در این بخش باید یک Abstract Syntax Tree بسازیم. پارسر توکن هایی را از لکسر میگیرد و با آن‌ها یک parse tree میسازد.
ریشه درخت از تایپ program است و برگهایش non-terminal اند.
درختمان را بکمک context free grammar میسازیم.

در بخش bison declarations ترمینال ها و غیر ترمینال های گرامر را تعریف میکنیم.
که union ها تایپ های مورد نظر یعنی non terminal ها اند. پس از این union ها برای تایپ non terminal ها استفاده میکنیم بصورت:

```
%type <expressions> expression
```

توکن هایی که توسط لکسر ساخته می‌شوند در برنامه ما بصورت terminal دیده میشوند.
برنامه ما تعدادی توکن ورودی میگیرد و ما میخواهیم با Bottom up parse آنرا تبدیل به سیمبول program کنیم که starting symbol است.

باید تعریف اولویت انجام دهیم یعنی precedence declaration
مثلاً برای 3+2+1 اول 3+2 را انجام میدهیم و سپس نتیجه را با 1 جمع کنیم پس درخت از سمت چپ به راست ساخته میشود
اولویت ها به اینصورت است که هرکدام پایینتر تعریف شود پس اولویت بیشتری دارد و در درختمان به برگ نزدیک‌تر است و عملگر هایی که اولویت کمتر دارند زودتر تعریغ شده و به ریشه نزدیک ترند.

```
right FLAG%
right ASSIGN%
right NOT%
nonassoc '<' '=' LE%
'-' '+' left%
'/' '*' left%
left ISVOID%
'~' left%
'@' left%
'.' left%
```

برای IN که در let آمده در ادامه تعریف میکنیم.

سپس باید قوانین گرامرمان را بنویسیم
در نوشتن production ها $n نشان دهنده مقدار n امین مولفه (non terminal) است و $$ نشان دهنده semantic value
مثلا:

```
exp:
        ...some right hand sides…
        | exp ‘+’ exp { $$ = $1 + $3; }
        …
```

پس وقتی داریم exp ::= expr + expr برای نشان دادن این production از سازنده ای بنام plus استفاده میکنیم
ساختارش بصورت زیر است:

```
plus (e1,e2:Expression): Expression
```

این ساختار ها در cool-tree.aps معرفی شده اند.

Let: در let ابهام داریم. این ابهام با دادن کمترین اولویت به IN برطرف میشود. مشکل در let این است که کامپایلر باید تا جایی که let ادامه دارد، parse را ادامه دهد و چون IN در ساختار عبارت let کمترین اولویت را دارد پس همیشه تا آنجا که برایش ممکن است عبارت parse شده را به let تخصیص میدهد. (پس بکمک دادن کمترین اولویت به IN کاری کردیم که let تا جای ممکن تا سمت راست ادامه داشته باشد)

error: وقتی توکن ورودی error باشد، نمیخواهیم عملیات parse متوقف شود پس در قوانین گرامرمان error را هم تعریف میکنیم.
به اینصورت که اگر در تعریف یک کلاس error داریم ولی کلاس بدرستی terminate شده و کلاس بعدی بدرستی تعریف شده، پارسر باید بتواند از class definition بعدی شروع کند و parse متوقف نشود.
همچنین پارسر باید بتواند از ارور هایی که در feature ها، let exp و exp block ها رد شود مثلاً در یک let binding اگر اروری وجود داشت، به متغیر بعدی برود.