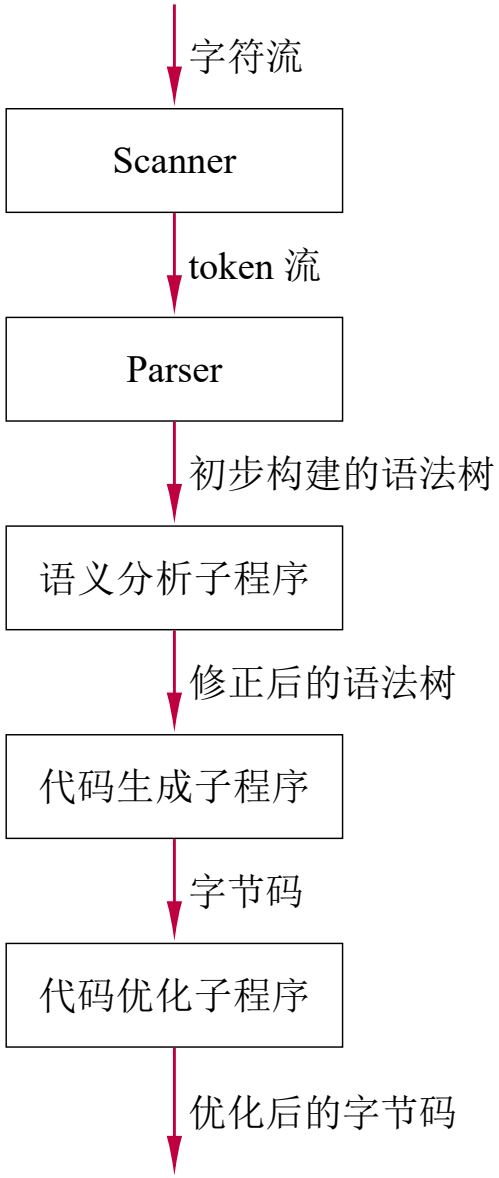
- 词法分析
- 语法分析
- 语义分析
- 代码生成
- 代码优化



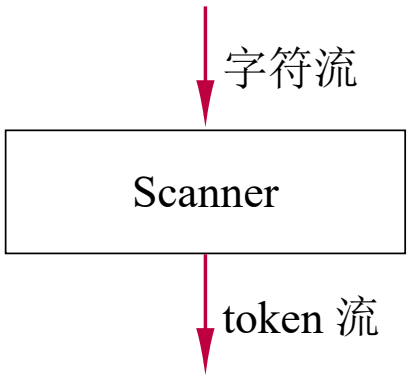
BO 编译器具有如下功能：

- 词法分析
- 语法分析
- 语义分析
- 代码生成
- 代码优化

每个功能模块都会用到符号表和出错处理程序。



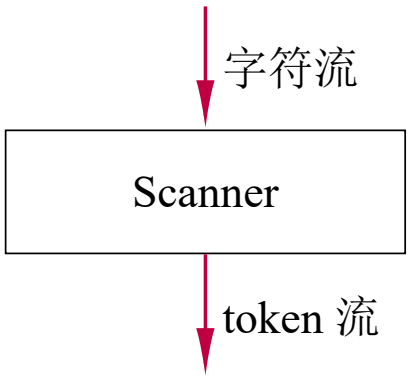
```
if [1 = a] { res : "ok"; }
```



⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨{⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨}⟩

词法分析器（Scanner）的工作流程：

```
if [1 = a] { res : "ok"; }
```

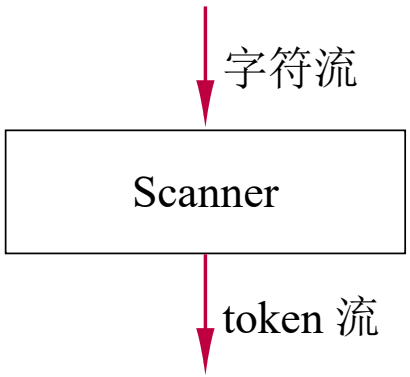


⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨{⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨}⟩

词法分析器（Scanner）的工作流程：

- 从源文件中不断读入一个个字符。

```
if [1 = a] { res : "ok"; }
```

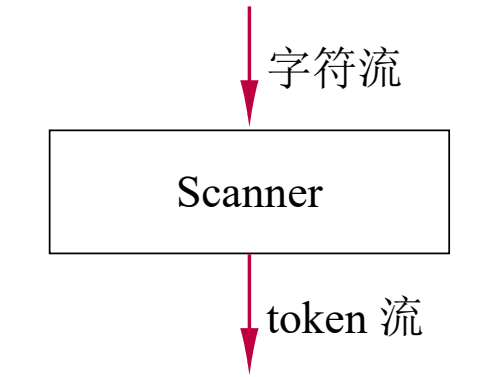


⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨{⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨}⟩

词法分析器（Scanner）的工作流程：

- 从源文件中不断读入一个个字符。
- 将单个字符组合成一个个词素。

```
if [1 = a] { res : "ok"; }
```

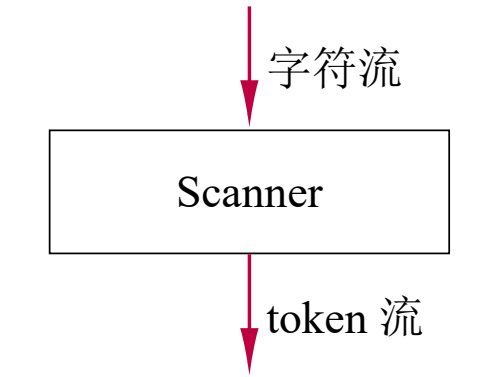


⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨{⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨}⟩

词法分析器（Scanner）的工作流程：

- 从源文件中不断读入一个个字符。
- 将单个字符组合成一个个词素。
- 以词法单元（token）的形式返回给调用者。

```
if [1 = a] { res : "ok"; }
```



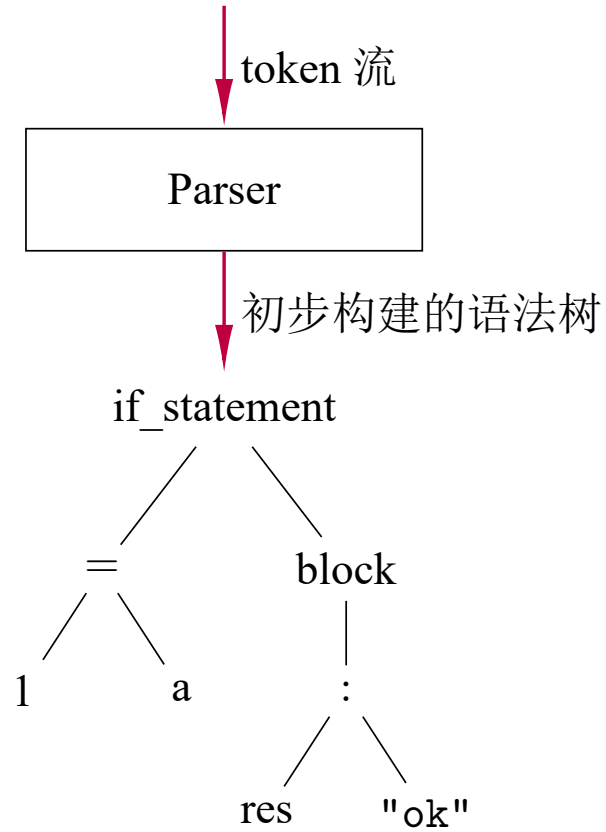
⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨{⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨}⟩

词法分析器（Scanner）的工作流程：

- 从源文件中不断读入一个个字符。
- 将单个字符组合成一个个词素。
- 以词法单元（token）的形式返回给调用者。

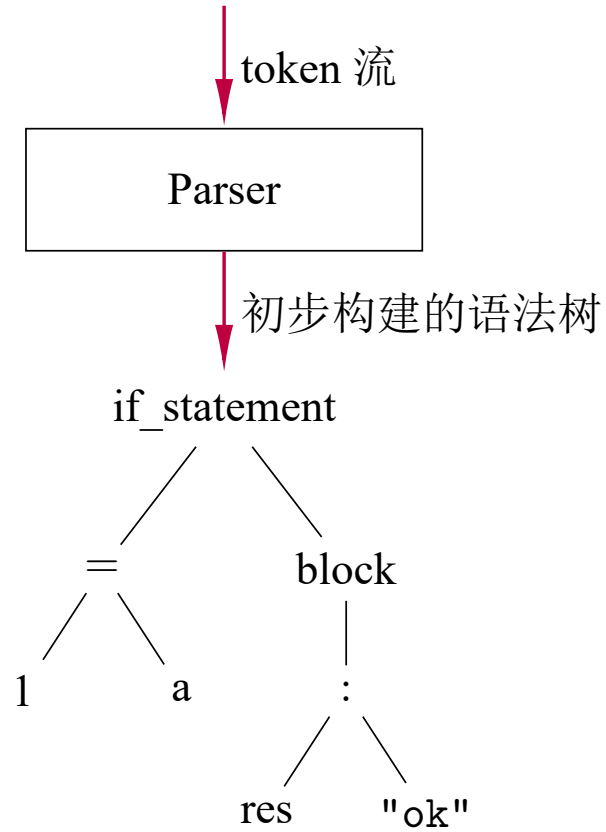
词法分析器发现非预期的字符时会报告相应错误。

⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨ { ⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨ } ⟩



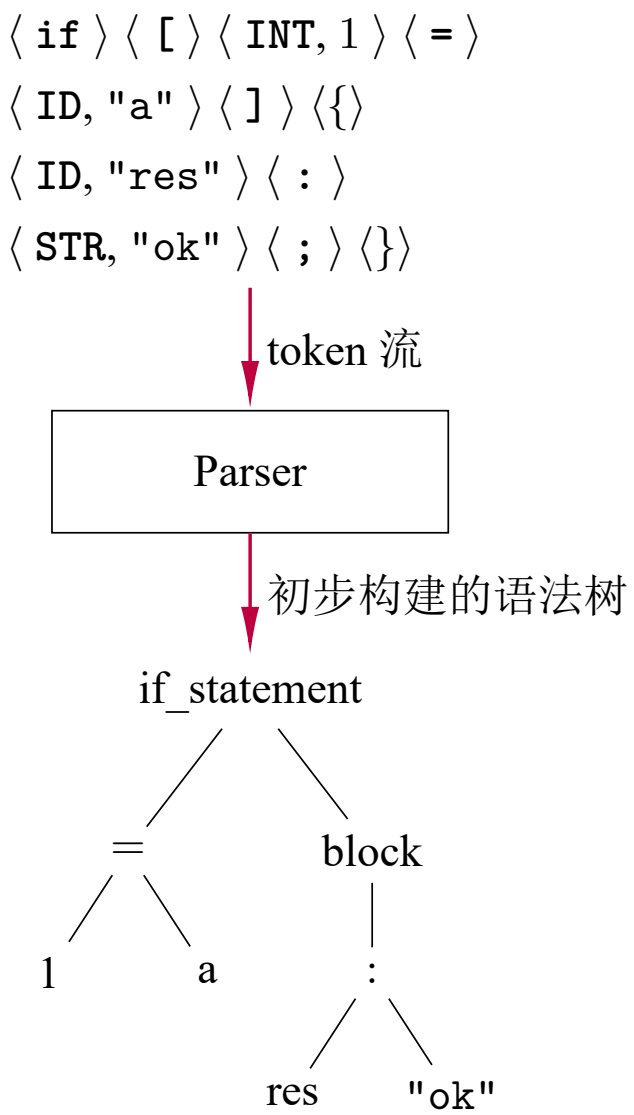
语法分析器（Scanner）的工作流程：

⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨{⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨}⟩



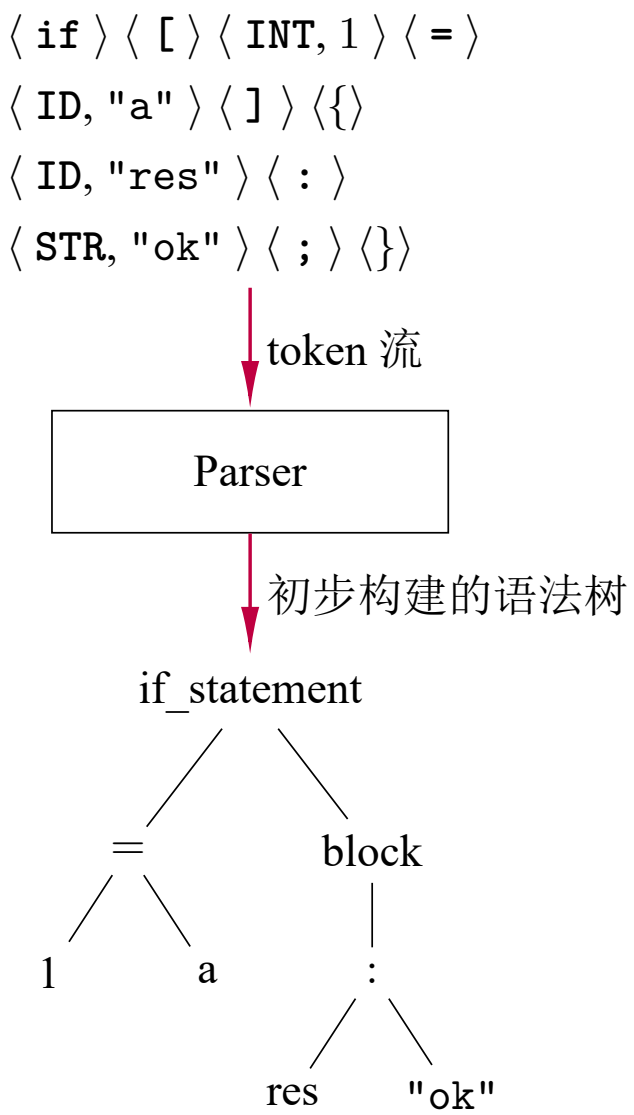
语法分析器（Scanner）的工作流程：

- 调用词法分析器获取一个个词法单元。



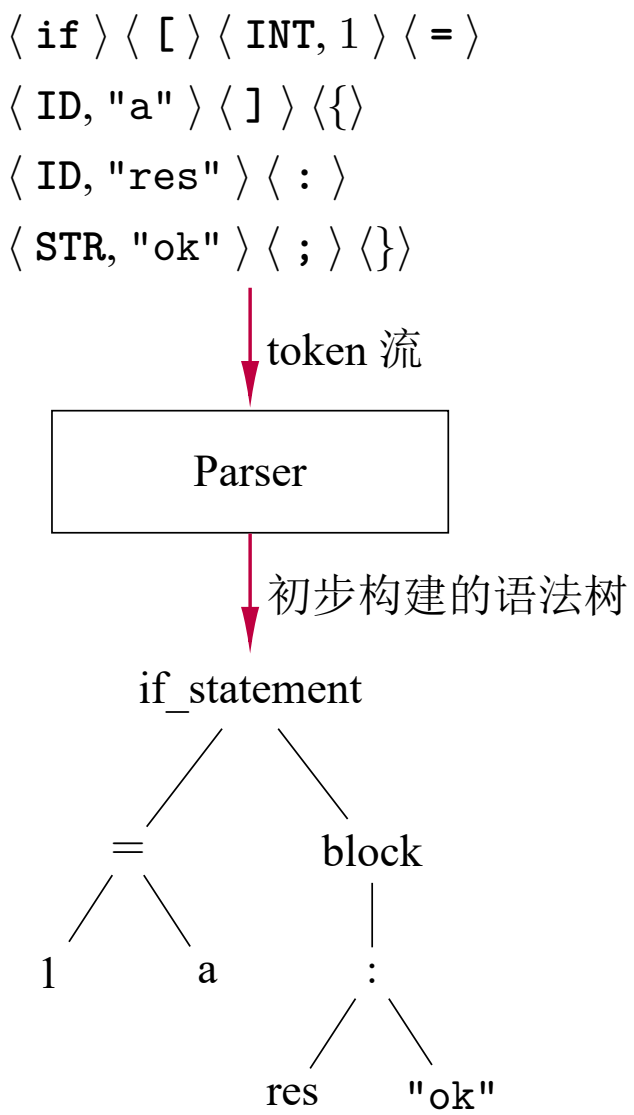
语法分析器（Scanner）的工作流程：

- 调用词法分析器获取一个个词法单元。
- 不断移进词法单元并进行规约。



语法分析器（Scanner）的工作流程：

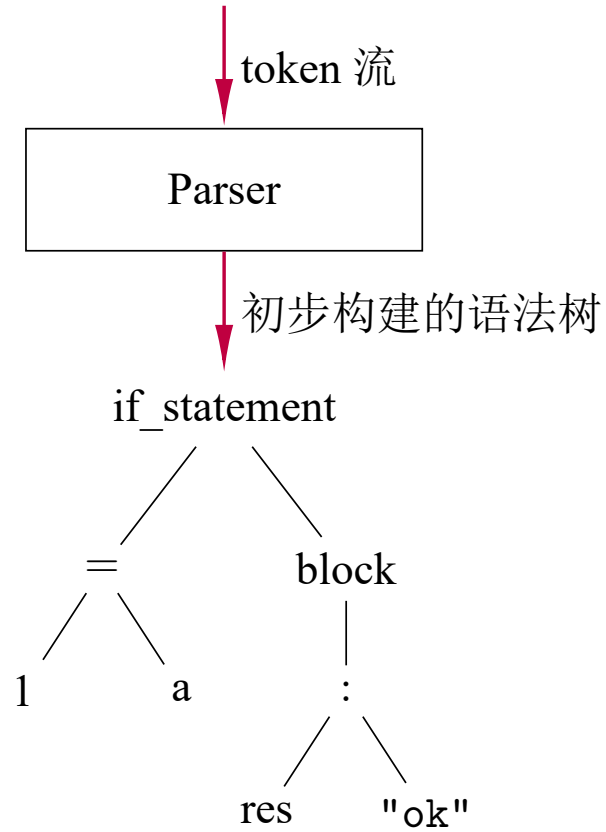
- 调用词法分析器获取一个个词法单元。
- 不断移进词法单元并进行规约。
- 在规约的过程中初步构建一颗抽象语法树。



语法分析器（Scanner）的工作流程：

- 调用词法分析器获取一个个词法单元。
- 不断移进词法单元并进行规约。
- 在规约的过程中初步构建一颗抽象语法树。
- 规约完成后得到的抽象语法树即为源程序的一个中间表示。

⟨ if ⟩ ⟨ [⟩ ⟨ INT, 1 ⟩ ⟨ = ⟩
⟨ ID, "a" ⟩ ⟨] ⟩ ⟨{⟩
⟨ ID, "res" ⟩ ⟨ : ⟩
⟨ STR, "ok" ⟩ ⟨ ; ⟩ ⟨}⟩

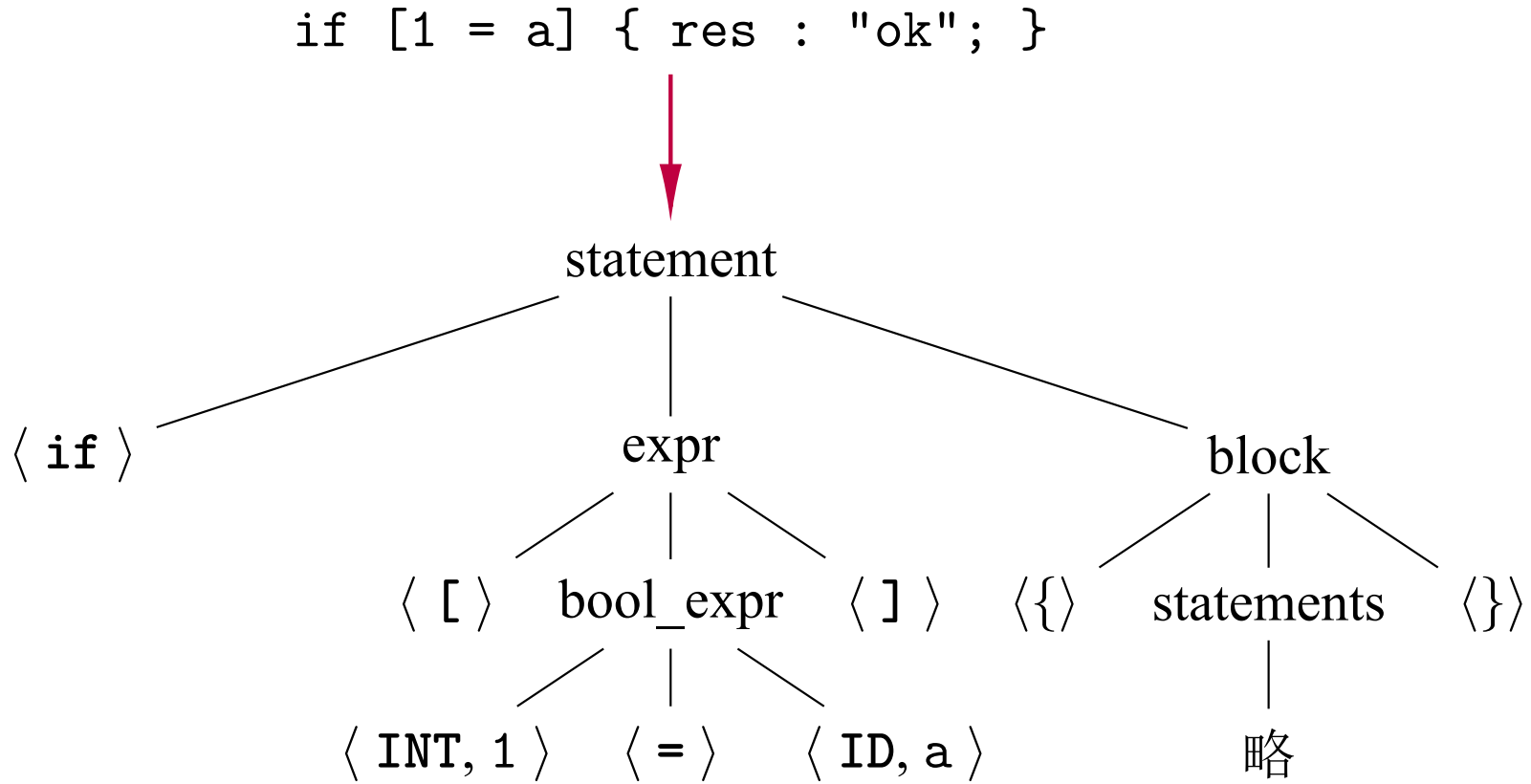


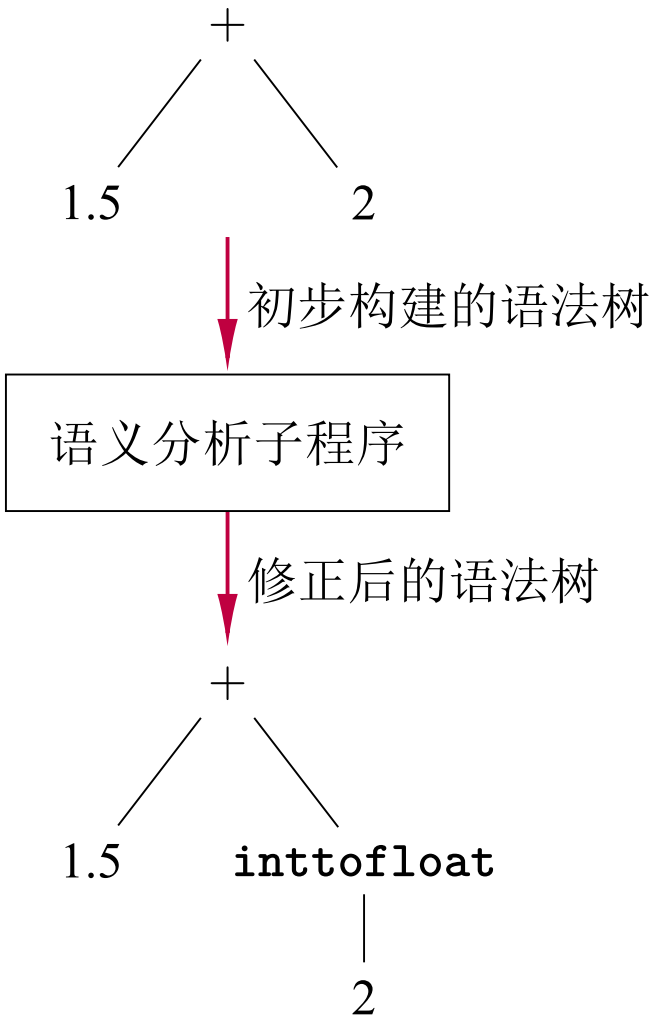
语法分析器（Scanner）的工作流程：

- 调用词法分析器获取一个个词法单元。
- 不断移进词法单元并进行规约。
- 在规约的过程中初步构建一颗抽象语法树。
- 规约完成后得到的抽象语法树即为源程序的一个中间表示。

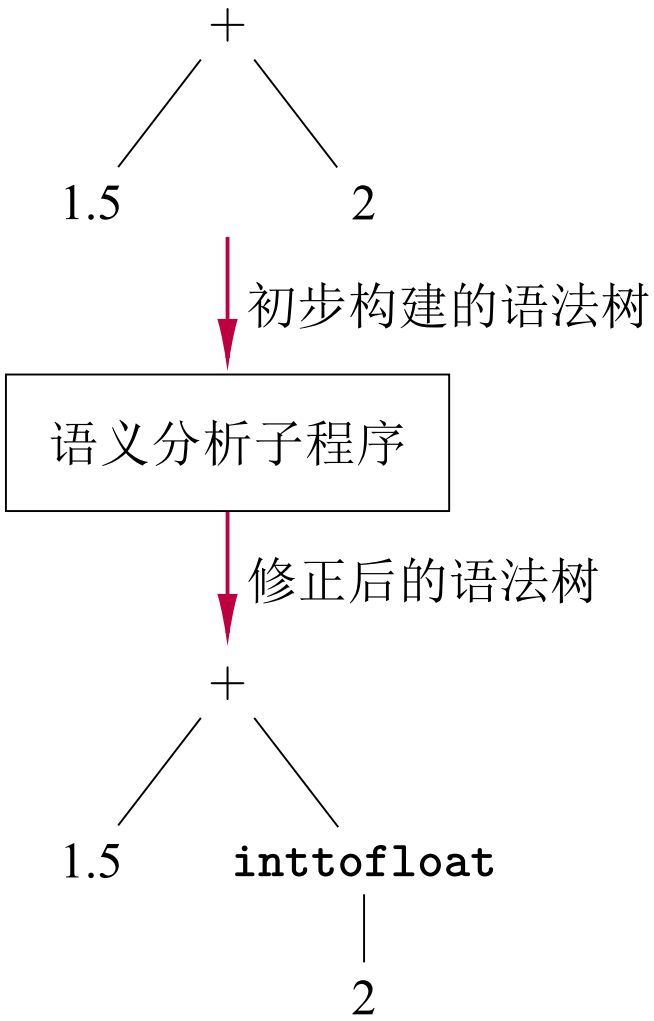
语法分析器发现非预期的词法单元时会报告相应错误。

语法分析器进行规约的过程，实际上是对语法分析树自底向上遍历的过程。

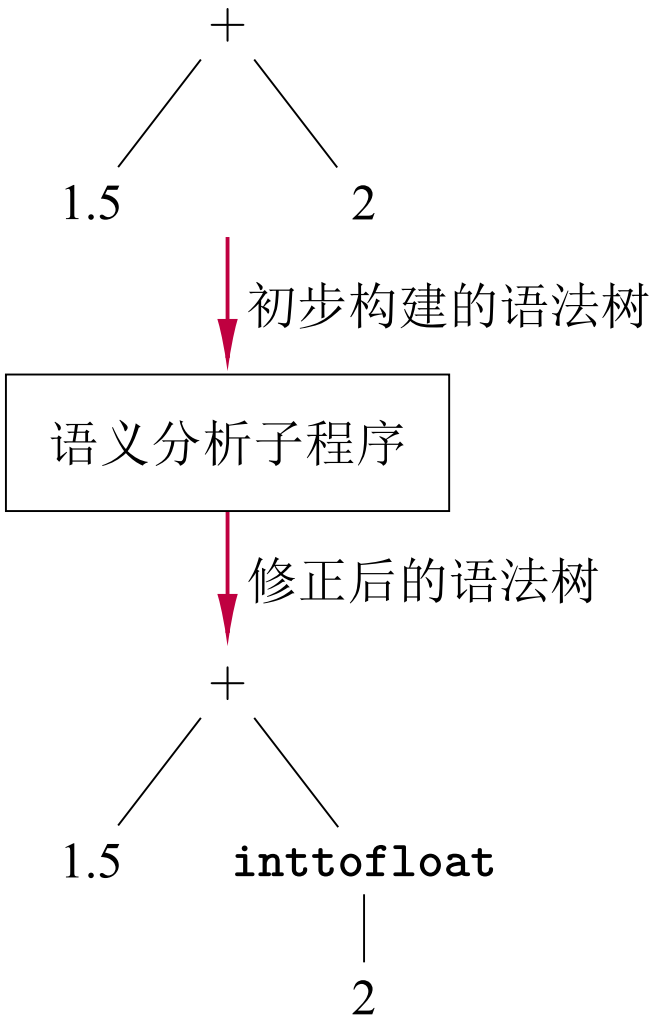




- 语法分析阶段已经初步构建出了一颗抽象语法树。

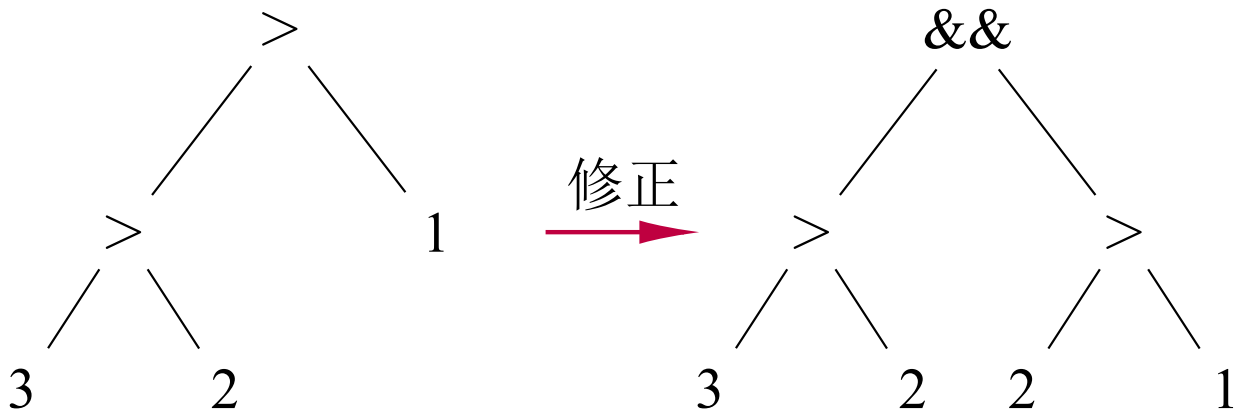


- 语法分析阶段已经初步构建出了一颗抽象语法树。
- 语义分析阶段要做的就是遍历抽象语法树，进行类型检查、变量作用范围检查、以及类成员访问合法性检查。

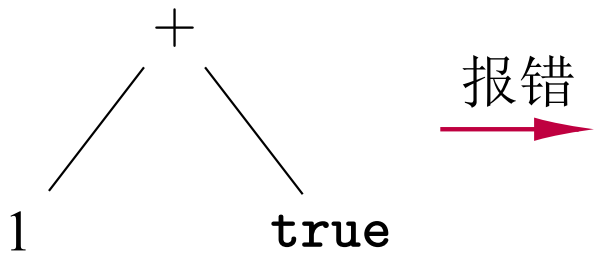


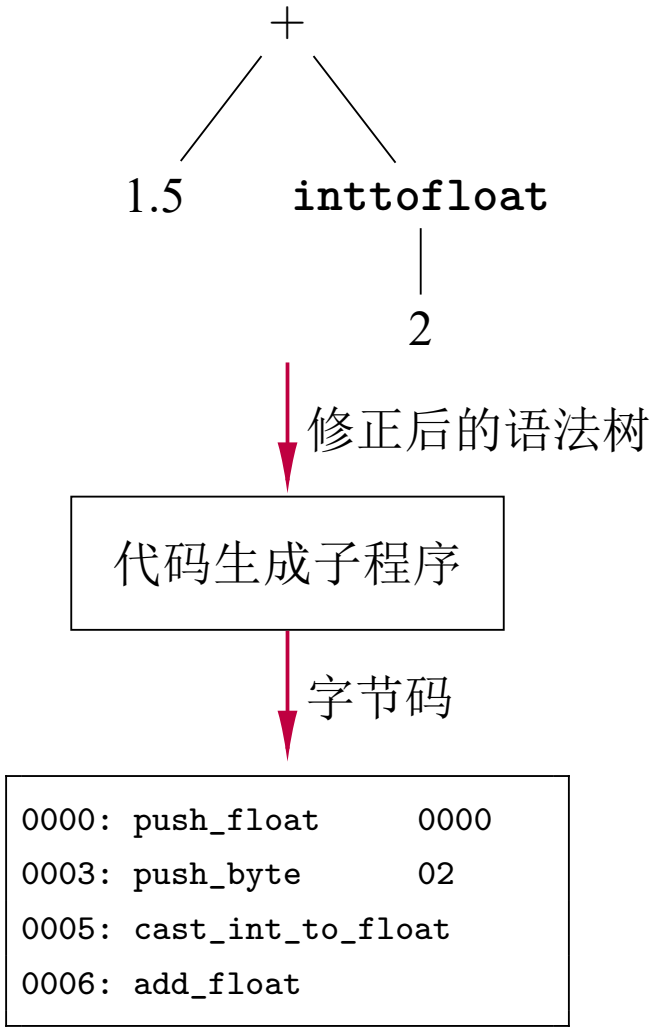
- 语法分析阶段已经初步构建出了一颗抽象语法树。
- 语义分析阶段要做的就是遍历抽象语法树，进行类型检查、变量作用范围检查、以及类成员访问合法性检查。
- 一旦语义分析子程序发现语义问题，就会进行修正或者直接报错。

BO 语言将 $3 > 2 > 1$ 解释为 $3 > 2 \&\& 2 > 1$，因此需要在语法分析阶段进行修正。

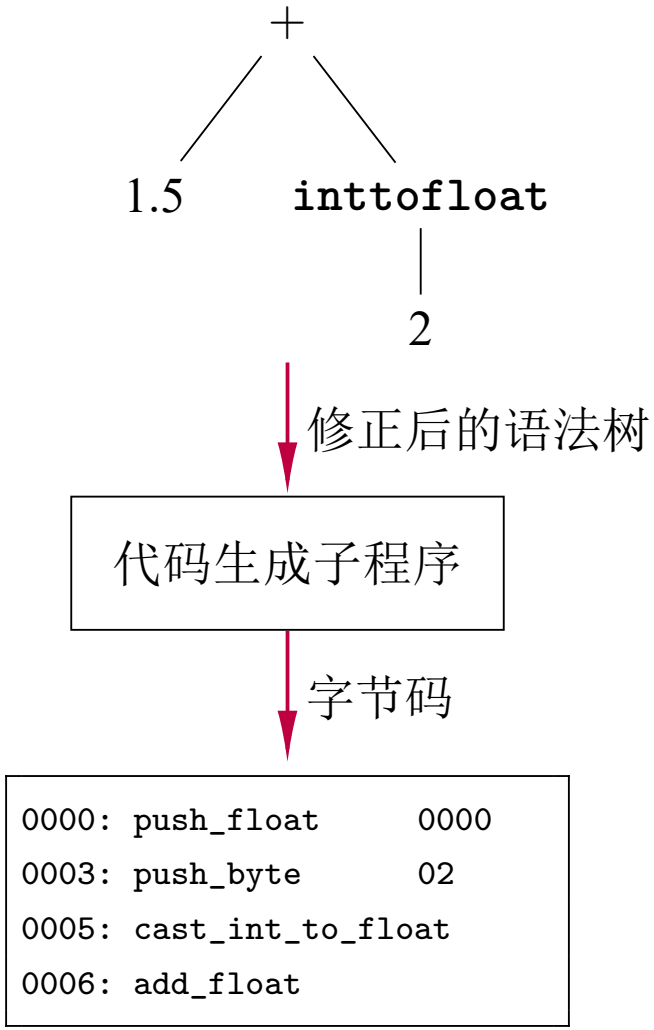


BO 语言不支持整数类型与布尔类型做算术运算。如果出现形如 $1 + \texttt{true}$ 的表达式，会在语义分析阶段发现表达式类型不匹配。

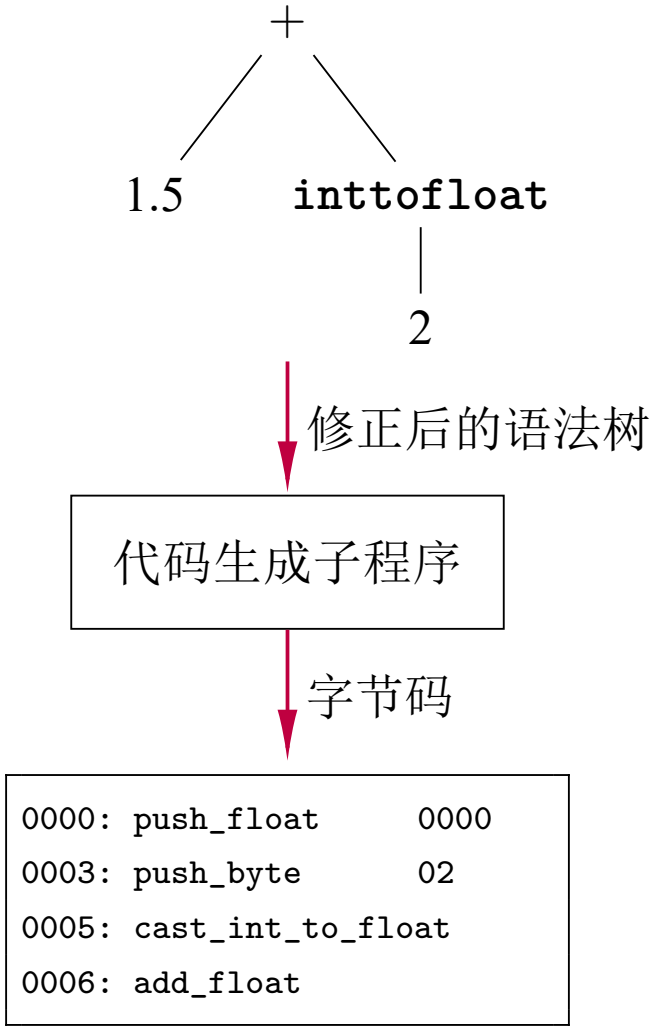




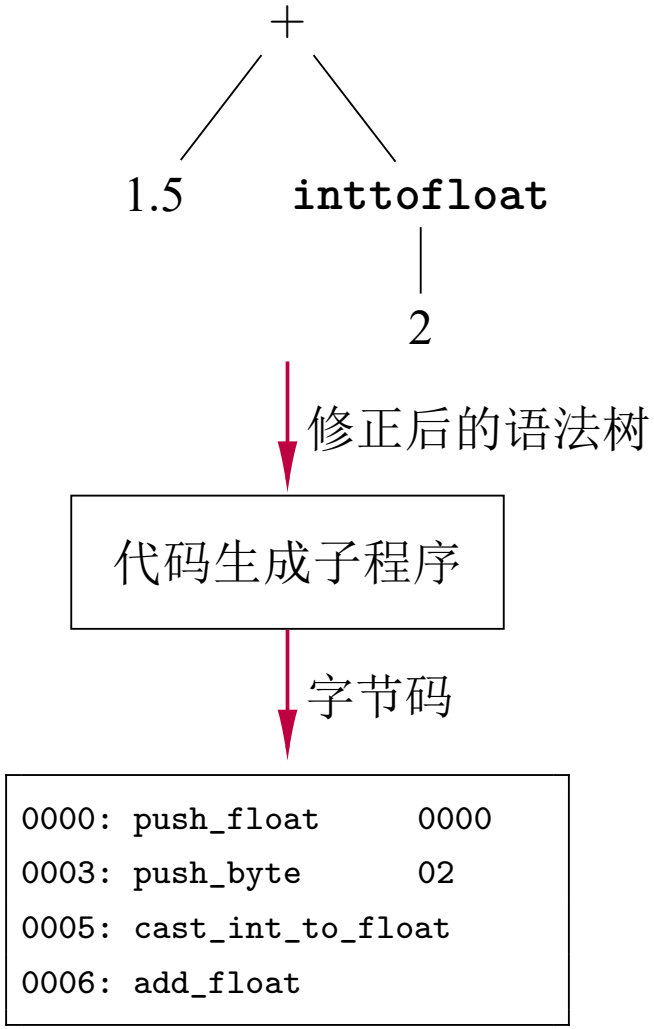
- 对修正后的语法树进行后序遍历，即可生成字节码。



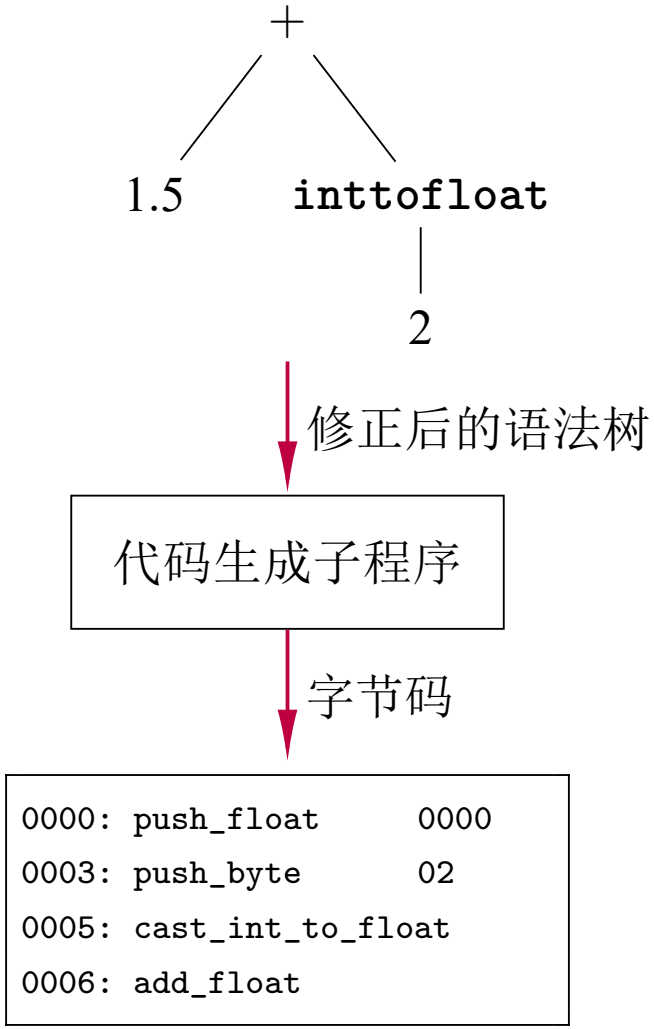
- 对修正后的语法树进行后序遍历，即可生成字节码。
- 一条字节码指令由操作码和操作数组成。



- 对修正后的语法树进行后序遍历，即可生成字节码。
- 一条字节码指令由操作码和操作数组成。
- 字节码指令最多占 3 个字节，其中操作码固定占 1 个字节。



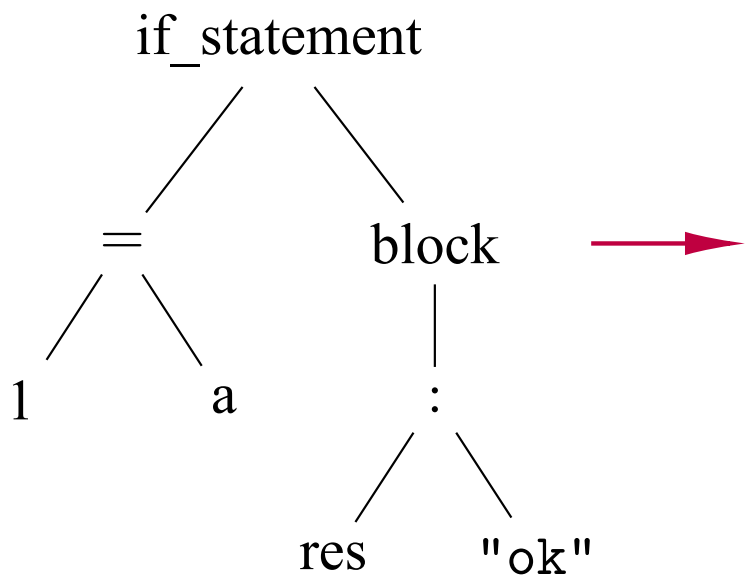
- 对修正后的语法树进行后序遍历，即可生成字节码。
- 一条**字节码指令**由操作码和操作数组成。
- 字节码指令最多占 3 个字节，其中操作码固定占 1 个字节。
- 对于超过 2 个字节的操作数，BO 编译器会将其加入**常量池**，并通过索引访问。



- 对修正后的语法树进行后序遍历，即可生成字节码。
- 一条**字节码指令**由操作码和操作数组成。
- 字节码指令最多占 3 个字节，其中操作码固定占 1 个字节。
- 对于超过 2 个字节的操作数，BO 编译器会将其加入**常量池**，并通过索引访问。

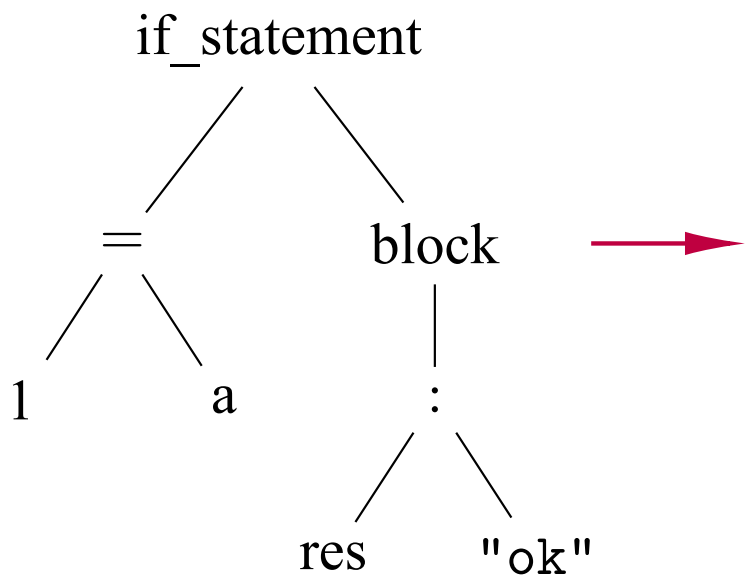
注：为方便演示、本幻灯片中的字节码均以**助忆符**的形式呈现。

为流程控制语句生成代码时，需要两次遍历，才能确定需要跳转的位置的地址。

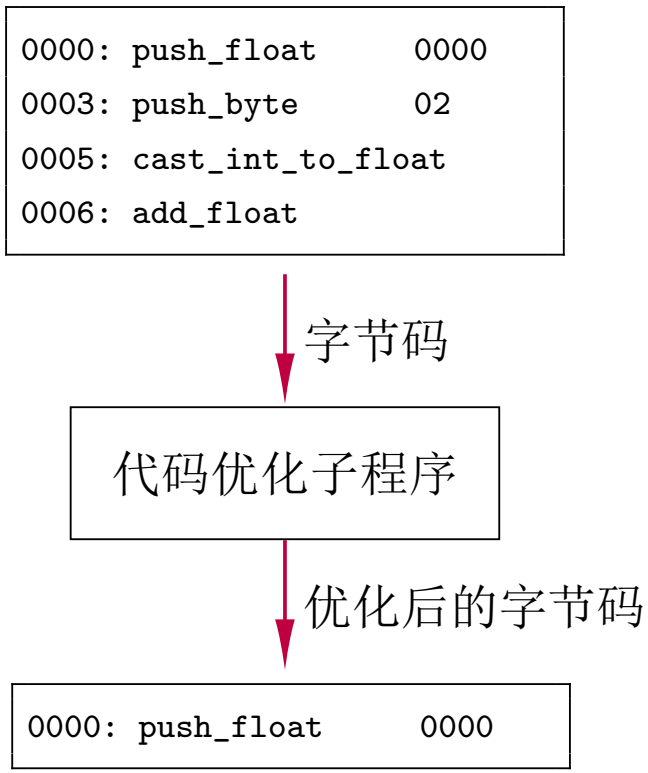



```
0000: push_byte       01
0002: push_static_int 0000
0005: eq_int
0006: jump_if_false   END
0009: push_str        0000
000C: pop_static_object 0001
000F: jump            END
```

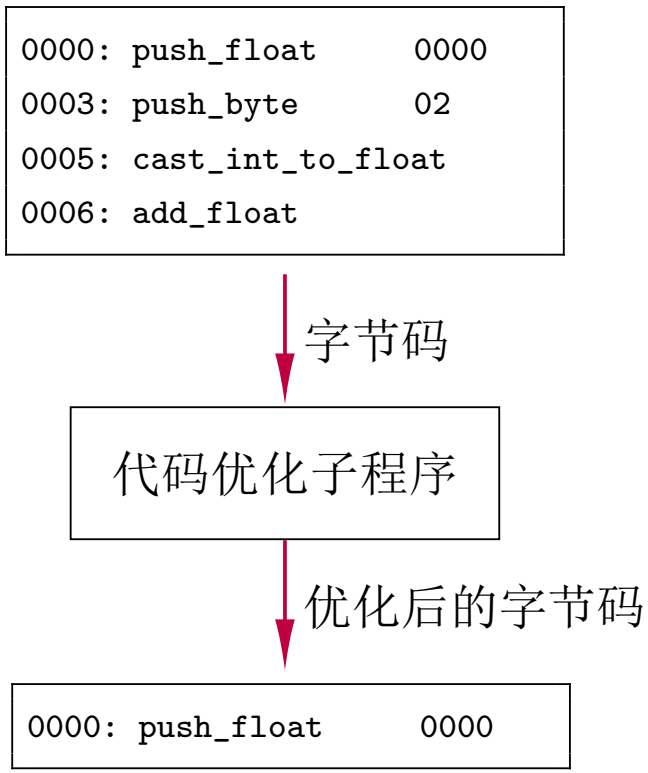
为流程控制语句生成代码时，需要两次遍历，才能确定需要跳转的位置的地址。



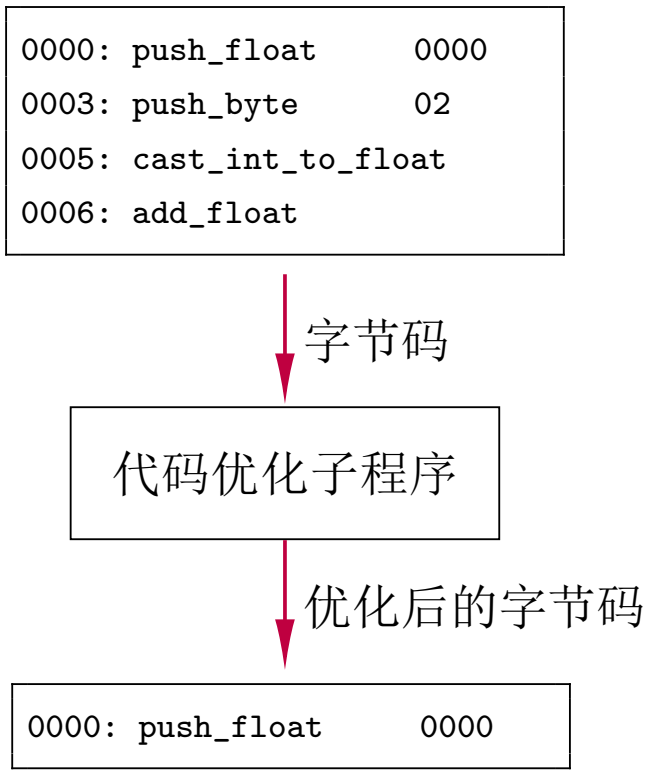

```
0000: push_byte       01
0002: push_static_int 0000
0005: eq_int
0006: jump_if_false   0012
0009: push_str        0000
000C: pop_static_object 0001
000F: jump            0012
```



BO 编译器会对生成的字节码进行简单的常量折叠和死码消除，

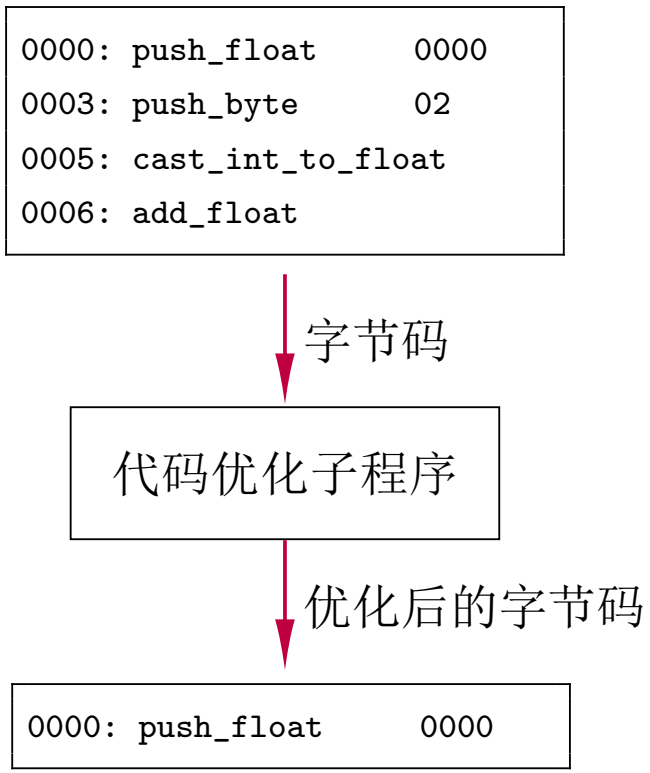


BO 编译器会对生成的字节码进行简单的常量折叠和死码消除，其中包括：



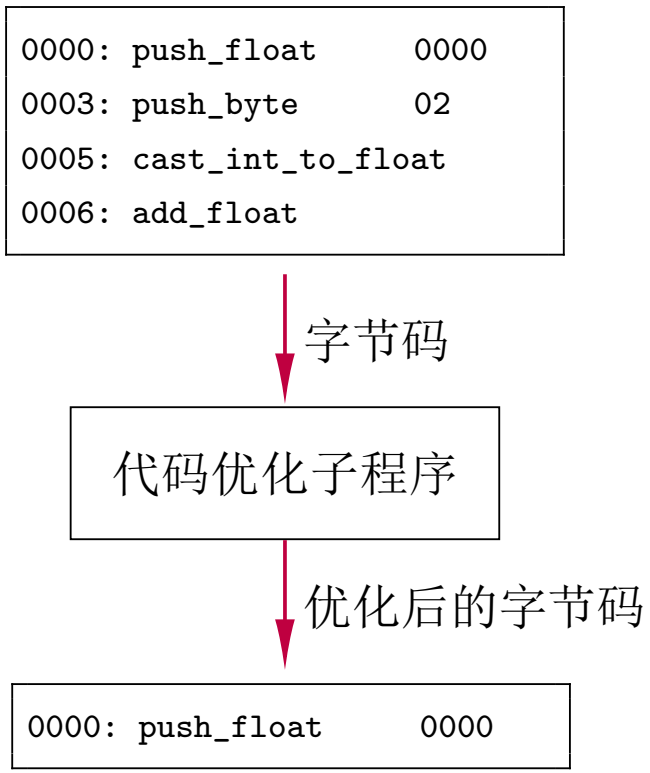
BO 编译器会对生成的字节码进行简单的常量折叠和死码消除，其中包括：

- 计算字面量表达式。



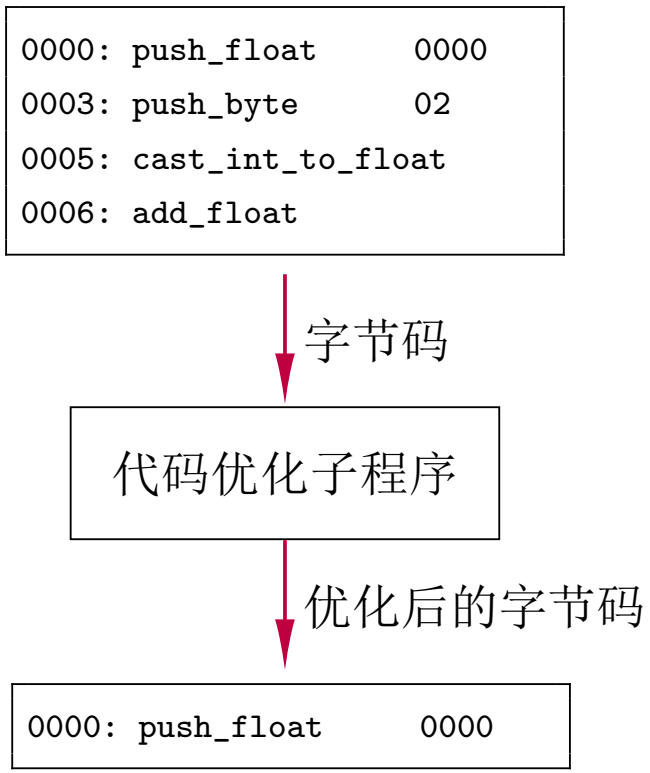
BO 编译器会对生成的字节码进行简单的常量折叠和死码消除，其中包括：

- 计算字面量表达式。
- 消除条件为 false 的 if 和 while 语句，以及循环次数为 0 的 repeat 语句。



BO 编译器会对生成的字节码进行简单的常量折叠和死码消除，其中包括：

- 计算字面量表达式。
- 消除条件为 false 的 if 和 while 语句，以及循环次数为 0 的 repeat 语句。
- 展开条件为 true 的 if 语句，以及循环次数为 1 的 repeat 语句。

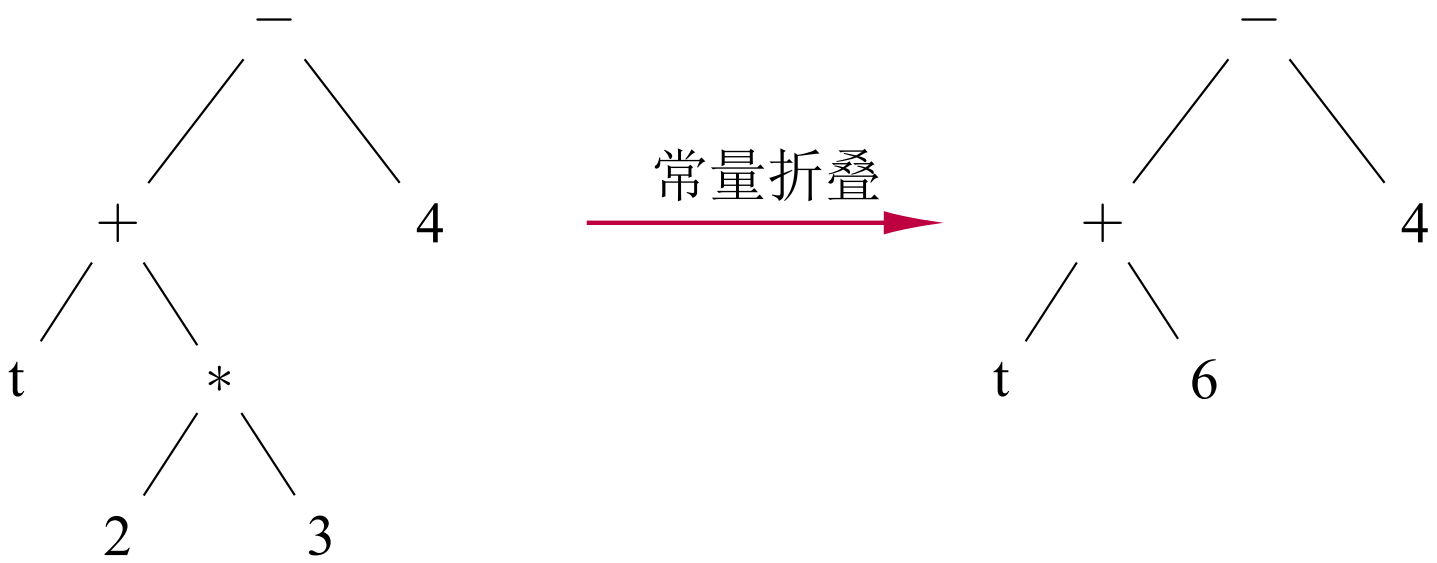


BO 编译器会对生成的字节码进行简单的常量折叠和死码消除，其中包括：

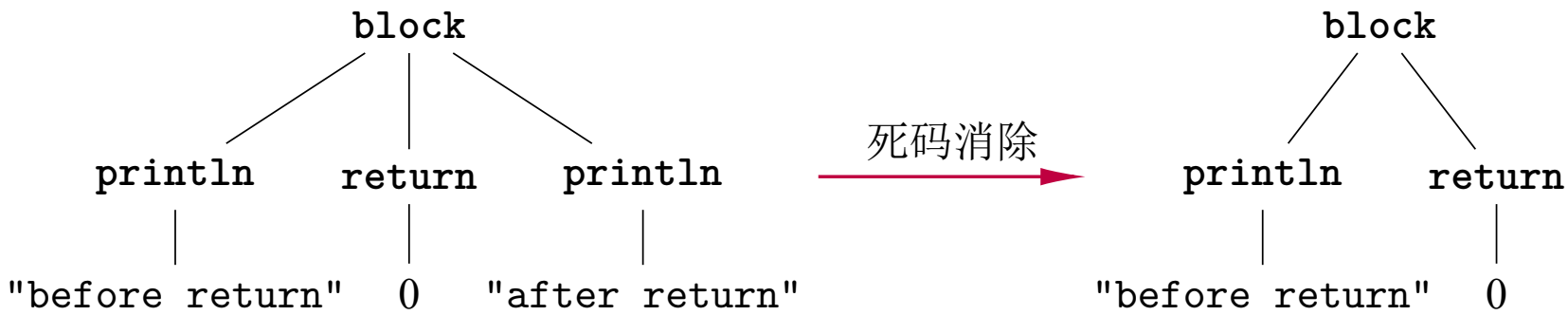
- 计算字面量表达式。
- 消除条件为 false 的 if 和 while 语句，以及循环次数为 0 的 repeat 语句。
- 展开条件为 true 的 if 语句，以及循环次数为 1 的 repeat 语句。
- 对每一个代码块，消除 return、break 和 continue 之后的语句。

简单的常量折叠工作。

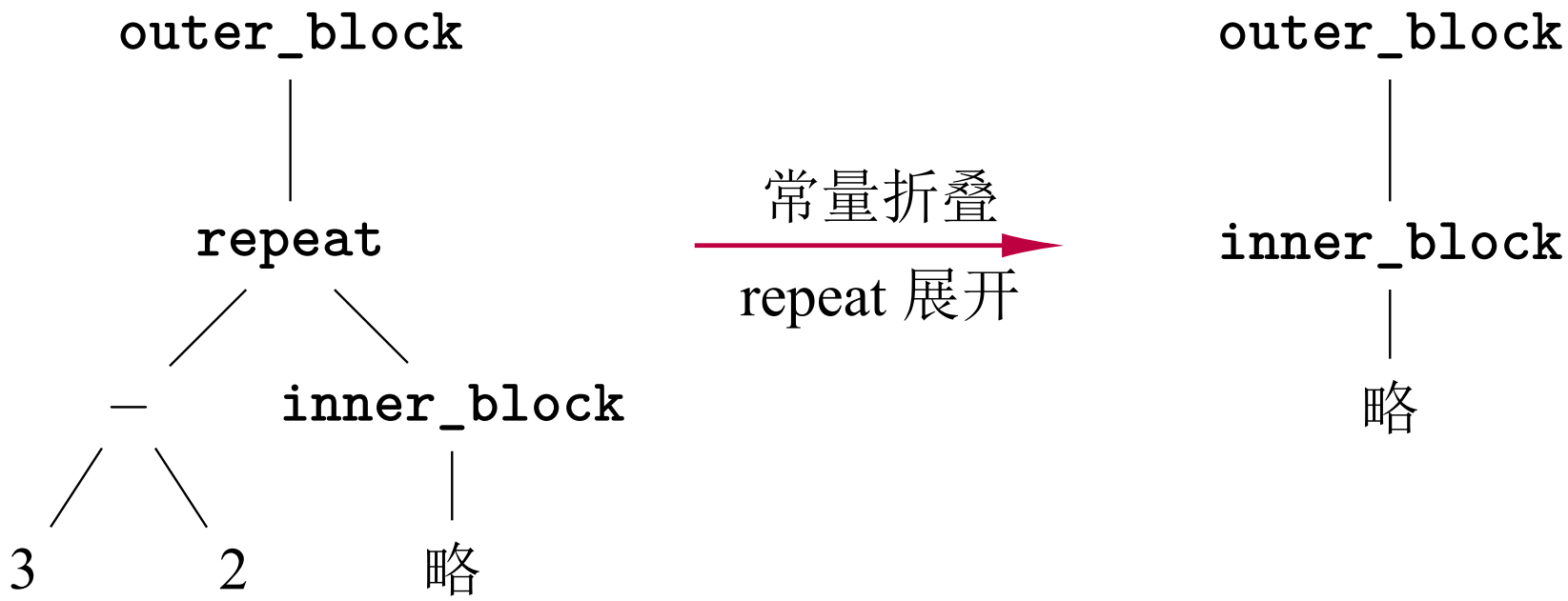
简单的常量折叠工作。



对 return 之后的语句直接消除。



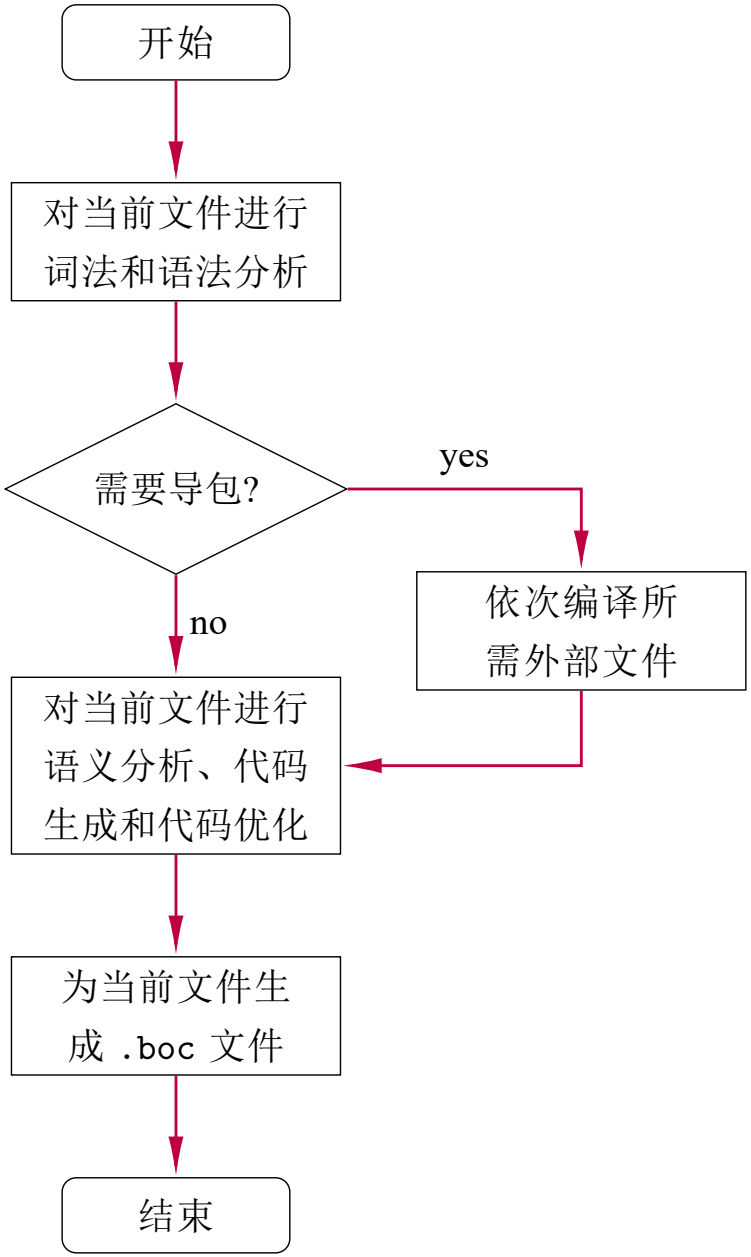
对循环次数为 1 的 repeat 语句进行展开。



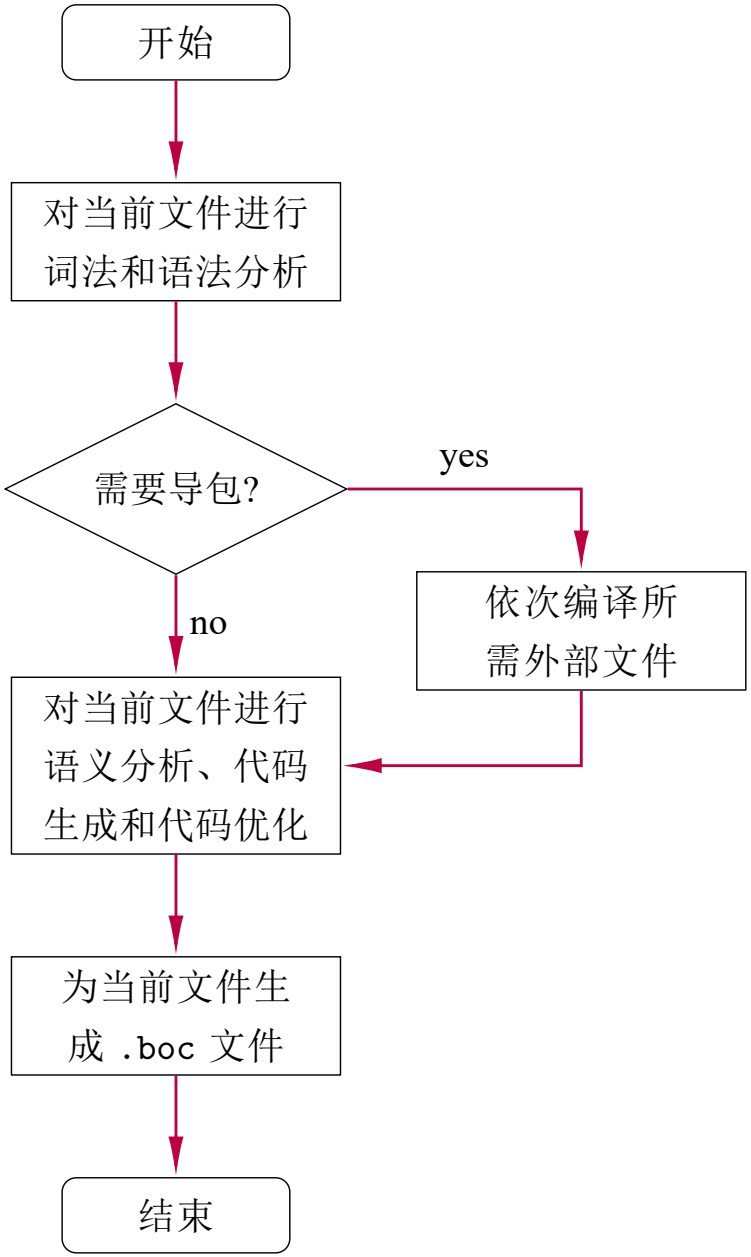
- 在 BO 语言中，一个 `.bo` 文件就是一个包。

- 在 BO 语言中，一个 .bo 文件就是一个包。
- 尽管 boc 命令只能指定一个文件名，但这个文件可能需要导入其他的包。

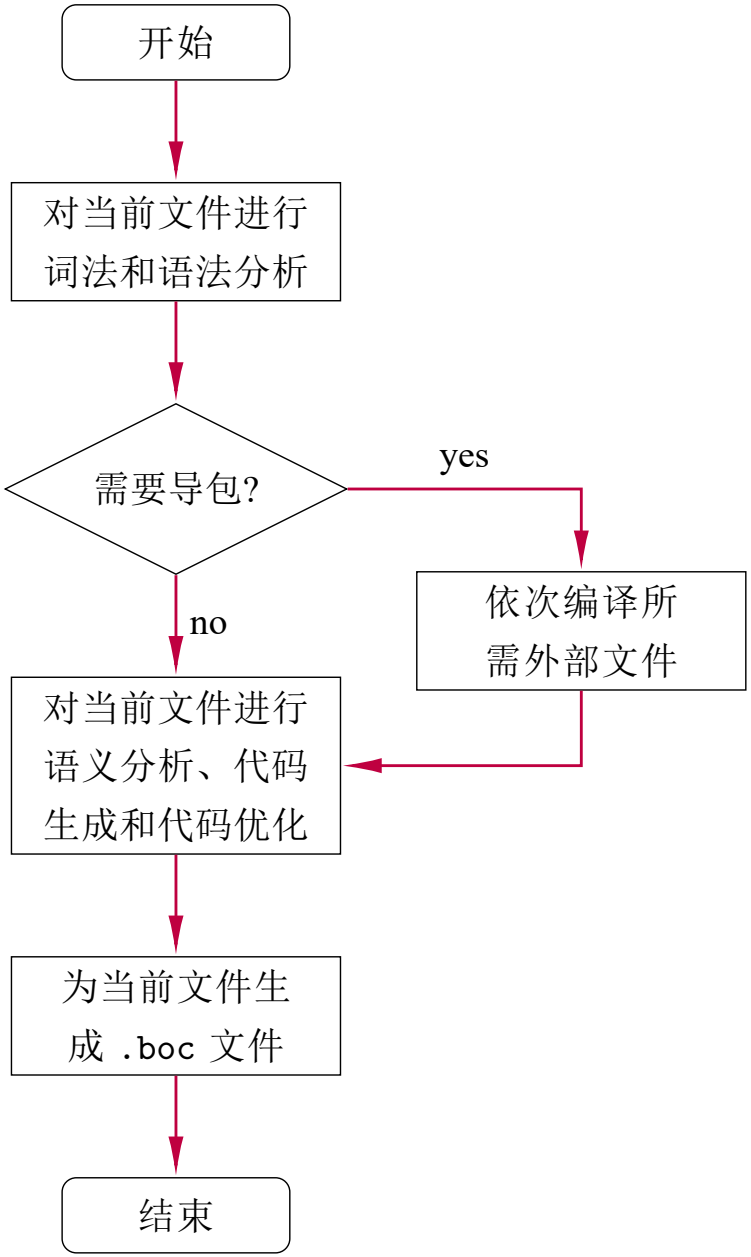
- 在 BO 语言中，一个 `.bo` 文件就是一个包。
- 尽管 `boc` 命令只能指定一个文件名，但这个文件可能需要导入其他的包。
- 因此 BO 编译器也可能需要同时编译多个文件。



- 使用 BO 编译器编译多个文件的完整工作流程如图所示。



- 使用 BO 编译器编译多个文件的完整工作流程如图所示。
- 每编译一个文件，都会对其进行完整的词法和语法分析、导包、语义分析、代码生成和代码优化过程。



- 使用 BO 编译器编译多个文件的完整工作流程如图所示。
- 每编译一个文件，都会对其进行完整的词法和语法分析、导包、语义分析、代码生成和代码优化过程。
- 对于菱形导包的情况，BO 编译器可以保证每个文件仅被编译一次。

- BO 语言的语法是在设计与实现 BO 编译器的实践中逐渐形成和完善的。

- BO 语言的语法是在设计与实现 BO 编译器的实践中逐渐形成和完善的。
- 因此可以认为能被 BO 编译器编译的语言就是 BO 语言。

- BO 语言的语法是在设计与实现 BO 编译器的实践中逐渐形成和完善的。
- 因此可以认为能被 BO 编译器编译的语言就是 BO 语言。
- 也因此 BO 语言的特点实则也是 BO 编译器的特点。

```
(*
define say_hello() -> void {
    println("hello");
    (* println("world"); *)
}

say_hello();
*)
print("ok"); # ok
```

```
(*
define say_hello() -> void {
    println("hello");
    (* println("world"); *)
}

say_hello();
*)
print("ok"); # ok
```

- BO 语言支持嵌套多行注释。

```
(*
define say_hello() -> void {
    println("hello");
    (* println("world"); *)
}

say_hello();
*)
print("ok"); # ok
```

- BO 语言支持嵌套多行注释。
- 当用户想要注释一个带多行注释的函数时，嵌套多行注释的功能就显得十分有效。

```
string ok : null;
integer a : 1;
if [ 1 = a ] { ok : "true"; }
else { ok : "false"; }
println(ok); # true
```

- BO 语言使用 : 作为赋值符号，将 = 留作普通的相等符号。

```
string ok : null;
integer a : 1;
if [ 1 = a ] { ok : "true"; }
else { ok : "false"; }
println(ok); # true
```

```
string ok : null;
integer a : 1;
if [ 1 = a ] { ok : "true"; }
else { ok : "false"; }
println(ok); # true
```

- BO 语言使用 : 作为赋值符号，将 = 留作普通的相等符号。
- 实践证明这种符号表示方法更加符合人的直觉，也更容易被初学者接受。

```
string ok : null;
integer a : 1;
if [ 1 = a ] { ok : "true"; }
else { ok : "false"; }
println(ok); # true
```

- BO 语言使用 : 作为赋值符号，将 = 留作普通的相等符号。
- 实践证明这种符号表示方法更加符合人的直觉，也更容易被初学者接受。
- 然而为了避免使用主流语言的程序员误将 = 符号用于赋值，BO 语言规定布尔表达式必须用 [] 括起来。这一点也可能带来额外的不便。

不同于 C++ 和 Java，BO 语言将表达式

$$a > b > c > d$$

解释为

$$a > b \wedge b > c \wedge c > d$$

这一点更符合数学上的直觉。

BO 语言提供 repeat 语句，可以指定重复次数。

BO 语言提供 repeat 语句，可以指定重复次数。

```
integer sum : 0;
integer i : 0;
repeat (100) {
    i : i + 1;
    sum : sum + i;
}
print("1 + 2 + ... + 100 = ");
println(sum); # 5050
```

BO 语言提供 repeat 语句，可以指定重复次数。

```
integer sum : 0;
integer i : 0;
repeat (100) {
    i : i + 1;
    sum : sum + i;
}
print("1 + 2 + ... + 100 = ");
println(sum); # 5050
```

这也是推荐初学者使用的循环语句，因为初学者使用 while 语句时容易造成死循环。

BO 语言的构造函数使用 constructor 关键字声明。

BO 语言的构造函数使用 constructor 关键字声明。

```
class Obj {
    integer a;
    constructor (integer x) { this.a : x; }
}
Obj obj : new Obj(222);
println(obj.a); # 222
```

BO 语言的构造函数使用 constructor 关键字声明。

```
class Obj {
    integer a;
    constructor (integer x) { this.a : x; }
}
Obj obj : new Obj(222);
println(obj.a); # 222
```

虽然很多主流语言使用类名作为构造函数名，但这确实不是一个好的规则，也不符合程序员的直觉。

BO 编译器的开发平台及开发工具如下：

BO 编译器的开发平台及开发工具如下：

- Ubuntu 20.04.3

BO 编译器的开发平台及开发工具如下：

- Ubuntu 20.04.3
- Scanner 生成器 Flex 2.6.4

BO 编译器的开发平台及开发工具如下：

- Ubuntu 20.04.3
- Scanner 生成器 Flex 2.6.4
- Parser 生成器 Bison 3.5.1

BO 编译器的开发平台及开发工具如下：

- Ubuntu 20.04.3
- Scanner 生成器 Flex 2.6.4
- Parser 生成器 Bison 3.5.1
- 采用 C++ 20 语言开发

BO 编译器的开发平台及开发工具如下：

- Ubuntu 20.04.3
- Scanner 生成器 Flex 2.6.4
- Parser 生成器 Bison 3.5.1
- 采用 C++ 20 语言开发
- C++ 开发环境 CLion 2020.2.1

BO 编译器的开发平台及开发工具如下：

- Ubuntu 20.04.3
- Scanner 生成器 Flex 2.6.4
- Parser 生成器 Bison 3.5.1
- 采用 C++ 20 语言开发
- C++ 开发环境 CLion 2020.2.1

下面将对 Flex 与 Bison 做进一步介绍。

Flex（Fast LEXical analyzer generator）

- 是一个词法分析器的自动生成工具。

Flex（Fast LEXical analyzer generator）

- 是一个词法分析器的自动生成工具。
- 封装了基于词法规则构造有限自动机的过程。

Flex（Fast LEXical analyzer generator）

- 是一个词法分析器的自动生成工具。
- 封装了基于词法规则构造有限自动机的过程。
- 支持使用正则表达式描述语法规则。

Flex（Fast LEXical analyzer generator）

- 是一个词法分析器的自动生成工具。
- 封装了基于词法规则构造有限自动机的过程。
- 支持使用正则表达式描述语法规则。

在 Flex 的输入文件中指定 BO 语言的词法规则，和识别到对应词素时的动作，即可生成能够识别相应规则的词法分析器。

Flex（Fast LEXical analyzer generator）

- 是一个词法分析器的自动生成工具。
- 封装了基于词法规则构造有限自动机的过程。
- 支持使用正则表达式描述语法规则。

在 Flex 的输入文件中指定 BO 语言的词法规则，和识别到对应词素时的动作，即可生成能够识别相应规则的词法分析器。

例：通过指定下列规则和对应的动作即可使词法分析器具有识别 BO 语言标识符并返回 token 的功能。

Flex（Fast LEXical analyzer generator）

- 是一个词法分析器的自动生成工具。
- 封装了基于词法规则构造有限自动机的过程。
- 支持使用正则表达式描述语法规则。

在 Flex 的输入文件中指定 BO 语言的词法规则，和识别到对应词素时的动作，即可生成能够识别相应规则的词法分析器。

例：通过指定下列规则和对应的动作即可使词法分析器具有识别 BO 语言标识符并返回 token 的功能。

```
[A-Za-z_][A-Za-z_0-9]* {
    return make_IDENTIFIER(std::string(YYText()));
}
```

Flex（Fast LEXical analyzer generator）

BO 语言嵌套多行注释的功能可通过下列规则和动作实现。

Flex（Fast LEXical analyzer generator）

BO 语言嵌套多行注释的功能可通过下列规则和动作实现。

```
<INITIAL>"(*"   depthOfComment = 1, BEGIN BO_COMMENT;
<BO_COMMENT>{
\n      increment_line_number();
"(*"    ++depthOfComment;
"*)"    if ( --depthOfComment == 0 ) BEGIN INITIAL;
<<EOF>> throw syntax_error("Unexpected end of line in comment.");
.       ; // 空语句
}
```

Flex（Fast LEXical analyzer generator）

BO 语言嵌套多行注释的功能可通过下列规则和动作实现。

```
<INITIAL>"(*"   depthOfComment = 1, BEGIN BO_COMMENT;
<BO_COMMENT>{
\n      increment_line_number();
"(*"    ++depthOfComment;
"*)"    if ( --depthOfComment == 0 ) BEGIN INITIAL;
<<EOF>> throw syntax_error("Unexpected end of line in comment.");
.       ; // 空语句
}
```

其中变量 depthOfComment 记录嵌套的深度。

Bison

- 是一个语法分析器的自动生成工具。

Bison

- 是一个语法分析器的自动生成工具。
- 封装了基于语法规则构造 LR 分析器的过程。

Bison

- 是一个语法分析器的自动生成工具。
- 封装了基于语法规则构造 LR 分析器的过程。
- 使用类 BNF 范式描述语法规则。

Bison

- 是一个语法分析器的自动生成工具。
- 封装了基于语法规则构造 LR 分析器的过程。
- 使用类 BNF 范式描述语法规则。

在 Bison 的输入文件中指定 BO 语言的语法规则，和识别到对应产生式时的动作，即可生成能够识别相应规则的语法分析器。

Bison

- 是一个语法分析器的自动生成工具。
- 封装了基于语法规则构造 LR 分析器的过程。
- 使用类 BNF 范式描述语法规则。

在 Bison 的输入文件中指定 BO 语言的语法规则，和识别到对应产生式时的动作，即可生成能够识别相应规则的语法分析器。

例：通过指定下列规则和对应的语义动作即可使词法分析器具有识别 BO 语言中 repeat 语句的功能。

Bison

- 是一个语法分析器的自动生成工具。
- 封装了基于语法规则构造 LR 分析器的过程。
- 使用类 BNF 范式描述语法规则。

在 Bison 的输入文件中指定 BO 语言的语法规则，和识别到对应产生式时的动作，即可生成能够识别相应规则的语法分析器。

例：通过指定下列规则和对应的语义动作即可使词法分析器具有识别 BO 语言中 repeat 语句的功能。

```
repeat_statement : REPEAT expression block {
    $$ = create_repeat_statement($expression, $block);
};
```

BO 编译器的不足：

BO 编译器的不足：

- BO 语言缺乏丰富的标准库。

BO 编译器的不足：

- BO 语言缺乏丰富的标准库。
- BO 语言不支持闭包、多线程等当代程序设计语言必备的特性。

BO 编译器的不足：

- BO 语言缺乏丰富的标准库。
- BO 语言不支持闭包、多线程等当代程序设计语言必备的特性。
- BO 编译器加载程序导入的包的过程还有待优化。

BO 编译器的不足：

- BO 语言缺乏丰富的标准库。
- BO 语言不支持闭包、多线程等当代程序设计语言必备的特性。
- BO 编译器加载程序导入的包的过程还有待优化。

因此 BO 语言和 BO 编译器距离真正的实践还要很长的路要走。

BO 语言要想在当代数百种程序设计语言中占有一席之地，就必须：

BO 语言要想在当代数百种程序设计语言中占有一席之地，就必须：

- 最大限度地考虑用户的需求。

BO 语言要想在当代数百种程序设计语言中占有一席之地，就必须：

- 最大限度地考虑用户的需求。
- 尽可能从其它语言中汲取精华、摒弃糟粕。

BO 语言要想在当代数百种程序设计语言中占有一席之地，就必须：

- 最大限度地考虑用户的需求。
- 尽可能从其它语言中汲取精华、摒弃糟粕。
- 扩充标准库以达到使用标准。

Thanks!