第一章：程序的编译

在计算机编程中，程序的编译执行够将我们的代码转变为可以运行在计算机上的实际应用程序。从高级语言编写的源代码程序到二进制可执行文件，这个过程承载着将抽象的代码转换为计算机理解的指令的重要任务，也是将创意和逻辑转化为计算机可理解指令序列的关键步骤。本章将从一个简单的 C 程序出发，深入探讨源代码如何一步步变成计算机可读的应用程序。在第三小节中，我们编写了一个简单语言 MiniC 的编译器，以此帮助大家更好地理解编译器在程序编译执行过程中扮演的角色，以及其基本工作原理。

1.1 程序执行

很多人在学习 C 语言时，都是从一个简单的 `hello.c` 开始：

```
// hello.c
#include<stdio.h>

#define MAX(a,b) a>b?a:b

int main(){
    //print Hello World
   printf("Hello World\n");
    //print MAX(2,3)
    printf("%d\n",MAX(2,3));
   return 0;
}
```

通常我们在集成开发环境（IDE）中运行这样一个 C 程序时，往往只需要点击一个运行（也许是构建+运行）的按钮，程序便会开始执行：

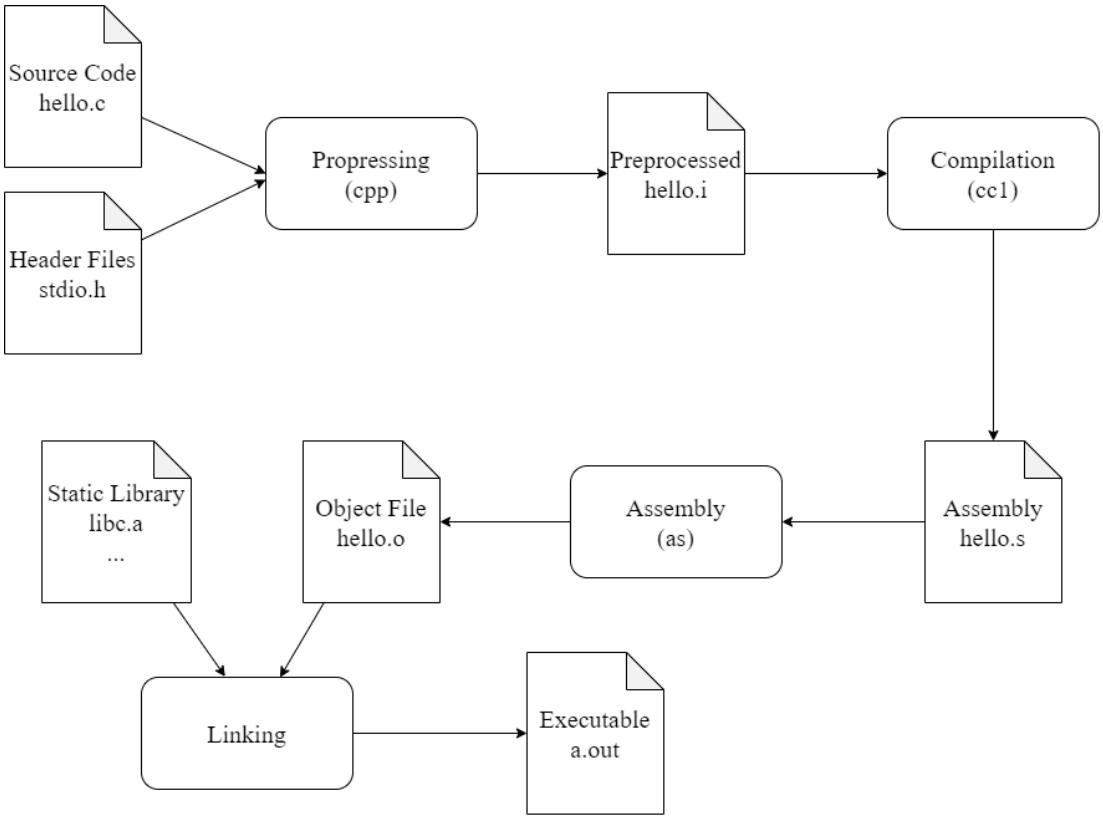
```
Hello World
3
```

但我们都知道计算机无法直接执行高级语言编写的程序，一个 C 程序必须编译成

二进制可执行文件才能在机器上运行。在命令行中，我们可以使用编译器将 C 程序编译成可执行文件再运行，以 gcc 为例：

```
$ gcc hello.c
$ ./a.out
Hello World
3
```

事实上，一个简单的 gcc hello.c 命令背后包含了一系列的步骤。一个 C 程序到可执行文件的过程可大致分为预处理（Prepressing）、编译（Compilation）、汇编（Assembly）和链接（Linking）四个阶段，如下图：



现代 IDE 或者编译器提供的默认配置、编译和链接参数虽然能简化我们日常的程序开发，但在这样的开发中，我们常常会被这些集成工具提供的强大功能所迷惑。面对程序很多莫名奇妙的错误无所适从，对程序运行时的性能瓶颈也束手无策。集成工具对程序运行背后的机制和机理的掩盖使我们无法看清这些问题的本质，而只有深入理解程序执行背后的每一个步骤，我们才能更游刃有余地应对开发中可能遇到的各种问题，同时也能更深刻地体会到计算机背后的魅力。

1.2 编译流程

上一小节提到，程序的执行大致可分为预处理、编译、汇编和链接四个阶段。在这一小节中，我们将更深入地去讨论在每一个阶段中计算机如何将一个 C 程序一步步处理成一个可执行的二进制文件。

1.2.1 预处理

预处理是整个编译过程的第一步。预处理的输入是源代码`*.c`和头文件`*.h`，输出文件是`*.i`。

在预处理过程中，预处理器会分析预处理指令。预处理器将`#include`的头文件内容复制到`*.c`中，如果`*.h`文件中还有`**.h`文件，就递归展开。如果`*.c`中有`#define`宏定义，也会在预处理阶段完成文本替换。此外，预处理阶段还会去除掉源代码中的注释。

我们可以通过`gcc -E`命令在或者直接调用`cpp`预处理器来对`hello.c`进行预处理：

```
$ gcc -E hello.c -o hello.i
# 或者直接调用 cpp
$ cpp hello.c -o hello.i
```

上述命令中：`gcc -E`是让编译器在预处理之后就退出，不进行后续的编译过程。`-o hello.i`则是指定生成的文件名为`hello.i`。

由于头文件的展开，`hello.i`会比`hello.c`多出很多代码，`hello.i`部分内容如下：

```
# 0 "hello.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "hello.c"

# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4

    .......

typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

    ......

# 6 "hello.c"
int main(){

 printf("Hello World\n");

    printf("%d\n",2>3?2:3);
 return 0;
}
```

在上述 hello.i 片段的第 8 行，原本 hello.c 中的 stdio.h 被拓展成其具体路径 /usr/include/stdio.h。第 8 行最前面的# 1 表示接下来 hello.i 中的内容的起始位置对应 stdio.h 的第一行。

同样的，上述片段中第 20 行的# 6 "hello.c"代表接下来的内容起始位置对应 hello.c

文件的第六行，即 main 函数的位置。

对于编译器来说，这些信息都是必要的，因为在预处理过后源文件的代码位置已经发生变化，编译器需要依赖这些信息以追溯代码到原始.c 文件的位置，方便后续调试等操作。

除此之外，可以看到源代码中 prinf 函数内的 MAX(2,3)已经被替换为 2>3?2:3，且源代码中的注释都未能在 hello.i 中保留。

1.2.2 编译

编译过程的输入文件是预处理后的文件*.i，输出文件是汇编文件*.s。

编译过程是把预处理完的文件进行一系列词法分析、语法分析、语义分析以及优化后生成相对应的汇编代码。这个过程往往是整个程序构建的核心部分，也是最复杂的部分之一。

编译的命令如下：

```
$ gcc -S hello.c -o hello.s
```

与预处理过程类似，gcc -S 是让编译器在进行编译后就停止，不进行后续的汇编等过程。

也可以直接调用 cc1 来完成编译，cc1 具体路径根据机器不同略有差别。

```
$ /usr/lib/gcc/x86_64-linux-gnu/11/cc1 hello.i -o hello.s
 main
Analyzing compilation unit
Performing interprocedural optimizations
 <*free_lang_data> {heap 904k} <visibility> {heap 904k}
<build_ssa_passes> {heap 904k} <opt_local_passes> {heap 1036k}
<remove_symbols> {heap 1036k} <targetclone> {heap 1036k} <free-
fnsummary> {heap 1036k}Streaming LTO
 <whole-program> {heap 1036k} <fnsummary> {heap 1036k} <inline>
{heap 1036k} <modref> {heap 1036k} <free-fnsummary> {heap 1036k}
<single-use> {heap 1036k} <comdats> {heap 1036k}Assembling
functions:
 <simdclone> {heap 1036k} main
Time variable                                   usr           sys
wall           GGC
 phase setup                        :   0.00 (  0%)   0.00 (  0%)
0.00 (  0%)  1298k ( 74%)
 phase opt and generate             :   0.01 (100%)   0.00 (  0%)
0.01 (100%)    61k (  4%)
 tree CFG construction              :   0.00 (  0%)   0.00 (  0%)
0.01 (100%)  1416  (  0%)
 integrated RA                      :   0.01 (100%)   0.00 (  0%)
0.00 (  0%)    24k (  1%)
 TOTAL                              :   0.01          0.00
0.01         1748k
```

上述两种方式都可以得到汇编文件`*.s`。汇编语言是二进制指令的文本形式，与二进制指令是一一对应的关系。只要还原成二进制，汇编语言就可以被 CPU 直接执行，所以它是最底层的可读的低级语言。

编译得到的 `hello.s` 汇编文件内容如下：

```
    .file  "hello.i"
    .text
    .section  .rodata
.LC0:
    .string   "Hello World"
.LC1:
    .string   "%d\n"
    .text
    .globl main
    .type  main, @function
main:
.LFB0:
    .cfi_startproc
    pushq  %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq   %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq   .LC0(%rip), %rax
    movq   %rax, %rdi
    call   puts@PLT
    movl   $3, %esi
    leaq   .LC1(%rip), %rax
    movq   %rax, %rdi
    movl   $0, %eax
    call   printf@PLT
    movl   $0, %eax
    popq   %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size  main, .-main
    .ident "GCC: (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0"
    .section  .note.GNU-stack,"",@progbits
```

汇编程序中以.开头的名称并不是指令的助记符，不会被翻译成机器指令，而是给汇编器一些特殊指示，称为汇编指示（Assembler Directive）或伪操作（Pseudo-operation）。以上述的 hello.s 为例：

符号	含义
.file	指示源文件名
.text	代表接下来的内容为代码段
.section	指示把代码分成若干段
.rodata	代表接下来的内容为只读数据段
.LC0	助记符号，类似 C 语言中的变量名
.string	对应符号地址存储值的类型
"Hello World"	对应符号地址存储的内容
.globl main	声明一个全局符号 main
.type main, @function	指定 main 为一个函数
.cfi_xxx	基于 CFI 规范生成的指令，用于生成有效的调试信息

main 函数中包含了许多 X84_64 架构的汇编指令，这些指令在汇编阶段会被翻译成对应的可在机器上运行的二进制机器指令，linux 下的 x86-64 指令格式采用 AT&T 格式，也即源操作数在左边，目的操作数在右边。 绝大多数 x86-64 指令采用后缀字母标示操作数的大小，例如 b 表示 8 位，w 表示 16 位，l 表示 32 位，q 表示 64 位。

汇编指令	含义
pushq %rbp	将寄存器%rbq 压到栈上，%rbq 通常存储执行函数的基址，这里压栈保存的还是 main 父函数的基址。
movq %rsp, %rbp	将%rsp 拷贝至%rbp 中。即令%rbp 存储 main 函数栈空间的基址。
movl $3, %esi	将 32 位立即数 3 加载至%esi 中，%esi 是%rsi 的低 32 位
leaq .LC0(%rip), %rax	将.LC0 标签所代表的内存地址加载到%rax 寄存器中。%rip 通常存储当前指令的地址，.LC0(%rip)是一种间址寻址的方法，通过当前指令地址+偏移量得到.LC0 的地址。
call put@PLT	调用 put 函数，PLT（Procedure Linkage Table）是动态链接时创建的表，用于重定位函数。

popq %rbp	弹出栈顶值到%rbq 中，现在%rbp 存储 main 函数的父函数的基址。
ret	返回父函数。

1.2.3 汇编

汇编过程的输入是汇编文件*.s，输出是二进制目标文件*.o。

在汇编过程中，汇编器会将汇编代码转变成机器可以执行的指令，每一个汇编语句几乎都对应一条机器指令。所以汇编器的汇编过程相对于编译器来讲比较简单，它没有复杂的语法，也没有语义，也不需要做指令优化，只需要根据汇编指令和机器指令的对照表一一翻译就可以了。

汇编过程可以用如下的命令完成：

```
$ gcc -c hello.c -o hello.o
# 或者直接调用汇编器 as
$ as hello.s -o hello.o
```

汇编生成的*.o 目标文件为二进制文件，其内容可以用二进制查看器进行查看。以 hello.o 为例，其开头部分内容以 16 进制展示为：

```
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00
01 00 3E 00 01 00 00 00 00 00 00 00 00 00 00 00
......
```

其中，前四个字节 7F 、45、4C、46 分别对应 ASCII 码的 Del(删除)、字母 E、字母 L、字母 F。这四个字节被称为 ELF 文件的魔数，操作系统在加载可执行文件时会确认魔数是否正确，如果不正确则拒绝加载。

第五个字节标识 ELF 文件是 32 位（01）还是 64 位（02）的。

第六个字节标识该 ELF 文件字节序是小端（01）还是大端（02）的。

第七个字节指示 ELF 文件的版本号，一般是 01。

后九个字节 ELF 标准未做定义。一般为 00。

此外，也可以使用 objdump 反汇编工具将查看 hello.o 的内容：

```
$ /hello$ objdump -d hello.o

hello.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f
<main+0xf>
   f:   48 89 c7                mov    %rax,%rdi
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   be 03 00 00 00          mov    $0x3,%esi
  1c:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 23
<main+0x23>
  23:   48 89 c7                mov    %rax,%rdi
  26:   b8 00 00 00 00          mov    $0x0,%eax
  2b:   e8 00 00 00 00          call   30 <main+0x30>
  30:   b8 00 00 00 00          mov    $0x0,%eax
  35:   5d                      pop    %rbp
  36:   c3                      ret
```

objdump -d 表示对代码段进行反汇编。不难看出，hello.o 的内容大部分都是对 hello.s 中汇编指令的一一翻译。

汇编生成的目标文件*.o 虽然是包含机器指令的二进制文件，但*.o 并不是最终的可执行二进制文件，而仍是一种中间文件，目标文件仍然需要经过链接才能变成可执行文件。

1.2.4 链接

链接过程的输入是二进制目标文件*.o，以及相关二进制库等，输出为可执行文件。

前面的过程只是将我们自己写的代码变成了二进制形式，它还需要和系统组件（比如标准库、动态链接库等）结合起来，这些组件都是程序运行所必须的。链接其实就是一个“打包”的过程，它将所有二进制形式的目标文件*.o 和系统组件组合成一个可执行文件。

链接过程可用如下命令完成：

```
$ gcc hello.o
```

链接后的可执行文件若不加-o 进行重定向，gcc 会默认输出文件名为 a.out。此时执行 a.out，程序便可真正在机器上运行：

```
$ ./a.out
Hello World
3
```

但其实在上面的 gcc hello.o 命令中，gcc 同样在背后做了许多事情。包括自动寻找需要的系统组件并链接等等。

链接过程可以通过调用 ld 来进行：

```
$ ld hello.o
ld: warning: cannot find entry symbol _start; defaulting to
0000000000401000
ld: hello.o: in function `main':
hello.c:(.text+0x13): undefined reference to `puts'
ld: hello.c:(.text+0x2c): undefined reference to `printf'
```

若直接进行链接，ld 会提示找不到_start 符号，_start 是程序运行的默认入口。通常情况下，_start 会存在于某一个系统标准库中，程序也会在这个位置开始运行。但此处我们没有加任何其他系统标准库，所以 ld 自然找不到_start。

我们可以使用-e 参数来指定程序入口，比如：

```
$ ld -e main  hello.o
ld: hello.o: in function `main':
hello.c:(.text+0x13): undefined reference to `puts'
ld: hello.c:(.text+0x2c): undefined reference to `printf'
```

这时 ld 不再提示找不到_start，因为我们已经手动把入口改为了 main。但 ld 提示找不到 put 和 printf。put 和 printf 都是系统调用，同样在某个标准系统库中。put 是比 printf 更简单的调用，对于简单的 printf 函数，如源代码中的 printf("Hello World\n");只是简单地打印字符串，编译器就会将其优化为 put。

现在我们在 hello.c 去掉所有的函数调用，只保留最简单的部分。理论上这能生成最简单的可执行文件

```
int main(){
   return 0;
}
```

生成目标二进制文件后，用 ld 指定运行入口进行链接：

```
$ gcc -c hello.c
$ ld -e main  hello.o
```

此时 ld 不再报错，并且能成功生成可执行文件 a.out。运行：

```
$./a.out
Segmentation fault (core dumped)
```

程序运行发生错误。这是因为在 main 函数中，return 0 会将 0 返回给调用者。但我们将程序入口指定为 main()函数，自然不存在调用者，程序也无法找到返回地址，故发生错误。事实上，一个正常的程序编译为可执行文件后，其最外层必须进行系统调用 exit，程序才能正常退出，否则这个程序执行就会发生 Segementation Fault。

1.3 MiniC 的编译器

本小节我们来构造一个 MiniC 编译器，以此来加深对上一小节编译流程的认识。MiniC 的文法如下：

```
e ::= n | x | e+e | e-e | e*e | e/e
s ::= x=e | print(e) | s;s
```

接下来，我们详细解释这两个产生式：

（1）e 是一个算术表达式，可以是：

- n: 一个数字，例如 5 或 42。
- x: 一个变量名，例如 a 或 varName。
- e+e: 两个算术表达式的和，例如 a+5 或 3+4。

.....

（2）s 是一个语句，可以是：

- x=e: 一个赋值语句，其中 x 是一个变量名，e 是一个算术表达式。例如 a=5 将数字 5 赋给变量 a。
- print(e): 打印语句，会输出算术表达式 e 的值。例如 print(a) 将输出变量 a 的值。
- s;s: 两个语句的序列。它们会依次执行。例如 a=5; print(a) 将数字 5 赋给变量 a 然后输出其值。

这个文法描述的语言能够处理基本的算术运算、变量赋值和打印操作。它为编译器提供了解析该语言句子的基础规则。

1.3.1 词法分析器

首先我们来看词法分析部分。构造词法分析器一种机械性的工作，很容易由计算机来自动实现，因此一种有意义的做法是，用一个词法分析器的自动生成器来将正则表达式转换为词法分析器。在 MiniC 中，我们选择 flex 作为用于生成词法分析器（或

称为扫描器）的工具，flex 的输入是一系列的正则表达式和要在匹配这些正则表达式时执行的 C 代码。

```
DIGIT      [0-9]
ID         [a-zA-Z_][a-zA-Z_0-9]*

%%
<<EOF>>                {return 0;}
"+"                    {return yytext[0];}
"-"                    {return yytext[0];}
"*"                    {return yytext[0];}
"/"                    {return yytext[0];}
"="                    {return yytext[0];}
"("                    {return yytext[0];}
")"                    {return yytext[0];}
";"                    {return yytext[0];}
[ \t\r\n]+             {}
"print"                { return PRINT;}
{ID}                   {yylval.var = strdup(yytext); return VAR; }
{DIGIT}+               {yylval.num = atoi(yytext); return NUM; }
.                      {printf("lex err at line %d: unexpected
character '%s'\n", yylineno, yytext); exit(1); }

%%
```

该实现定义了一个名为 scanner 的词法分析器，采用 flex 工具进行构造。flex 要求其输入文件的名字通常以`.l`或`.lex`为后缀。正则表达式定义部分详述了各类词法单元的匹配规则，如基本操作符（例如“+”代表加号）、分隔符（如“;”代表分号）等等。某些规则关联了具体的动作。例如，对于每一个识别到的操作符，动作部分简单地返回了相应的字符；而对于 ID，动作部分将词法单元的文本值存储到 yylval.var 中，并返回一个代表变量的标记 VAR。正则表达式[\t\r\n]+匹配输入中的空白字符，如空格、制表符、换行等。与之关联的动作是空的，这意味着词法分析器将忽略这些空白字符，

并继续识别下一个词法单元。在 flex 的环境中，<<EOF>>匹配输入结束，通常与一个特定的返回值关联，表示词法分析的终止。最后的规则`.`用于捕获不符合任何前面规则的字符，当匹配时，它会产生一个错误消息，指出在哪一行发现了意外的字符，并终止程序执行。

总体而言，这是一个简单词法分析器的实现，能够识别基本的操作符、标识符、数字以及一些关键词，并为它们分配相应的标记或执行特定动作。

1.3.2 语法分析器

在语法分析实现部分，我们选择 bison 作为生成语法分析器的工具，bison 通常与 flex 结合使用。bison 的输入是一个上下文无关文法和当规则被识别时要执行的 C 代码。

```
%union{
    int num;
    char *var;
    Stm_t stm;
    Exp_t exp;
}
%token<var> VAR
%token<num> NUM
%token PRINT
%type<stm> program
%type<stm> stm
%type<exp> exp
%left '|' ';'
%left '+' '-'
%left '*' '/'
%%
program     : stm                { root = $1; $$ = $1; return
0; }
            ;
stm         : VAR '=' exp         { $$ = Stm_Assign_new($1,
$3); }
          | PRINT '(' exp ')'     { $$ = Stm_Print_new($3); }
          | stm ';' stm           { $$ = Stm_Seq_new($1, $3); }
          ;
exp          : NUM                { $$ = Exp_Num_new($1); }
          | VAR                   { $$ = Exp_Var_new($1); }
          | exp '+' exp           { $$ = Exp_Add_new($1, $3); }
          | exp '-' exp           { $$ = Exp_Sub_new($1, $3); }
          | exp '*' exp           { $$ = Exp_Times_new($1, $3); }
          | exp '/' exp           { $$ = Exp_Div_new($1, $3); }
          | '(' exp ')'           { $$ = $2; }
          ;
%%
```

上述程序主要用于定义如何从源代码中解析出不同的语法结构，并如何为这些结构构建相应的抽象语法树（AST）。

1. Union 定义 (%union):

这部分定义了将在后续的产生式中使用的联合体类型。它可以包含多种数据类型，这里包括了整数、字符指针以及两种自定义的数据结构 (Stm_t 和 Exp_t，代表语句和表达式)。

2. Token 和类型定义:

这部分定义了词法分析器（如 flex 生成的）将返回的词法单元类型和它们的关联数据类型。例如，VAR 是一个带有 char * 类型数据的 token，而 NUM 带有 int 类型的数据。

3. 优先级和结合性:

通过%left 声明，该部分定义了算术运算符的优先级和结合性。这确保了像 3 + 4 * 2 这样的表达式会被解析为 3 + (4 * 2)，而不是(3 + 4) * 2。

4. 文法规则:

这是该程序的核心部分，定义了如何从源代码中解析出不同的语法结构。

- program: 表示整个程序的入口点，即主语句。
- stm: 代表一个语句，可以是赋值语句、打印语句或两个语句的序列。
- exp: 代表一个算术表达式，可以是数字、变量或两个表达式的算术操作。

5. 动作:

在产生式的{}中定义的代码片段是当规则被成功匹配时执行的动作。例如，当匹配到一个赋值语句 VAR = exp 时，它将创建一个新的赋值语句的 AST 节点。

总之，这段程序定义了一个简单的算术和赋值语言的语法规则，并为匹配的语法结构创建了相应的 AST。

1.3.3 抽象语法树

定义抽象语法树节点的数据结构来表示不同的语法结构，以及构建该节点的函数，然后在 Bison 的语法规则中使用这些结构。

```
// statement
enum Stm_Type_t{
    STM_ASSIGN,
    STM_PRINT,
    CMD_SEQ
};
typedef struct Stm_t *Stm_t;
struct Stm_t{
    enum Stm_Type_t type;
};
void Stm_print(Stm_t stm);

typedef struct Exp_t *Exp_t;
typedef struct Stm_Assign *Stm_Assign;
struct Stm_Assign{
    enum Stm_Type_t type;
    char *x;
    Exp_t exp;
};
Stm_t Stm_Assign_new(char *x, Exp_t exp);
......
```

值得注意的是，结构体 Stm_t 可视为面向对象语言中的基类，仅存放 type 这一个枚举类型，当我们创建新的语法树节点后，将其强制类型转换为 Stm_t 类型插入抽象语法树中，当处理具体的语法树节点时，根据第一个字段 type 转换为具体的语法树节点，从而通过 C 语言实现了面向对象中的“多态”。

1.3.4 代码生成

MiniC 编译器采用递归下降法实现代码生成。

```
void compile(Stm_t prog, int fd);
```

compile 函数接收抽象语法树的根节点，对抽象语法树进行递归下降分析，针对不同的语法树节点类型，发射不同的汇编指令。最后，将生成的代码保存到指定路径

下的.s 文件中。

```
void compile_stm(Stm_t stm){
    switch (stm->type) {
        case STM_ASSIGN: {
            Stm_Assign s = (Stm_Assign) stm;
            add_var(s->x);
            compile_exp(s->exp);
            sprintf(instr, "\tmovq %%rax, %s\n", s->x);
            emit(instr);
            break;
        }
        ......
    }
}
```

以赋值操作为例，当前节点的类型为 STM_ASSIGN 时，先将该节点强制类型转换为 Stm_Assign 类型，以便获取其中的变量名和表达式节点，将变量名添加到符号表中，然后递归解析表达式节点，根据节点类型发射相应的汇编指令并将表达式的计算结果存放在 rax 寄存器中，最后，发射汇编指令 movq %rax，变量名，将表达式计算结果存放在变量中。

1.3.5 汇编与链接

由于汇编器与链接器的实现细节比较繁琐，所以 MiniC 编译器直接使用 GCC 的工具链进行汇编与链接。

```
    // assemble
    sprintf(cmd, "as %s -o %s", asm_file_name, obj_file_name);
    sprintf(cmd, "cp %s ./a.s", obj_file_name);
    if (system(cmd) != 0) {
        fprintf(stderr, "Assembly failed.\n");
        return 1;
    }

    // link
    sprintf(cmd, "gcc %s -o output", obj_file_name);
    if (system(cmd) != 0) {
        fprintf(stderr, "Linking failed.\n");
        return 1;
    }
    printf("Compilation and linking successful. Executable is
named 'output'.\n");
```

1.3.6 MiniC 编译选项

MiniC 编译器支持的编译选项有-c -s -o -v --help。

```
$ ./mcc input.txt              默认  汇编  生成 a.s
$ ./mcc -c input.txt           生成 a.s  a.o
$ ./mcc -s input.txt           汇编  生成 a.s
$ ./mcc -v -c input.txt        生成 a.s  a.o    同时输出编译过程
$ ./mcc -v -s input.txt        汇编  生成 a.s    同时输出编译过程
$ ./mcc input.txt -o input     指定输出的可执行文件名称
$ ./mcc --help                 输出全部编译选项信息
```

1.4 本章小结

在这一章中，我们首先讨论了从程序源代码到最终可执行文件的四个步骤：预处理、编译、汇编以及链接，分析了它们的作用以及相互之间的联系。

在第三小节中，我们还更深入探讨了上述步骤中最复杂也是最关键的部分，即编译步骤。通过编写一个简单的 MiniC 语言的编译器，我们介绍了最简单的编译器将高级语言编译为汇编代码的最基本步骤：词法分析、语法分析、抽象语法树构建和目标代码生成。

1.5 深入阅读

《Compilers: Principles, Techniques, and Tools》

《程序员的自我修养:链接、装载与库》