دسته بندی زبان های برنامه سازی

۱- زبان هایی با مبنای عددی (FORTRAN –ALGOL)

۲- زبان های تجاری (COBOL)

۳- زبان های هوش مصنوعی (LISP –Prolog)

۴- زبان های سیستم (Java , C++ , C)

معیارهای یک زبان خوب

۱- وضوح، سادگی، یکپارچگی

۲- طبیعی بودن برای کاربرد

۳- سادگی بازبینی برنامه

۴- محیط برنامه نویسی

۵- هزینه استفاده

۶- قابلیت حمل برنامه

۷- پشتیبانی از انتزاع

۸- قابلیت تعامد

قابلیت تعامد

قابلیت تعامد(orthogonality) در زبان برنامه سازی یعنی مجموعه نسبتاً کوچکی از ساختارهای اولیه بتوانند به چند روش با هم ترکیب شوند و ساختمان داده‌ها و ساختارهای کنترلی یک زبان را بسازند.

مثال

در زبان اسمبلی (برای کامپیوترهای بزرگ IBM)، برای جمع کردن دو مقدار صحیح ۳۲ بیتی دو دستورالعمل داریم:

A R1 , M

محتویات ثبات و حافظه جمع شده و در ثبات ذخیره می‌شود.

AR R1 , R2

محتویات دو ثبات جمع شده و در ثبات R2 ذخیره می‌شود.

دستور جمع در زبان اسمبلی (برای ریز کامپیوترهای VAX) :

ADDL O1 , O2

محتویات دو اپراند را جمع کرده و در اپراند ۲ ذخیره می‌کند. در این حالت هر یک از عملوندها می‌توانند ثبات یا محل حافظه باشند.

نتیجه: دستور جمع در VAX یک طراحی متعامد را نشان می‌دهد.

در IBM طراحی متعامد نیست، چون فقط دو ترکیب از چهار ترکیب ممکن برای عملوندها (ثبات یا حافظه) ممکن است.

برای دو حالت دیگر نیاز به دستورالعمل‌های دیگری داریم. در IBM نمی‌توان دو مقدار را جمع کرده و نتیجه را در یک محل حافظه قرار داد.

وجود مجموعه کوچکی از ساختارهای اولیه و مجموعه قوانین سازگار برای ترکیب آنها (تعامد)، بهتر از وجود تعداد زیادی از ساختارهای اولیه است.

وجود قابلیت تعامد زیاد، می‌تواند مشکل ساز باشد. (آزادی در ترکیب ساختارها، منجر به ساختارهای پیچیده‌ای می‌شود.)

متعامدترین زبان برنامه‌سازی ALGOL 68 است.
هر ساختار در این زبان دارای نوع است و هیچ محدودیتی در مورد انواع وجود ندارد.

نحو زبان (syntax)

نحو زبان، بر خوانایی برنامه تاثیر گذار است.

در فرترن ۷۷، طول نام شناسه حداکثر ۶ کاراکتر است و نمی توان اسامی توصیفی را برای شناسه ها تعیین کرد.

در بیسیک اولیه، نام شناسه باید با حرف شروع شود.

شکل و معنا نیز از ملاحظات نحو زبان است. (در C معنای واژه static به محل استفاده آن وابسته است.)

قابلیت خواندن و قابلیت نوشتن را باید در دامنه مسئله در نظر گرفت. (نباید قابلیت پردازش آرایه دوبعدی را در کوبول و فرترن با هم مقایسه کرد، چون فرترن برای این کارها ایده‌آل است.)

پشتیبانی از انتزاع

انتزاع (abstraction): توانایی تعریف و استفاده از ساختارها یا عملیات پیچیده، به طوری که بسیاری از جزئیات نادیده گرفته شوند.

زبان‌های برنامه‌سازی می‌توانند از دو دسته انتزاع پشتیبانی کنند:

۱- انتزاع فرآیند:

می‌توان الگوریتمی مانند مرتب سازی را توسط یک زیر برنامه پیاده سازی کرد، که باعث جلوگیری از تکرار در برنامه و جلوگیری از ترکیب جزییات الگوریتم با برنامه می شود.

۲- انتزاع داده‌ها:

در فرترن از اشاره‌گرها و مدیریت حافظه پویا استفاده نمی‌شود، بنابراین برای پیاده سازی یک درخت که داده‌های نوع صحیح را در گره‌هایش ذخیره می‌کند، از سه آرایه باید استفاده کرد. (دو آرایه برای مشخص کردن گره‌ها و یک آرایه برای نگهداری داده.)
در C ++ می‌توان چنین درختی را با استفاده از انتزاع گره درخت به شکل کلاسی با دو اشاره‌گر و یک مقدار صحیح پیاده سازی کرد.

مدل های زبان

۱- دستوری (imperative) : مانند Ada , c++ , c

در زبان‌های دستوری(رویه ای)، برنامه شامل دنباله‌ای از دستورات است و اجرای هر دستور موجب می‌شود تا مفسر، یک یا چند سلول حافظه را تغییر دهد، یعنی ماشین را به حالت جدیدی ببرد. از دیدگاه این زبان‌ها، حافظه از چند سلول تشکیل شده است.

۲- تابعی (functional) : مانند ML , LISP

در این زبان‌ها، محاسبات به وسیله تابع انجام می‌شود و آخرین تابع، پاسخ را از داده‌های اولیه تولید می‌کند.

۳- منطقی (Logic) : مانند Prolog

در زبان‌های منطقی(قانونمند)، شرطی بررسی می‌شود و در صورت برقرار بودن، فعالیتی انجام می‌شود.

اجرای این زبان شبیه زبان دستوری است با این تفاوت که دستورات به ترتیب اجرا نمی شوند بلکه فعال شدن شرط ها ترتیب اجرا را تعیین می کند.

این زبانها اغلب در برنامه های کاربردی هوش مصنوعی و سیستمهای خبره مورد استفاده قرار می گیرند.

۴- شیء گرا (Object oriented)

در زبان‌های شیءگرا، اشیای موجود در دنیای واقعی مدل سازی می‌شوند.

زبان‌هایی که از برنامه نویسی شیءگرا پشتیبانی می‌کنند به عنوان طبقه جداگانه‌ای از زبان‌ها در نظر گرفته نمی‌شود.

کنترل نوع

کنترل نوع مشخص می‌کند که قبل از اجرای یک عمل، تعداد و نوع عملوندهای آن درست باشد.

کنترل نوع ایستا : کنترل نوع در زمان ترجمه (توسط کامپایلر)

کنترل نوع پویا : کنترل نوع در زمان اجرا

در Java کنترل نوع ایستا انجام می‌شود. بنابراین تمام خطاهای نوع، در زمان ترجمه از بین می‌روند.

امتیازات کنترل نوع ایستا : افزایش سرعت اجرا و استفاده بهینه از حافظه .

عیب کنترل نوع ایستا : عدم قابلیت انعطاف

معایب کنترل پویا

۱- مصرف حافظه بیشتر (چون لازم است اطلاعات مربوط به نوع در زمان اجرا نگهداری شود)

۲- دشوار بودن اشکال زدایی برنامه و حذف تمام خطاهای نوع

۳- کنترل نوع پویا باید به صورت نرم افزاری پیاده سازی شود.

کنترل نوع پویا گران تمام می‌شود.

اطلاعات مورد نیاز برای انجام کنترل نوع ایستا

۱- تعداد، ترتیب و نوع پارامترها و نتایج آن (برای هر عمل)　　۲- نوع هر متغیر　　۳- نوع هر ثابت

با وجود اینکه تمام متغیرها در C و C++ به طور ایستا به انواع بایند می‌شوند، تمام خطاهای نوع را نمی‌توان با کنترل نوع ایستا تشخیص داد.

یونیون‌ها سلول‌های حافظه‌ای هستند که در زمان‌های مختلف، مقادیری از نوع متفاوت را ذخیره می کنند. در این حالت کنترل نوع پویا است.

کنترل نوع قوی (strong typing)

زبانی با کنترل نوع قوی، زبانی است که همیشه بتوان خطاهای نوع را تشخیص داد.

زبان‌های C و C++ از نظر کنترل نوع قوی نیستند، زیرا هر دو دارای ساختار union هستند که از نظر نوع کنترل نمی‌شوند.

زبان C# از نظر کنترل نوع، تقریباً قوی است. تبدیل نوع به صورت صریح انجام می‌شود و در نتیجه خطاهای نوع مشخص می‌شوند و خطایی بدون کشف نمی‌ماند.

بایند (Binding)

هنگام پیاده سازی و یا اجرای یک برنامه، عنصری از این برنامه می تواند صفتی از مجموعه صفات ممکن را به خود بگیرد، به این عمل بایند گفته می شود.
بایند (انقیاد): محدود کردن یک عنصر برنامه به ویژگی یا صفت خاص .
بایند یک وابستگی است، مانند وابستگی بین یک متغیر و یک مقدار.

زمان‌های بایند

۱- زمان اجرا (مانند بایند متغیر به مقدار)
۲- زمان ترجمه(کامپایل) (مثلاً در Java ، متغیر در زمان ترجمه به نوع خاصی بایند می‌شود.)
۳- زمان پیاده سازی زبان (مانند بایند نوع int به بازه‌ای از مقادیر در زبان C)
۴- زمان تعریف زبان (مانند شکل دستورات، انواع ساختمان داده.)

مجموعه مقادیر ممکن برای یک متغیر از هر نوع در زمان پیاده سازی مشخص می شود و ممکن است این مجموعه مقادیر در پیاده سازی های مختلف تفاوت داشته باشد

اینکه یک نوع در زبان برنامه نویسی موجود باشد یا نباشد در زمان تعریف زبان، اینکه یک متغیر در تعریف برنامه از چه نوعی باشد در هنگام ترجمه زبان

و مقدار یک متغیر در هر لحظه در هنگام اجرای زبان می باشد.

مثال

تعیین بایندها و زمان‌های بایند برای دستورات زیر :

```
int   x;
x = x*2;
```

۱- بایند متغیر x به نوع int در زمان ترجمه

۲- بایند مجموعه مقادیر ممکن برای متغیر x در زمان پیاده سازی (طراحی کامپایلر)

۳- بایند نماد * برای عمل ضرب در زمان طراحی زبان

۴- تعیین مقدار x در نقطه‌ای از اجرای برنامه با دستور انتساب (بایند در زمان اجرا)

۵- انتخاب نمایش‌دهدهی برای عدد ۲ .

انواع بایند

۱- زودرس (early) : در زمان ترجمه (مانند C و فرترن) (کارایی زیاد و انعطاف پذیری کم)

۲- دیررس (Late) : در زمان اجرا (مانند ML و lisp) (کارایی کم و انعطاف پذیری زیاد)

بایند در زمان اجرا

۱- هنگام ورود به زیر برنامه یا بلوک (مانند بایند پارامتر رسمی به پارامتر واقعی در زبان C)

۲- در نقطه خاصی از اجرای برنامه (مانند بایند متغیر به مقادیر در ضمن عمل انتساب)

بایند در زمان ترجمه

۱- توسط برنامه نویس (مانند انتخاب نام برای متغیر توسط برنامه نویس)

۲- توسط مترجم (مانند شکل ذخیره آرایه (سطری یا ستونی))

۳- توسط بارکننده (هنگامی که برنامه ها متشکل از چند زیربرنامه هستند هنگام بارکردن آنها در حافظه، آدرس متغیرهای موجود در زیربرنامه ها، باید به آدرس واقعی در کامپیوتر بایند شوند.)

بسیاری از تفاوت های بین زبانهای برنامه نویسی، در واقع به تفاوت زبانها در زمان بایند برمی گردد و اغلب وابسته به این است که بایند در زمان ترجمه صورت می گیرد یا در زمان اجرا.

به عنوان مثال در Fortran کارکردن با آرایه های بزرگ، ساده ولی در ML مشکل است.

اغلب بایندها در فرترن در زمان ترجمه و در ML در زمان اجرا است.

بنابراین Fortran بایندها را فقط یکبار در زمان ترجمه انجام میدهد. در حالیکه ML بیشتر وقت خود راصرف ایجاد و حذف بایندها در زمان اجرا می کند.

انعطاف پذیری ML در دستکاری رشته ها بیشتر از Fortran است چراکه در زبان Fortran اندازه رشته ها در زمان ترجمه باید مشخص و معین باشد ولی در ML اینگونه بایندها می توانند تا خواندن رشته از ورودی به تعویق بیفتد.

بنابراین عموماً کارایی (یاسرعت) یک زبان با انعطاف پذیری آن نسبت عکس دارد.

بنابراین زمان بایند می تواند روی انعطاف پذیری و سرعت برنامه مؤثر باشد.

نوع متغیرها از طریق اعلان صریح (explicit) یا اعلان ضمنی(implicit) مشخص می‌شود.

اعلان صریح، دستوری است که اسامی و انواع متغیرهای موجود در برنامه را مشخص می‌کند. همچنین طول عمر آن‌ها را نیز تعیین می‌کند.

مثال: توسط دستور int x; در زبان C ، متغیر x به طور صریح از نوع int اعلان شده است.

اعلان ضمنی: بدون استفاده از دستور اعلان، نوع متغیرها را مشخص می‌کند.

مثال: در زبان Fortran اگر نوع متغیری تعیین نشود، اگر با یکی از حروف I , J , K , L, M یا N شروع شود، به طور ضمنی از نوع صحیح اعلام می‌گردد.

اعلان ضمنی به قابلیت اعتماد برنامه آسیب می‌رساند، چون کامپایلر در یافتن خطاهای برنامه نویس، دچار مشکل می‌شود.

اعلان موجب تخصیص حافظه نمی‌شود، ولی تعریف موجب تخصیص حافظه می‌شود.

الگوی تابع در C، آن را اعلان می‌کند و تعریف تابع، آن را کامل می‌کند.

دسته بندی متغیرها (Data Objects)

ایستا	این متغیرها قبل از شروع اجرای برنامه به سلول‌های حافظه بایند می‌شوند و تا خاتمه برنامه، به آن سلول‌ها بایند می‌مانند.
پویای پشته‌ای	متغیرهایی که بایندهای حافظه آن‌ها وقتی صورت می‌گیرد که دستورات اعلان آن‌ها ایجاد شود، اما نوع آن‌ها به طور ایستا بایند می‌شود. در C++ و C# و Java ، متغیرهایی که در متدها تعریف می‌شوند، از این نوع هستند.
پویای صریح heap	سلول‌های حافظه بی‌نامی هستند که با دستوراتی که توسط برنامه نویس مشخص می‌کند، تخصیص می‌یابند و آزاد می‌شوند. این متغیرها از heap تخصیص می‌یابند و توسط اشاره‌گرها یا متغیرهای مرجع قابل دستیابی‌اند.
پویای ضمنی heap	این متغیرها فقط وقتی حافظه heap بایند می‌شوند که مقادیری به آن‌ها نسبت داده شود. در این صورت، تمام صفات آن‌ها بایند می‌شوند. از این نظر آن‌ها اسامی هستند که به هرشکلی می‌توان از آن‌ها استفاده کرد.

کاربردهای متغیر ایستا

۱- متغیرهای عمومی

۲- متغیرهای static در یک زیر برنامه

آدرس‌دهی به متغیرهای ایستا به طور مستقیم انجام می‌شود و به همین علت، کارایی آنها بالا است.

عیب بایند حافظه به طور ایستا

۱- کاهش قابلیت انعطاف. (در زبانی که فقط متغیرهایی دارند که به طور ایستا به حافظه بایند می‌شوند، نمی‌توان از بازگشتی استفاده کرد.)

۲- نمی‌توان حافظه را بین چند متغیر به طور اشتراکی به کار برد.

فواید متغیر پشته ای:

۱- زیر برنامه‌های بازگشتی به حافظه محلی پویا نیاز دارند که این نیاز با متغیرهای پویای پشته‌ای برآورده می‌شود.

۲- تمام زیر برنامه‌ها از یک فضای حافظه برای متغیرهای محلی خود استفاده می‌کنند.

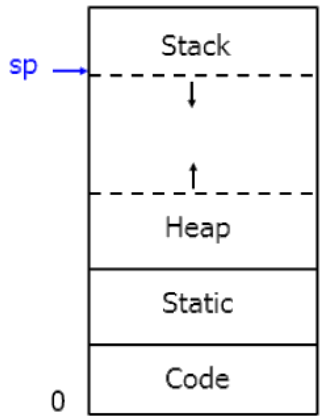
معایب متغیر پشته ای :

۱- سربار زمان اجرا (ناشی از تخصیص و آزاد سازی)

۲- کند بودن سرعت دستیابی (به دلیل آدرس دهی غیر مستقیم و این که زیر برنامه‌ها نمی‌توانند به سابقه اجرا حساس باشند)

متغیر پویا

heap : مجموعه‌ای از سلول‌های حافظه است که سازمان آن، نظم خاصی ندارند، چون استفاده از آن قابل پیش بینی نیست.



در c++ ، تخصیص حافظه از heap ، توسط عملگر new و آزادسازی آن توسط عملگر delete انجام می شود.

از اشیای پویای صریح heap معمولاً برای ایجاد لیست پیوندی و درخت‌ها استفاده می‌شود.

عیب متغیر پویای صریح heap

۱- دشوار بودن کار با اشاره‌گرها و متغیرهای مرجع
۲- هزینه ارجاع به متغیرها
۳- تخصیص‌ها و آزاد سازی ها.

عیب متغیر پویای ضمنی هیپ

۱- دور ماندن خطاها از دید کامپایلر

۲- نگهداری صفات پویا (مثل نوع اندیس آرایه و بازه آن‌ها)

گرامر

گرامرها : راهکار تولید رسمی زبان .

در سال ۱۹۵۵ ، چامسکی چهار دسته از گرامرها را توصیف کرد که چهار دسته از زبان‌ها را تعریف می‌کنند.

گرامر G به صورت چهار تایی $G = (V, T, S, P)$ تعریف می‌شود که :

V : مجموعه متناهی از اشیاء به نام متغیرها

T : مجموعه متناهی از اشیاء به نام سمبل‌های پایانی (ترمینال)

S : سمبل ویژه ای به نام متغیر شروع $(S \in V)$

P : مجموعه متناهی از قوانین $x \rightarrow y$

$x \in (V \cup T)^+$

$y \in (V \cup T)^*$

گرامر مستقل ,

گرامر مفروض $G=(V,T,S,P)$ در صورتی مستقل از متن خوانده می شود که تمام قوانین P به فرم $A\to x$ باشند که در آن $x\in(V\cup T)^*$ و $A\in V$.

به طور کلی شرط مستقل از متن بودن این است که در سمت چپ قوانین، فقط یک متغیر وجود داشته باشد.

$S\to Aab$

$A\to Aab$

$A\to a$

I $\{a,b\}$

مثال

گرامری برای مجموعه اعداد صحیح در C (بدون محدویت برای تعداد ارقام)

$S \rightarrow AB$

$A \rightarrow + \mid - \mid \lambda$

$B \rightarrow D \mid DB$

$D \rightarrow 0|1|2|3|4|5|6|7|8|9$

مثال

$S \rightarrow aSb \mid ab$ $\quad L = \{a^n b^n : n \geq 1\}$

$S \rightarrow aSb \mid b$ $\quad L = \{a^n b^{n+1} : n \geq 0\}$

$S \rightarrow aSbb \mid \lambda$ $\quad L = \{a^n b^{2n} : n \geq 0\}$

گرامر BNF

Backus-Naur Form

کمی بعد از چامسکی، نشانه گذاری به نام BNF ارائه شد، که ابداء خوبی برای توصیف **نحو زبان** می‌باشد.

BNF تقریباً معادل گرامرهای مستقل از متن چامسکی است.

BNF یک متا زبان برای زبان‌های برنامه سازی است. (زبانی که برای توصیف زبان دیگری به کار می‌رود را **متا زبان** گویند.)

دستورالعمل انتساب ساده در C می‌تواند با انتزاع <assign> نمایش داده شود:

<assign> → <var> = <expression>

به نماد سمت چپ پیکان LHS و به متن موجود در سمت راست پیکان، RHS می‌گویند.

نمونه‌ای از جمله‌ای که توسط این قانون (rule) تعریف می‌شود، عبارت است از:

sum = a + b

انتزاع‌ها در توصیف BNF را غیر پایانه (non terminal) و لغات و نشانه‌های قوانین را پایانه می‌نامند.

اشتقاق برنامه توسط گرامر :

<program> → begin < s > end
< s > → <stmt> | <stmt> ; < s >
<stmt> → <var> = < e >
<var> → A | B | C
< e > → <var> + <var> | <var> - <var> | <var>

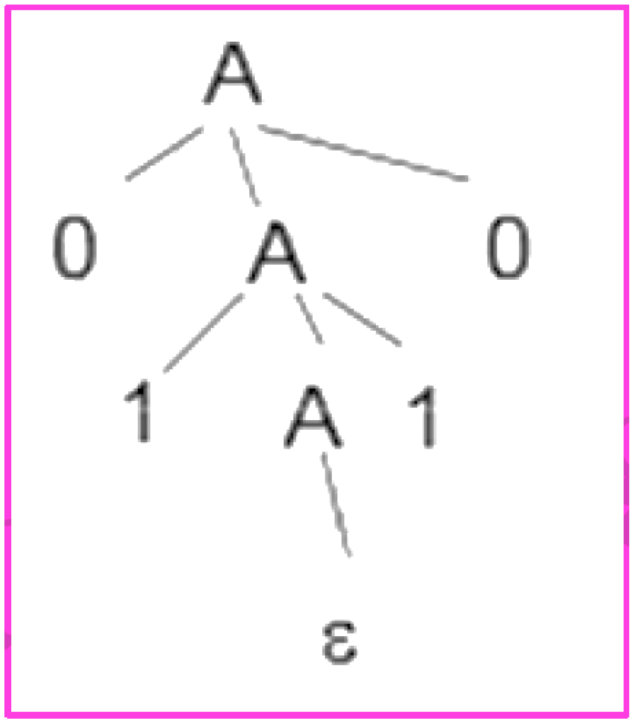
```
begin
    A = B + C;
    B = C
end
```

<program> ⇒**begin** **< s >** **end**
begin <stmt> ; <s > end
begin <var> = < e >; < s > end
begin A = < e >; < s > end
begin A = <var> + <var>; < s > end
begin A = B + <var>; < s > end
begin A = B + C; < s > end
begin A = B + C; <stmt> end
begin A = B + C; <var> = <e> end
begin A = B + C; B = <e> end
begin A = B + C; B = <var> end
begin A = B + C; B = C end

مثال

G:

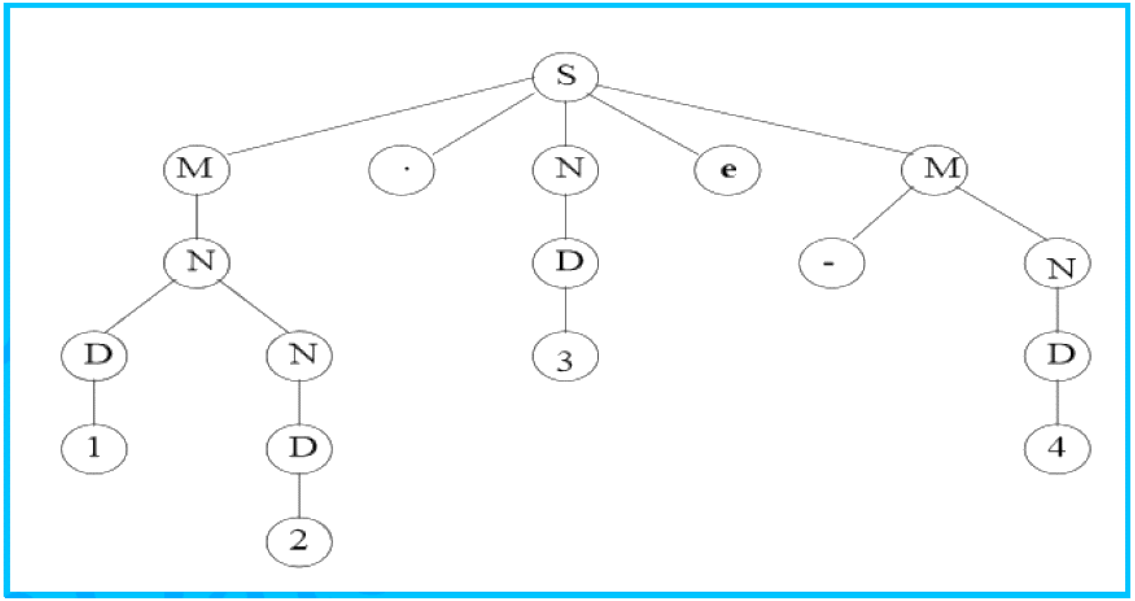
A => 0A0 | 1A1 | 0 | 1 | ε



0110

مثال

$$
\begin{aligned}
S &\rightarrow M.N|M\mathbf{e}M|M.N\mathbf{e}M \\
M &\rightarrow N|+N|-N \\
N &\rightarrow DN|D \\
D &\rightarrow 0|1|2|3|4|5|7|8|9
\end{aligned}
$$



$$S \Rightarrow \underline{M}.N\mathbf{e}M \Rightarrow \underline{N}.N\mathbf{e}M \Rightarrow \underline{D}N.N\mathbf{e}M \Rightarrow 1\underline{N}.N\mathbf{e}M \Rightarrow 1\underline{D}.N\mathbf{e}M \Rightarrow$$

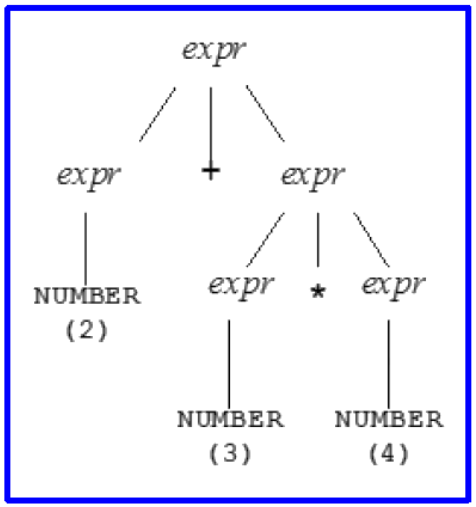
$$12.\underline{N}\mathbf{e}M \Rightarrow 12.\underline{D}\mathbf{e}M \Rightarrow 12.3\mathbf{e}\underline{M} \Rightarrow 12.3\mathbf{e}-\underline{N} \Rightarrow 12.3\text{e-}\underline{D} \Rightarrow 12.3\text{e-}4$$

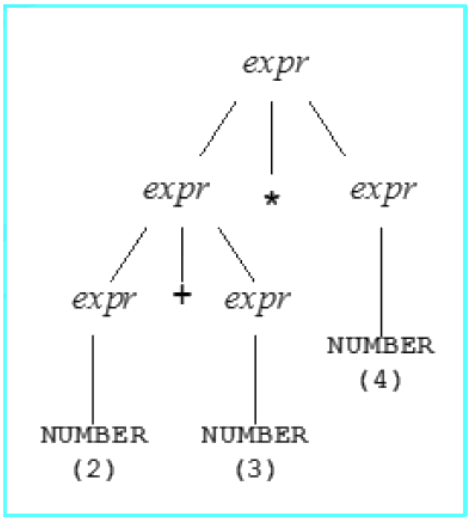
گرامر مبهم

اگر برای یک شبه جمله تولید شده توسط گرامری، بیش از یک درخت تجزیه وجود داشته باشد، آن گرامر مبهم می باشد.

expr –> expr + expr | expr * expr | (expr) | NUMBER

- 2 + 3 * 4

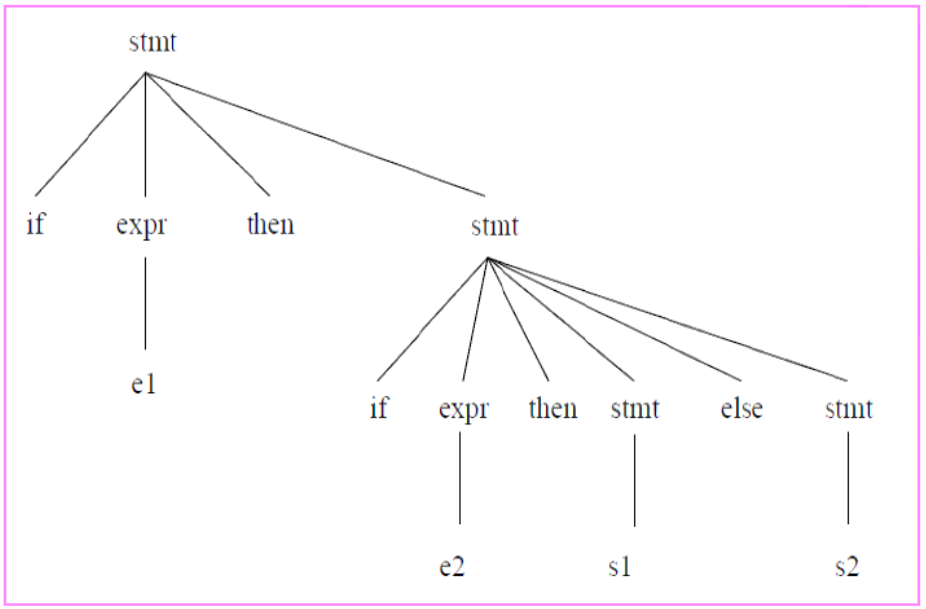


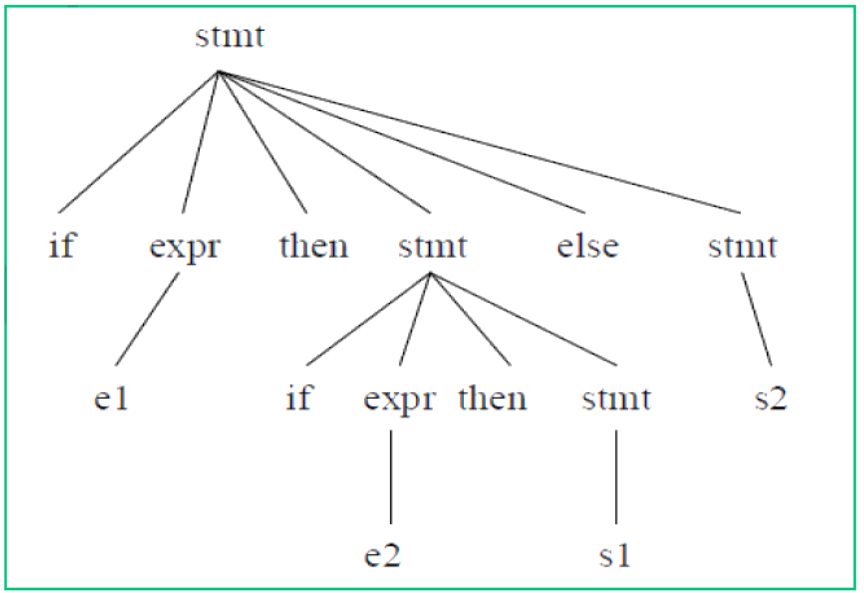


مثال

گرامر زیر مبهم است، چون برای جمله If e1 then if e2 then s1 else s2 ، دو درخت تجزیه وجود دارد:

stmt -> if expr then stmt
stmt -> if expr then stmt else stmt
stmt -> other





روش‌های پیاده سازی زبان‌های برنامه سازی

۱- پیاده سازی کامپایلری

۲- تفسیر محض

۳- مختلط (hybrid)

- زبان های مفسری: لیسپ ، ام ال، پرل ، پست اسکریپت و اسمالتاک
- زبان های کامپایلری : C,C++ ، فرترن ، پاسکال و ادا .

syntax زبان : شکل عبارات، دستورالعمل‌ها و واحدهای برنامه.

semantic زبان : معنای عبارات، دستورالعمل‌ها و واحدهای برنامه.

مثال: Syntax دستورالعمل while در C :

while (<عبارت منطقی>)
<statement>

semantic این دستورالعمل :

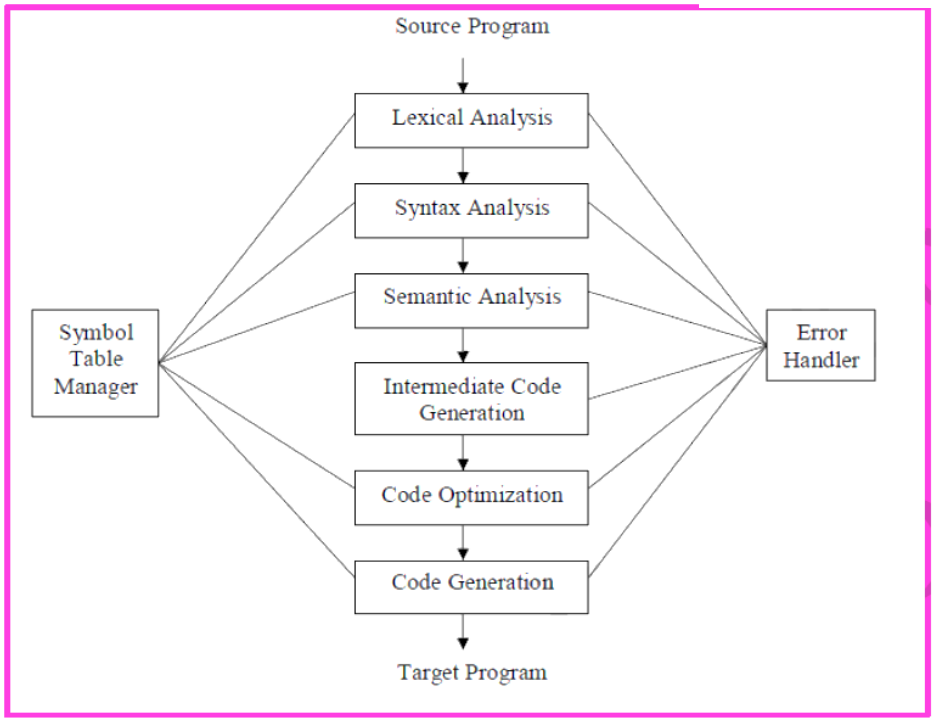
اگر مقدار فعلی عبارت بولی درست باشد، دستورالعمل موجود در آن اجرا می‌شود. سپس کنترل اجرا به طور ضمنی به عبارت بولی برمی‌گردد تا این فرایند تکرار شود.

Syntax و Semantic زبان کاملاً به هم مربوط هستند.

اگر زبان به خوبی طراحی شده باشد، semantic باید از syntax مشخص باشد، یعنی شکل دستورالعمل باید بیانگر معنای آن باشد.

توصیف Syntax آسان‌تر از توصیف semantic است، چون نشانه گذاری جهانی قابل قبولی هنوز برای توصیف semantic وجود ندارد.

فرآیند کامپایل



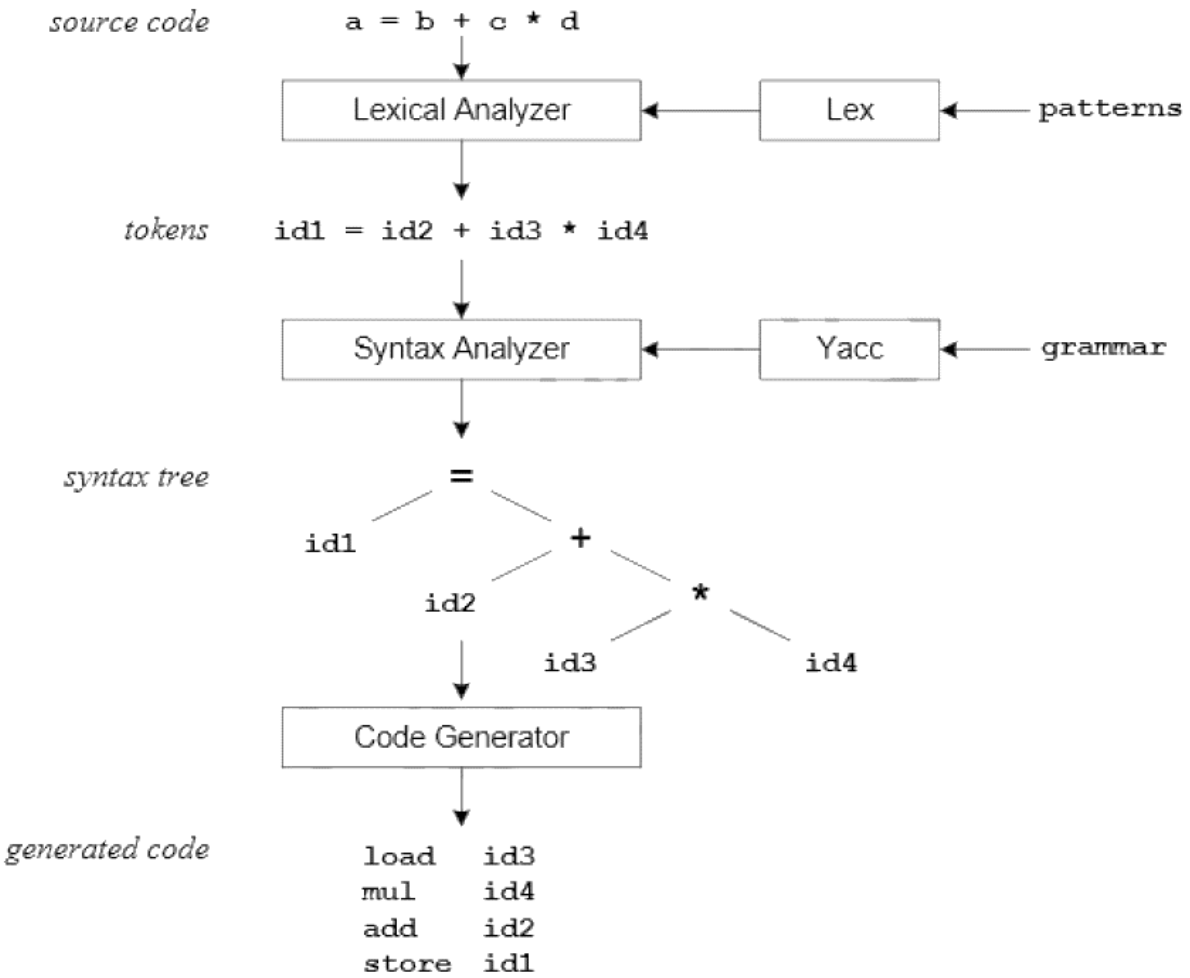
تحلیل‌گر لغوی، کاراکترهای برنامه مبدا را به واحدهای لغوی تبدیل می‌کند.
واحدهای لغوی شامل شناسه ها، واژه های خاص، عملگرها و نمادهای نشانه گذاری است.

تحلیل‌گر نحوی، واحد لغوی را از تحلیل گر لغوی گرفته و به کمک آن، ساختار سلسله مراتبی به نام درخت‌های تجزیه (Parse trees) می‌سازد.
این درخت‌های تجزیه، ساختار نحوی برنامه را نشان می‌دهند.

تحلیل‌گر معنایی (Semantic)، بخشی از مولد کد میانی است.
این تحلیل‌گر خطاهایی را می‌یابد که یافتن آنها در مرحله تحلیل نحوی دشوار است.

مولد کد میانی، برنامه را به زبان دیگری تولید می‌کند که در سطح میانی بین برنامه مبدا و نتیجه نهایی کامپایلر (زبان ماشین) قرار دارد.
زبان‌های میانی گاهی شبیه به اسمبلی و گاهی خود اسمبلی می‌باشند.

جدول نمادها (Symbol table)، برای فرآیند کامپایل کردن به عنوان بانک اطلاعاتی عمل می‌کند.
در این جدول، اطلاعات نوع و صفات نام‌هایی که برنامه نویس تعریف کرده است، قرار می‌گیرد.
این جدول توسط تحلیل‌گر لغوی و نحوی پر شده و توسط تحلیل‌گر معنایی و مولد کد از آن استفاده می‌شود.


source code
a = b + c * d
Lexical Analyzer
Lex
patterns
tokens
id1 = id2 + id3 * id4
Syntax Analyzer
Yacc
grammar
syntax tree
=
id1
+
id2
*
id3
id4
Code Generator
generated code
load id3
mul id4
add id2
store id1