**Lego Compiler **

در فاز اول پروژه کامپایلر ، طراحی تحلیل گر لغوی را داریم . در ابتدا کلاس به اسم Lexical_Analysis را تعریف کردیم (کلاس شناسایی token ها).

setPath

ابتدا در این متد کل فایل ورودی را در string ای به اسم code ذخیره میکنیم . آرگومان این متد مسیر فایل ورودی میباشد .

فایل کد ورودی باید کارکتر به کارکتر خوانده شود تا باعث شناسایی token ها شود برای این منظور متغیری برای نگهداری مکان کارکتر فعلی و متغیری برای نگه داری state فعلی نیاز داریم . در کد Logo Compiler متغیرهایی به اسم currentPoint و currentState تعریف کردیم و مقدار currentPoint را -1 میگزاریم تا با شناسایی اولین کارکتر مقدار 0 بگیرید .

DFA

با استفاده از الگوریتم DFA از درس نظریه زبان ها ، الگوریتمی برای شناسایی انواع token ها تعریف کردیم .

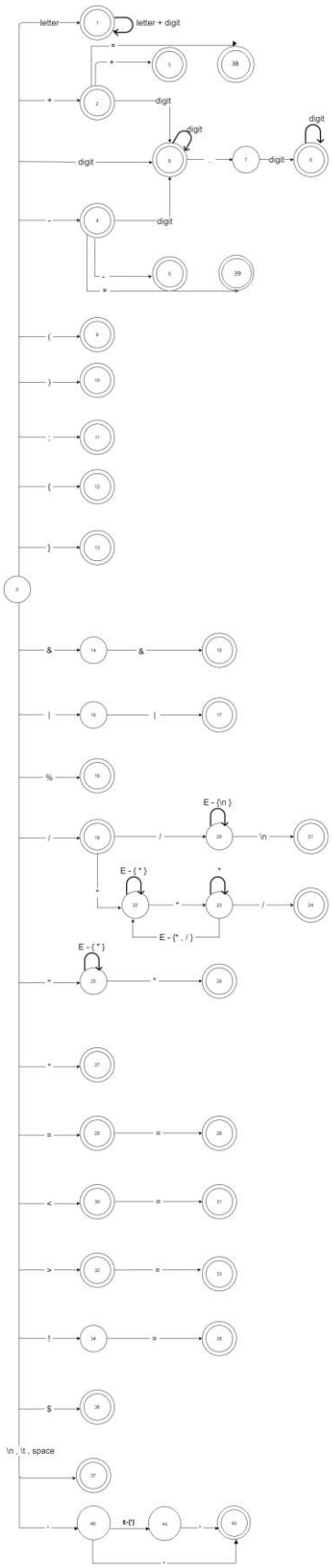
همان طور که در شکل صفحه بعد میبینید .

برای مثال در STATE 0 هستیم ، در صورت مشاهده حروف به state =1 میرویم و در صورت مشاهده دوباره حروف یا عدد در همین state باقی میمانیم که یک state نهایی است .

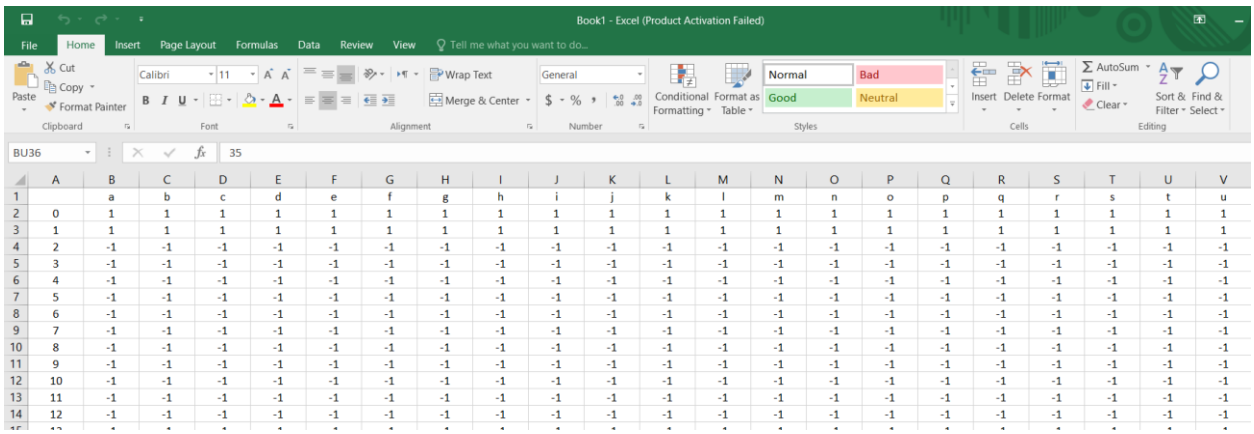
در صورت مشاهده هر کارکتری به غیر از حروف الفبا و ارقام token شناسایی شده از نوع id میباشد .(در این کد تفاوتی برای keyword و id قائل نشدیم).

الفبای مجاز برای Lego Compiler در آرایه هایی با نام های letters , digits , keywords , symbols, whitespaces

تعریف شده است و در متد های isInletters , isIndigits , isInkeywords , isInsymbols , isInwhitespaces چک میکنیم که کارکتر ورودی در الفبای زبان ما وجود دارد یا خیر .


letter
letter + digit
=
+
digit
digit
digit
digit
digit
-
digit
(
)
;
{
}
&
&
|
|
%
/
E - {\n }
\n
E - { * }
*
E - {* , / }
"
E - { " }
"
*
=
=
<
=
>
=
!
=
$
\n , \t , space
-
E-{'}
'

برای سرعت و راحتی کار از جدول انتقال استفاده کردیم تا مشخص شود در هر state با خواندن هر کارکتر به چه state می رویم . از excel استفاده کردیم همه ستون ها نشان از یک کارکتر هستند که در الفبا زبان تعریف شدند . هر سطر نشان از شماره state می باشد .



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
2	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
14	12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

برای مثال در شکل بالا اگر در state=0 باشیم با خواندن تمامی حروف الفبا به state=1 منتقل خواهیم شد . این جدول را برای تمامی حالات ها مورد بررسی قرار دادیم .

این داده هارا در آرایه دو بعدی به اسم Ttable تعریف کردیم و در متدی به اسم nextState مورد استفاده قرار دادیم .(مقدار 1- برای مسیر های تعریف نشده قرار داده شده است)

nextState

این تابع داری دو آرگومان ورودی شماره state و کارکتر میباشد که با استفاده از این دو میخواهیم state بعدی را tTable به دست آوریم .

شماره state ها مشخص بود اما شماره index کارکتر که در ستون های excel بود باید به دست می آمد پس از حلقه های for استفاده شد . طول تمامی کارکتر ها را نداشتیم از حلقه های for جداگانه استفاده کردیم به شرطی که اگر در حلقه اول حرف ها را مورد بررسی قرار میگرفت در حلقه دوم که به بررسی عدد ها میپرداخت index به دست آمده با طول letters ها جمع میشد .

Nextchar

این متد با استفاده از current point کارکتر بعدی را به ما برمیگرداند.(در صورت رسیدن به پایان string code کارکتر $ را برمیگرداند).

whachNextChar

این متد بدون تغییر دادن currentPoint کارکتر بعدی را به ما برمیگرداند .

nextToken

در این متد token ها را شناسای و برمیگردانیم . در این متد با استفاده از دستور switch case ، state ها را مورد بررسی قرار دادیم .

currentState شماره case مورد نظر ماست . nChar کارکتر بعدی است . با دادن این دو به متد nextState ، state بعدی به دست می آید و در nState ذخیره میشود .

برای مثال

Case 0 :

کارکتر جدید رابا استفاده از nextchar در ch نگهداری میکنیم و ch هر بار به string token_name اضافه میشود .

currentState = nextState(currentState, ch); این دستور که با استفاده از جدول تبدیل بسیار ساده شد شماره state فعلی و کارکتر را به متد nextState میفرستد تا state بعدی به دست آید و مورد بررسی مجدد قرار گیرد . در state=0 اولین کارکتر مسیر مارا مشخص خواهد کرد .(اولین کارکتر از هر token)

state=0 نهایی نیست پس در صورت -1 بودن state بعدی token شناسایی نمیشود و پیغام خطا شناسایی میشود .

case=1 :

شناسایی کلمات کلیدی و id ها در این state انجام میشود . state=1 خود یک state نهایی تعریف شده است پس میتواند مرحله شناسایی token باشد .

اگر state بعدی -1 شود به این معناست که کلمه به پایان رسیده است پس باید نوع این token را تشخیص دهیم . در ابتدا با تمامی کلمات کلیدی الفبای خود مقایسه میکنیم در صورت برابر بودن ، token شناسایی و از نوع keyword ثبت میشود . اگر با هیچ یک از کلمات کلیدی برابر نبود از نوع id خواهد بود .

Case 2,3 , 38:

کارکتر + خوانده شده است و ما به state=2 منتقل شده ایم . طبق DFA این state خود نهایی است و میتواند به تنهایی علامت جمع باشد ، اما اگر کارکتر بعدی عدد یا digit باشد به state =6 میرویم و این کارکتر بخشی از عدد خواهد بود که نشانه مثبت بودن است . حالت سوم اگر کارکتر بعدی نیز + باشد به state=3 میرویم و عملگر منطقی plusplus نامیده خواهد شد (case 3). حالت چهارم اگر کارکتر بعدی = باشد به state 38 میرویم و عملگر منطقی plusEqual شناسایی میشود

State های 4 و 5 ,39نیز مانند 2,3 ,38 پیاده سازی شده اند .

case 6:

مطمینا با یک عدد مواجه هستیم . اگر همچنان کارکتر بعدی digit باشد در همین state باقی می مانیم . اگر به کارکتری به غیر از . رسیدیم شناسایی token به پایان میرسد و نوع آن int number خواهد بود . اما اگر به کارکتر . برسیم به state=7 منتقل میشویم . case =6 نهایی خواهد بود.

case 7,8:

در صورت مشاهده کارکتر . بعد از digits به این state=7 آمده ایم پس اگر کارکتر بعدی نیز digits باشد با عدد float مواجه هستیم(case 8) .(اگر بعد از . هر کارکتر دیگری دریافت کنیم بی معنی است و در زبان ما وجود ندارد پس پیام درستی را به کاربر نشان خواهیم داد).

case 9,10,11,12,13:

در این state ها کارکتر های) (; } { خوانده شده اند و در state نهایی هستیم.پس token شناسایی شده و نوع آن مشخص میشود .

case 14,15:

کارکتر & به تنهایی دارای معنی نیست پس اگر کارکتر بعدی نیز & نباشد باید پیغامی به کاربر نشان داده شود اما در صورت مشاهده & به state 15 خواهیم رفت که state نهایی شناسایی عملگر منطقی and است .

case 16,17:

در این sate ها همانند state های 14 و 15 عملگر منطقی or شناسایی میشود .

case 18:

کارکتر ٪ تشخیص داده شده و در state نهایی هستیم . پس token mod شناسایی میشود .

case 19,20,21,22,23,24:

کارکتر / به تنهایی به معنای عمل تقسیم است پس در صورت مشاهده میتوانیم در یک state نهایی باشیم اما اگر کارکتر بعدی / یا * باشد یعنی شروع یک کامنت را داریم (// کامنت تک خطی و /* کامنت چند خطی)

State=20 : در صورتی به این state میرسیم که // را مشاهده کرده باشیم پس مطمینا با یک کامنت مواجه هستیم و تازمانی که به خط بعدی نرفته ایم مشاهده هر کارکتری مجاز است و جزوی از کامنت حساب میشود و هر بار به token_name اضافه میشود .

State های 22 و 23 نیز مانند 20و21 پیاده سازی شده اند و شروع کامنت چند خطی را مشاهده کرده اند .

(حالت استثنا : اگر در خط پایانی باشیم خط بعدی وجود ندارد پس در کد بررسی کردیم که اگه به $ رسیدیم(پایان کد) شناسایی کامنت به پایان رسیده است و نیازی به خط بعد رفتن نیست)

پایان کامنت چندخطی با مشاهده * و / پشت سرهم اتفاق می افتد .

(حالت استثنا 2 : اگر * و / پشت هم دیده نشده اند تا پایان کد به این معناست که کاربر فراموش کرده است پایان کامنت خود را قرار دهد که اگر /* یا فقط / را فراموش کند پیغامی مناسب به او نشان خواهیم داد .

case 25,26:

اگر کارکتر " را مشاهده کرده باشیم به معنای شروع یک string خواهد بود که وارد state =25 میشویم . بعد از این کارکتر تا زمانی که کارکتری به غیر از " میبینیم در state =25 باقی میمانیم و به token_name اضافه میکنیم در صورت مشاهده " به پایان رشته رسیده ایم و پس به state نهایی 26 خواهیم رفت و token از نوع string شناسایی شده است

حالت استثنا : اگر تا پایان کد کارکتر " (برای بار دوم که معنای پایان رشته است) را نبینیم به این معناست که کاربر فراموش کرده است پس پیغامی مناسب به او نشان خواهیم داد.

Case 27 :

کارکتر * شناسایی شده است و ما به state=27 آمده ایم . این state نهایی است پس token از نوع multiple شناسایی میشود .

Case 28,29 :

کارکتر = شناسایی شده است و ما به state=28 آمده ایم . این state نهایی است و به تنهایی میتواند token از نوع equal باشد . اما اگر کارکتر بعدی = باشد به state 29 میرویم که state نهایی شناسایی token equaltwice می باشد .

Case 30,31 :

کارکتر > شناسایی شده است و ما به state=30 آمده ایم . این state نهایی است و به تنهایی میتواند token

از نوع less باشد . اما اگر کارکتر بعدی = باشد به state 31 میرویم که state نهایی شناسایی token lessequal می باشد .

Case 32,33 :

کارکتر< شناسایی شده است و ما به state=32 آمده ایم . این state نهایی است و به تنهایی میتواند token از نوع more باشد . اما اگر کارکتر بعدی = باشد به state 33 میرویم که state نهایی شناسایی token moreEqual می باشد .

Case 34,35 :

کارکتر ! شناسایی شده است و ما به state=34 آمده ایم . اما اگر کارکتر بعدی = باشد به state 35 میرویم که state نهایی شناسایی notequal, token می باشد در غیر این صورت در state 34 می مانیم که stste نهایی نیست و پیغام خطا چاپ میکنیم .(token از نوع err برمیگرداند)

Case 36 :

$ Token شناسایی شده است

Case 37 :

space , \t , \n دیده شده است و به state 37 آمده ایم .در این state ، currentState را مساوی 0 قرار میدهیم و token را خالی میکنیم .

Case 40 :

با مشاهده ' به state =40 منتقل شدیم . در این state با مشاهده یک کارکتر به state=41 منتقل میشویم .

Case 41,42

برای مشخص شدن پایان کارکتر باید کارکتر ' دوباره مشاهده شود که در صورت مشاهده به state 41 میرویم اما در غیر این صورت اگر تا پایان کد این کارکتر را نبینیم باید به کاربر پیغام مناسبی را نشان دهیم .state 42 یک state نهایی است و شناسایی token از نوع character انجام میشود .

در کلاس Lexical_Analysis آرایه ای از token ها به نام list ساخته شده است که token های شناسایی شده در این آرایه ذخیره میشوند .

isID

در این متد در ابتدا در آرایه list جستجو میکنیم تا token هایی از نوع id شناسایی کنیم . id شناسایی شده به متد setValueAndType داده میشود تا مقدار و نوع آن set شود سپس این id در آرایه ای از token ها به اسم Symbol_table ذخیره میشود .