فصل اول

آشنایی با اصول طراحی کامپایلر

سیر تکاملی زبان های برنامه نویسی

۱. زبان ۰۱ یا زبان ماشین: چون ماشین فقط متوجه زبان 01 هست نیازی به ترجمه برنامه برای کامپیوتر نبود.

۲. زبان اسمبلی :کار با برنامه نویسی با زبان اسمبلی کار سخت و دشواری هست چرا؟

چون برنامه نویس باید درگیر معماری کامپیوتر بشه و کلی سختی داره و در نهایت با برنامه مترجمی به نام اسمبلر مانند turbo assembeler و marco assembeler برای ماشین ترجمه می شد.

۳. زبان های سطح بالا: مثل؟

کامپیوتر فقط ۰۱ می دونه پس نیاز به ترجمه داریم.

روش های ترجمه و اجرای زبان سطح بالا

۱. **استفاده از مفسرها**

۲. **استفاده از کامپایلر**

استفاده از مفسر

- مفسر برنامه ای است که برنامه منبع و داده های مورد نیاز آن را به عنوان ورودی دریافت کرده و در خروج، نتیجه اجرای برنامه منبع بر روی داده های وروردی را تولید می کند.

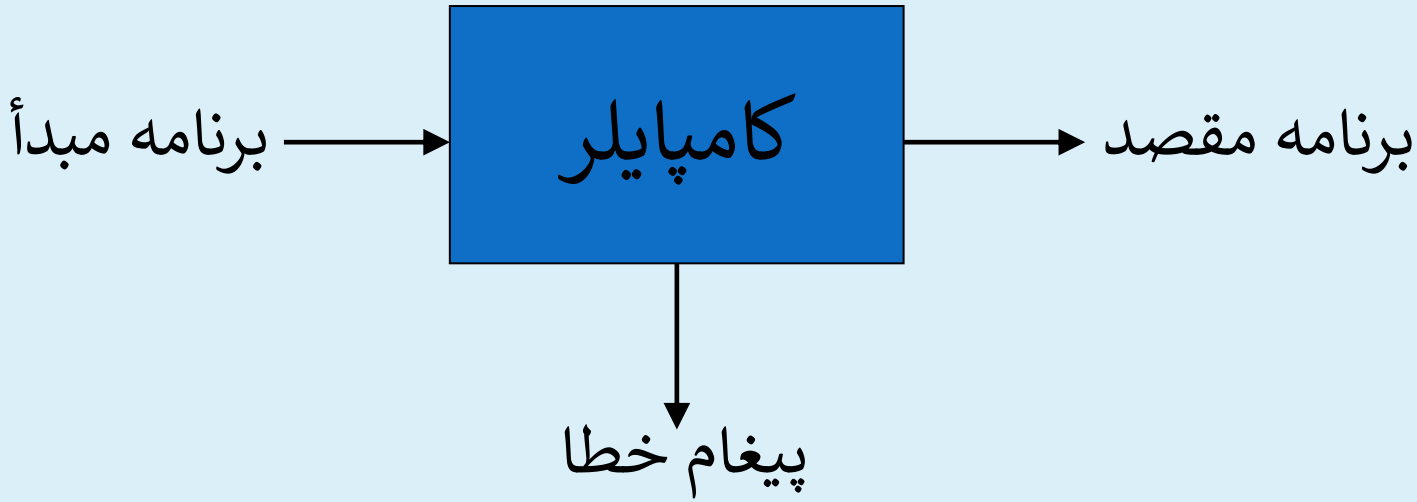
- برنامه مفسر دستورالعمل های برنامه ورودی را یک به یک خوانده، تفسیر و اجرا می کند.

کامپایلر چیست

- انواع نرم افزار ها به صورت متداول:
- ۱- سیستم عامل
- ۲- کاربردی
- ۳- سیستمی
- کامپایلر، یک نرم افزار سیستمی است که یک زبان مبدا(یک زبان سطح بالا) را دریافت می کند و آن را به زبان مقصد تبدیل می کند.

- تفاوت کلی مفسر و کامپایلر: کامپایلر کل برنامه را ترجمه می کند و تبدیل به فایل خروجی می کند و برنامه سپس به مرحله اجرا می رود. ولی مفسر خط به خط ترجمه و اجرا می کند.

ساختار کلی کامپایلر



مزایای مفسر

۱. پیاده سازی آسان: نوشتن یک مفسر آسان تر است و زمان کمتری می برد.

۲. قابلیت حمل بالا: برنامه توسط مفسر اجرا می شود پس هر جایی که مفسر باشد، برنامه نیز مستقل از نوع ماشین و سخت افزار اجرا خواهد شد. اگر در زبان جاوا از مفسر استفاده شود ، روی همه کامپیوتر ها می توان آن را اجرا کرد فقط به شرطی که مفسر حضور داشته باشد.

۳. سهولت اشکال زدایی : نیاز به ترجمه کل برنامه نیست و خط هایی که نیاز هست ترجمه و اجرا می شه چرا؟ چون خط به خط کار ترجمه رو انجام می ده. این ویژگی برای برنامه های در حال توسعه مفید است چون برنامه مدام در حال تغییر است و با هر تغییر نیاز به اجرای مجدد برای بازبینی عملکرد نیست.

مزایای مفسر

۴. قابلیت انعطاف پذیر بالا: مفسر در زمان اجرا حضور دارد و برخی اطلاعات که فقط در زمان اجرا قابل دستیابی است توسط مفسر جمع آوری شده و می تواند بر اساس این اطلاعات تصمیم گیری کند. به عنوان مثال، اگر به تکه کد زیر توجه کنید:

-
- Int A[10],i;
- ....
- i=11;
- A[i]=55;
- اگر این کد توسط کامپایلر ترجمه شود، در حین کامپایل خطایی نمی گیرد چون مقداردهی به متغییر ها در حالت کلی در زمان اجرا اتفاق می افتد پس کامپایلر مقداردهی ها رو در زمان کامپایل بررسی نمی کند. ولی مفسر خط به خط ترجمه و اجرا می کند ! و اگر این تکه کد با مفسر ترجمه شود قطعا خطا خواهد داد.

معایب مفسر

- ۱- سرعت اجرای پایین: در زمان اجرا حضور دارد و خط به خط دستورات را تفسیر و اجرا می کند.

For(i=1; i<=1000;i++) ◦

J= i+j ◦

- مفسر ۱۰۰۱ بار این کد را اجرا می کند ولی کامپایلر فقط یک بار اجرا می کند. **مهم ترین عیب مفسر نسبت به کامپایلر!!!!**

۲- نیاز به مفسر در هر زمان که بخواهیم اجرا کنیم.

۳- دسترسی به کد منبع: برای اجرا حتما باید کد منبع موجود باشد و برخی شرکت ها مایل به این کار نیستند.

مزایای کامپایلر

۱. سرعت اجرای بالا: برنامه یک بار ترجمه می شود و برای هر بار اجرا نیازی به ترجمه نیست و فورا اجرا می شود.

۲. اجرای برنامه مستقل از کامپایلر.

۳. حفاظت از کد منبع برنامه: نیاز به وجود سورس کد نیست.

۴. عدم تکرار کامپایل: یک بار ترجمه می شه و تمام!

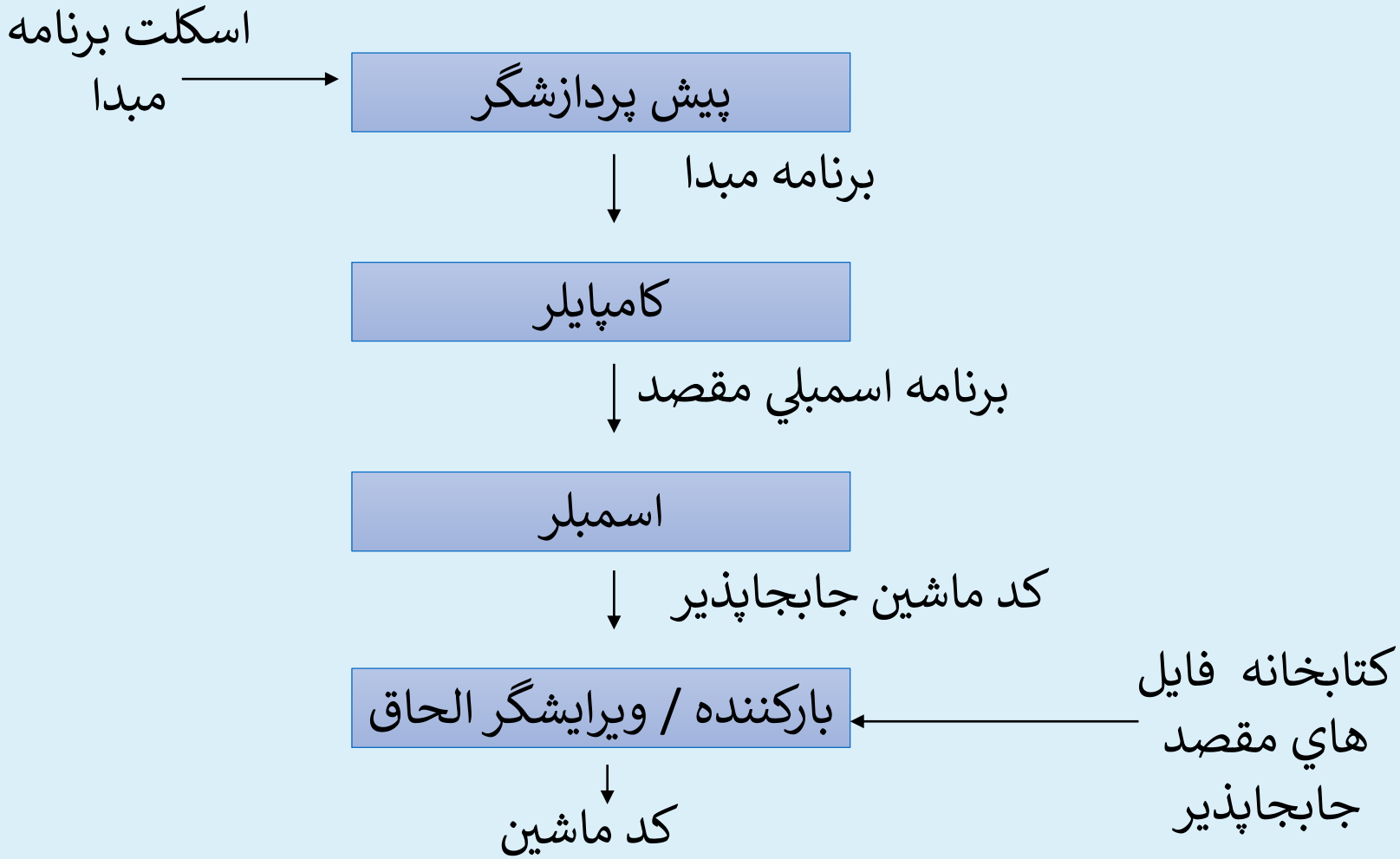
معایب کامپایلر

۱. زمان بر بودن اشکال زدایی: چون هر بار که تغییرات می دیم باید از اول ترجمه بشه.

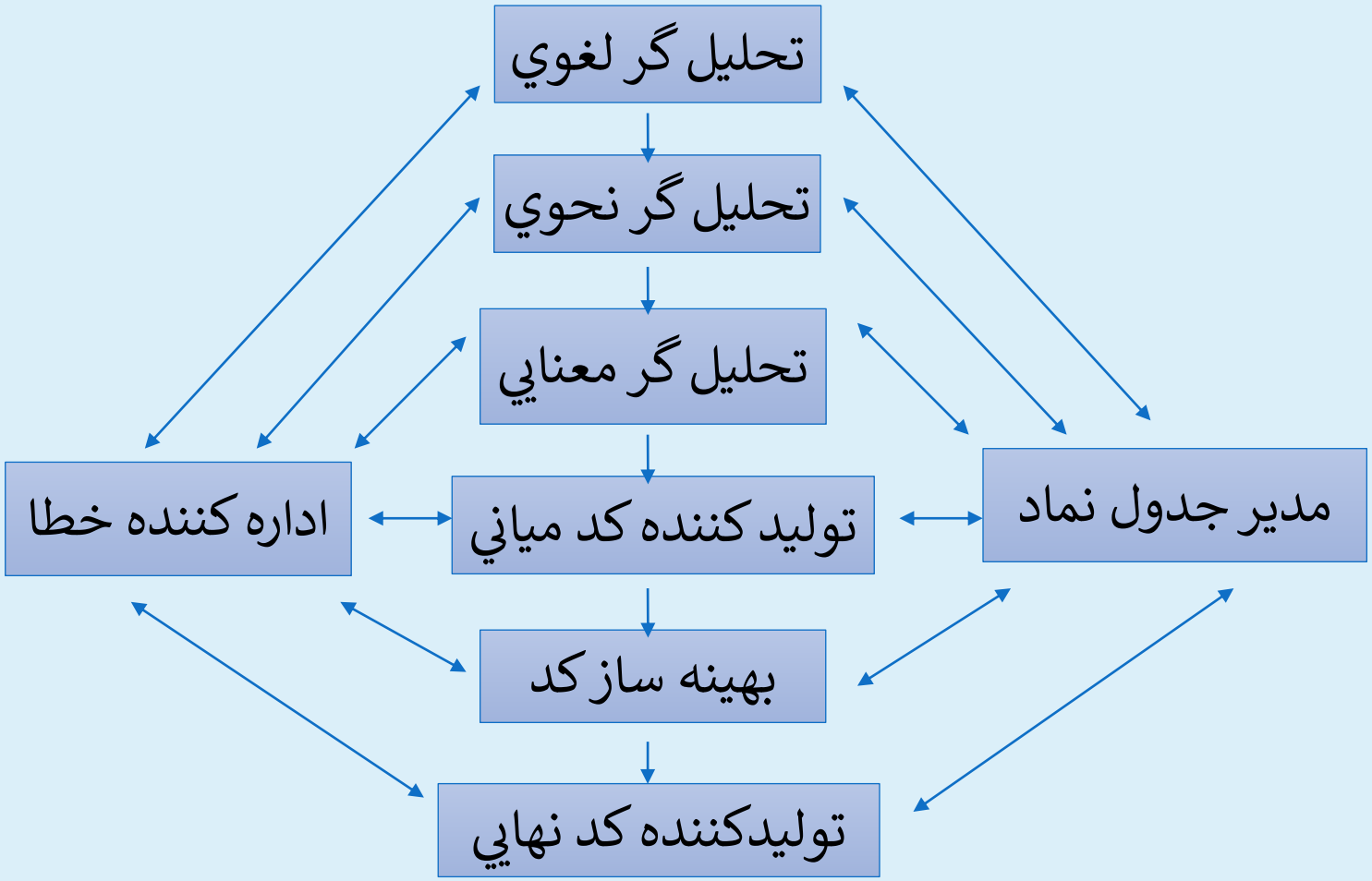
۲. قابلیت حمل پایین: وقتی برنامه ای کامپایل می شه برنامه خروجی کامپایل وابسته به معماری نوع ماشین خواهد بود پس روی هر ماشینی نمی توان از برنامه ترجمه شده با کامپایلر استفاده کرد.

۳. پیاده سازی مشکل.

با توجه به مزایا و معایب نظر شما چیست؟ استفاده از کامپایلر هست یا مفسرها؟


اسکلت برنامه
مبدا
پیش پردازشگر
برنامه مبدا
کامپایلر
برنامه اسمبلي مقصد
اسمبلر
کد ماشین جابجاپذیر
کتابخانه فایل
هاي مقصد
جابجاپذیر
بارکننده / ویرایشگر الحاق
کد ماشین

ساختار کامپایلر



تحلیل گر لغوی(lexical analyzer):

مرور متن برنامه به صورت حرف به حرف

تبدیل آنها به نشانه ها (کلمات کلیدي، عملگر، جداکننده، ثوابت و شناسه)

تحلیل گر لغوی(lexical analyzer):

- وظیفه این مرحله تبدیل برنامه ورودی به دنباله ای از کلمات یا نشانه ها لغوی (token) می باشد. در واقع این مرحله دنباله کاراکترهای تشکیل دهنده زبان مبدا از چپ به راست خوانده شده و به گروه هایی به نام token که معنای واحدی دارند تبدیل می شود.

- <علی به دانشگاه رفت> : <علی و اسم> <به، حرف اضافه> <دانشگاه ، اسم> <رفت، فعل>
- **If (a<100)then b :=sum;**
- **<کلمه رزرو شده، if> <جداکننده، (> <شناسه، a> <عملگر رابطه ای ، < > <عدد ، ۱۰۰>**
  <جداکننده،) > <کلمه رزروشده ، then> <شناسه، b> < عملگر انتساب، := > <شناسه، sum>
  <جداکننده، ;>

- ممکن است تحلیل گر لغوی در مرحله تشخیص یک توکن با خطا مواجه شود ، به خطای شناسایی شده در این مرحله، خطای لغوی یا املایی گفته می شود. هر مرحله از کامپایل ممکن است با خطا مواجه شود.

استراتژی کامپایلرها در برخورد با خطا

۱. در برخورد با اولین خطا متوقف شود و کامپایل ادامه پیدا نکند
۲. در برخرود با اولین خطا نادیده بگیرد وبه فرایند خطا ادامه دهد تا همه خطا ها را پیدا کند.
۳. در برخورد با خطا آن را اصلاح کند و به فرایند کامپایل ادامه دهد.

- مثال برای خطا های لغوی:
- علی طوپ را برداشت.
- A$B=100 که در اینجا $ کاراکتر غیر مجاز است.
- If(3a <100) i=j; اینم شما بگین ؟ چه خطای لغوی دارد؟ 3a
- رفع یا ترمیم خطا توسط مولفه خطا پرداز انجام می شود. روال های رفع خطای مربوط به تمامی مراحل را می توان در مولفه ای به نام خطا پرداز گذاشت.

استراتژی رفع خطاهای لغوی

۱. حذف کاراکتر غیر مجاز
   - تبدیل sa#lary به salary
۲. جایگزین کردن کاراکتر غیر مجاز با کاراکتر مجاز
   - Sa#lary به sa9lary
۳. جا به جا کردن دو کاراکتر
   - تبدیل := به =: در جمله a=:100;
۴. درج یک کاراکتر جدید
   - تبدیل : به =: در جمله a:100;

توجه

- در هر صورت هدف این است که با کمترین تغییر در کلمه، املای کلمه را تصحیح کنیم.
- **به مرحله تحلیل گر لغوی اسکنر یا پویشگر نیز گفته می شود.**
- **دید اسکنر محدود است و فقط با کلمات سر و کار دارد.**
- **معمولا این نوع خطا ها زیاد نیستند**
- **این خطا ها در زمان کامپایل شناسایی می شوند.**
- **این مرحله تنها مرحله ای است که با برنامه ورودی سر و کار دارد.**

انواع توکن های متداول در زبان های برنامه نویسی

۱. شناسه ها: به اسامی متغییرها، توابع، به طور کلی هر اسمی که برنامه نویس در برنامه تعریف می کند، شناسه گفته می شود. در هر زبان شناسه دارای یک الگو یا شکل کلی است. نمونه هایی از نوع توکن id عبارت است از:

- {a,tax,salary,fact,i1,mergsort,....}
- الگوی این شناسه ها یک حرف و به دنبال آن هر تعداد حرف و عدد است.

۲. اعداد: نمونه هایی از توکن num

-
- {25,-77,10.5,11.25,155E10,22.5E-8,...}

انواع توکن های متداول در زبان های برنامه نویسی

۳. کلمات کلیدی:

- {if,for, while,then,else,…}

۴. عملگرها:

- {+,*,/,<,<=,…}

۵. ثابت های رشته ای:

- “this is thest!” در جمله printf(“this is test!”)

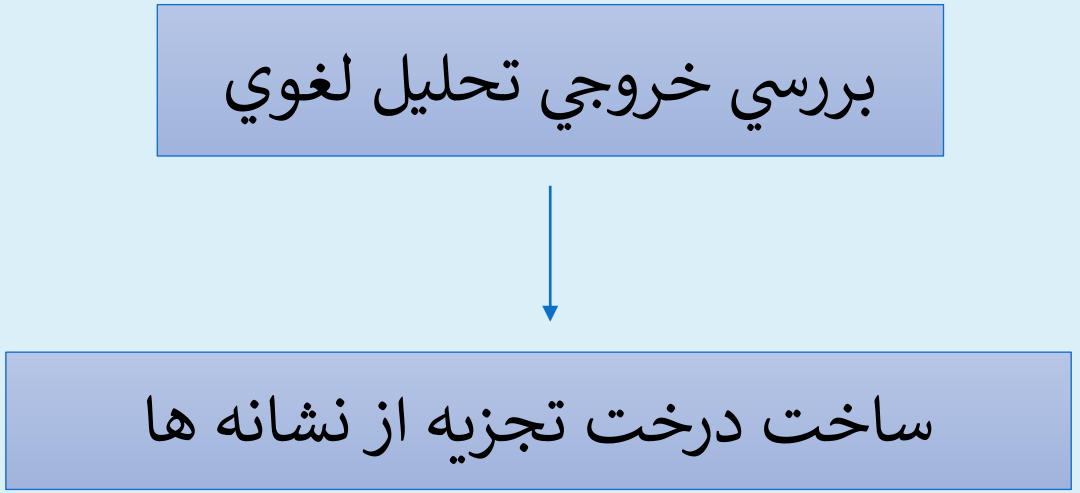
۶. جدا کننده ها مانند {.... ,;,[,],(,) }

تحلیل گر لغوی

- **توضیحات و فضای خالی در این مرحله شناسایی شده و دور ریخته می شوند و به مرحله بعد ارسال نمی شوند.**

- مثال:
- S:= p-q*r/30
- بعد از مرحله اسکنر:
- Id , := ,id, -, id, *, id, /, num

تحلیل گر نحوی



تحلیل گر نحوی

- وظیفه این مرحله بررسی درستی جملات برنامه ها (token ها) از نظر ساختاری، نحوی یا دستوری می باشد. به عبارت دیگر بررسی می کند که دنباله ای از نشانه های لغوی شناسایی شده توسط مرحله قبل(تحلیل گر لغوی یا اسکنر) می توانند از نظر دستوری کنار هم قرار گیرند یا خیر؟
- **بیشتر خطا ها در برنامه ها از این نوع می باشند**
- **به این مرحله همچنین تجزیه گر یا پارسر گفته می شود.**
- **تمامی خطا های این مرحله در زمان کامپایل تشخیص داده می شود.**
- مثال فارسی: علی برداشت توپ را
- A=2++3
- For(i=1, i<100, i++)

تحلیل گر نحوی

- خروجی مرحله تحلیل گر نحوی یک درخت پارس است. خروجی این مرحله درخت تجزیه یا درخت نحو است.
- W=”adbce” به گرامرزیر توجه کنید:

- برای رشته بالا از اشتقاق چپ یا راست می توان استفاده کرد. به طور کلی تجزیه گر ها را به دو دسته بالا به پایین و پایین به بالا دسته بندی می کنند.

$$S \rightarrow AbC$$
$$A \rightarrow aA \mid d$$
$$C \rightarrow cC \mid e$$

تحلیل گر نحوی

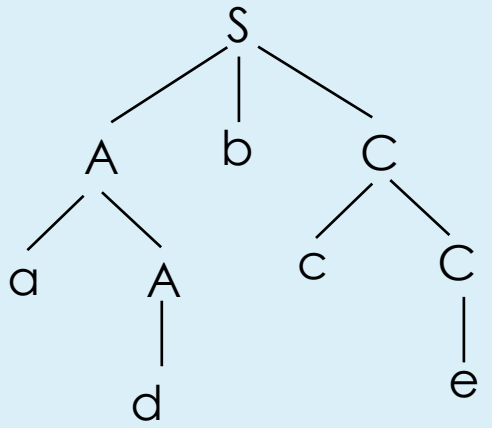
- در تجزیه گر های بالا به پایین درخت تجزیه (اشتقاق) از ریشه به برگ ساخته می شود و از اشتقاق چپ استفاده می شود، و در تجزیه گرهای پایین به بالا درخت تجزیه از برگ ها به سمت ریشه ساخته می شود. راجع به انواع تجزیه گرها در فصل سوم به صورت مفصل بحث می کنیم.

- اگر برای رشته w از اشتقاق چپ استفاده کنیم خواهیم داشت:

$S \Rightarrow AbC \Rightarrow aAbC \Rightarrow adbC \Rightarrow adbcC \Rightarrow adbce$

تحلیل گر نحوی

- تجزیه این رشته با موفقیت انجام شده است و ملاحظه می شود که درخت تجزیه ای ساخته نشده است اگر بخواهیم درخت تجزیه را نمایش دهیم کافی است از همان قواعد استفاده شده در اشتقاق چپ برای نمایش درخت استفاده کنیم.

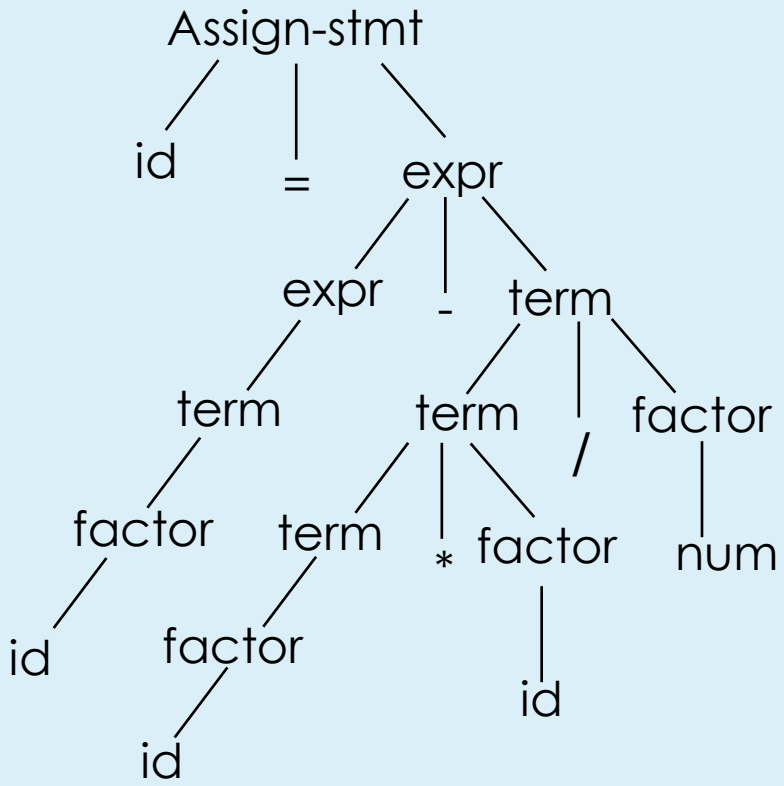


تحلیل گر نحوی

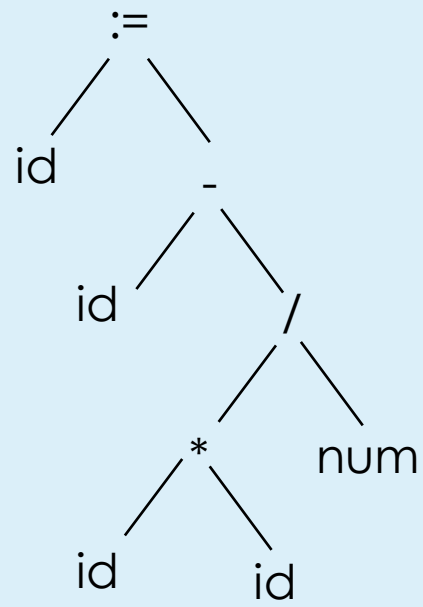
- درخت نحو یک روش نمایش بینابینی است.
- گرامر زیر داده شده است می خواهیم جمله s:=p-q*r/30 را تجزیه کنیم.

Assign-stmt → id := expr
expr→ expr+term | expr-term| term
term→ term*factor|term/factor|factor
factor→id|num

تحلیل گر نحوی



درخت تجزیه



درخت نحو

تحليل گر معنايى

بررسي برنامه مبدا براي يافتن خطاهاي معنايي

جمع آوري اطلاعات مربوط به نوع داده ها

تحلیل گر معنایی

- وظیفه این مرحله این است که بررسی کند که آیا جملاتی که از نظر نحوی درست هستند دارای معنا و مفهوم درست می باشند یا خیر؟
- علی توپ را خورد : از نظر املایی و گرامری درست ولی از نظر معنایی نادرست می باشد.

i=k+d

که:

k=float

d=int

کنترل معنایی

۱. کنترل نوع عملوندهای عملگرها: مثال بالا یا مثال دیگر مثل : i=n+k*m با فرض این که m از نوع رشته باشد و k از نوع صحیح است، این جمله از نظر لغوی و نحوی صحیح است ولی از نظر معنایی درست نمی باشد.

۲. متغیری تعریف شده ولی استفاده نشده باشد.

- این موضوع از نظر برنامه نویسی خطا نیست ولی به هر حال تشخیص داده می شود و در اکثر زبان ها هشداری نیز صادر می شود تا حافظه به صورت بی جا استفاده نشود.

```
Int f(int i , int k){
Int m , n;
m=i+k;
p=m*2;
Return p;
}
```

کنترل معنایی

۱. متغییری استفاده شده باشد ولی تعریف نشده باشد.
۲. پرش به برچسبی که وجود ندارد.
۳. کنترل تعداد و تطابق پارامتر های ظاهری و پارامترهای واقعی توابع .

```
Int f(int m , float k){
…….
}
Int g(){
P=f(“test”,10.5,i);
}
```

کنترل معنایی

۱. کنترل محدوده اندیس آرایه : به عنوان مثال از این تکه کد زیر را در نظر بگیرید:

```
int  A[10],i;
……
Cin>>i;
A[i]=100
```

- کنترل مقدار i در زمان کامپایل نمی تواند انجام گیرد چون کاربر در زمان اجرا مقدار را وارد می کند و این تکه کد دارای مشکل معنایی است.
- کامپایلر می تواند دستورات لازم برای اعمال این کنترل را به کد نهایی اضافه کند تا این دستورات در زمان اجرا انجام شوند. پس کامپایلر به جای تولید کد برای دستور A[i]=100; می تواند کد برای جمله شرطی زیر را تولید کند:

```
If(i>=0 && i<=9)
  A[i]=100;
Else
   Cout<<”out of range!”;
```

انواع کنترل معنایی

- دو دسته کنترل معنایی داریم:

۱. کنترل معنایی ایستا:

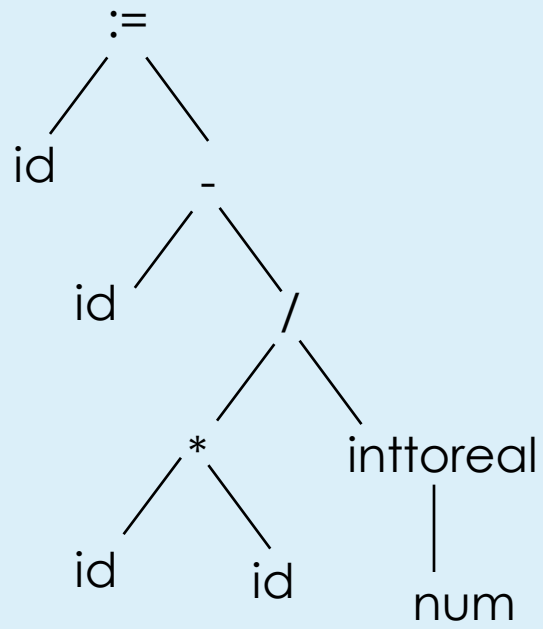
- در زمان کامپایل صورت می گیرد مثل متغیر باید قبل از استفاده تعریف شود

۲. کنترل معنایی پویا:

- در زمان اجرا صورت می گیرد. مثل مثال بالا که زده شد. یا مثل کنترل محدوده اندیس آرایه ها
- پس خطاهای معنایی می توانند از نوع خطای زمان کامپایل و یا زمان اجرا باشند.
- کنترل معنایی پویا که در زمان اجرا اعمال می شود باعث کاهش سرعت برنامه اجرایی می شود.
- در زبان هایی که کنترل نوع به صورت ایستا است ، سرعت برنامه اجرایی بالاتر، مصرف حافظه کمتر، و حجم کد کمتر و در عوض قابلیت انعطاف کمتر می شود.

تحلیل گر معنایی

- جمله s:=p-q*r/30 را از نظر لغوی و نحوی تجزیه کردیم. فرض شد که s,q,p,r ازنوع اعشاری اند .با این فرض مشاهده می شود که یکی از عملوند های عملگر دودویی تقسیم از نوع اعشاری (حاصلq*r) بوده و دیگری عدد ۳۰ از نوع صحیح است. پس به یک تبدیل نوع نیاز داریم که عدد ۳۰ را به یک عدد اعشاری تبدیل کند .



تولید کننده کد میانی

تولید کننده کد میانی

- وظیفه این مرحله تبدیل برنامه مبدا به برنامه های به زبان بینابینی (میانی) می باشد. این زبان میانی معمولا کدهای ۳ آدرسه هستند. ولی کدها می توانند تک آدرسه یا ۲ آدرسه و یا ۳ آدرسه باشند.
- انتخاب زبان میانی مناسب به عوامل زیادی از جمله کاربرد و اهمیت بهینه سازی بر می گردد. برای مثال استفاده از کدهای ۳ آدرسه امکان بهینه سازی بیشتری فراهم خواهد کرد.
- Opcode address
- Opcode address1, address 2
- Opcode address1, address 2, address3

تولید کننده کد میانی

- انتخاب هر کدام از روش های بالا به هدف طراح کامپایلر است که کدام برایش بهتر است.
- یکی از اهداف تبدیل برنامه مبدا به زبان میانی این است که برنامع میانی مستقل از زبان مبدا و مقصد باشد و هیچ وابستگی به سخت افزار و زبان ماشین نداشته باشد و باعث به وجود آمدن این امکان است که بهینه سازی راحت تر روش انجام بدیم.

(mult,q,r,t1) (inttoreal,#30,,t2) (div,t1,t2,t3) (sub,p,t3,t4) (=,t4,,s)	≡	t1=q*r t2=inttoreal(30) t3=t1/t2 t4=p-t3 s=t4

بهينه سازی کد ميانی

بهينه کردن کد مياني (حذف متغيرهاي مياني غير ضروري)

سرعت بخشيدن به توليد کد نهايي

بهینه سازی کد میانی

- در این مرحله سعی می شود روی کد میانی تکنیک های بهینه سازی اعمال شود و هدف از اعمال این تکنیک ها عبارت اند از :

۱. کاهش حافظه های موقت مصرف شده

۲. کاهش حجم کد برنامه

۳. بالا بردن سرعت اجرای برنامه

- در یک کاربرد ممکن است سرعت اجرای برنامه مهم تر از حافظه مصرفی باشد، پس طراح کامپایلر سعی خواهد کرد که بهینه سازی هایی را روی کد اعمال کند که در این راستا باشد. گاهی نیز ممکن است بهینه سازی ها با هم برخورد داشته باشد یعنی اعمال یک تکنیک باعث افزایش سرعت باشد ولی به جای کاهش حجم کد، حجم کد را افزایش دهد. به هر حال باید یک trade off ای این بین وجود داشته باشد و در راستای هدف بهینه سازی انجام گیرد.

بهینه سازی کد میانی

- از انجایی که کد میانی مستقل از زبان مبدا و مقصد و سخت افزار است پس مجموعه تکنیک های بهینه سازی که روی کد میانی اعمال می شوند، روی ویژگی مهم قابلیت حمل کامپایلر اثر منفی نخواهد داشت.

```
A=i*j;

B=i*j+i*j;

C=A[i*j];
```

- مقدار i*j حدودا ۴ بار محاسبه می شود ولی می توان یک بار آن را محاسبه کرد و هر کجا نیاز بود از مقدار محاسبه شده آن استفاده کرد.این تبدیل بهینه سازی تحت عنوان حذف زیر عبارت مشترک بوده و می تواند روی کد میانی اعمال شود. اعمال این تبدیل باعث می شود حجم کد کاهش یافته و سرعت اجرا نیز افزایش یابد.

بهینه سازی کد میانی

- در کد میانی عبارت s:=p-q/30 می توان به جای اینکه ابتدا عدد ۳۰ را به اعشاری تبدیل کرده و درادامه از آن استفاده کنیم، در خود کد به جای ۳۰ صحیح از ۳۰ اعشاری استفاده کنیم.همچنین تعداد متغییر های موقتی استفاده شده را نیز می توان کاهش داد.

```
(mult,q,r,t1)
(div,t1,#30.0,t1)
(sub,p,t1,s)
```

تولید کننده کد نهایی

تبدیل کد میاني بهینه به کد جابجاپذیر یا اسمبلي

تعیین مکانهاي حافظه براي متغیرهاي برنامه

انتساب متغیرها به ثبات هاي ماشین

تولید کننده کد نهایی

- وظیفه این مرحله تولید برنامه نهایی معادل برنامه میانی (بهینه شده) می باشد. برنامه نهایی می تواند به صورت کد اسمبلی یا کد ماشین جا به جا پذیر باشد. تولید کننده کد نهایی دستورالعمل های مناسبی را انتخاب می کند که به زبان ماشین نزدیک باشد. آدرس های حافظه مورد نیاز متغییرها را تخصیص داده و ثبات های مورد نیاز برای محاسبات میانی را تعیین می کند.

```
Mov     R1,q
Mult    R1,r
Div     R1,#30.0
Mov     R2,p
Sub     R2,R1
MOV     s,R2
```

بهینه سازی کد نهایی

- وظیفه این مرحله اعمال تکنیک های بهینه سازی روی کد نهایی است. این تکنیک ها به زبان نهایی یا مقصد وابسته اند. معمولا این تکنیک ها روی تکه تکه های کد نهایی اعمال شده و جهت اعمالشان نیاز به کل کد نهایی نیست. در صورتی که برای اعمال تکنیک های بهینه سازی روی کد میانی معمولا نیاز به کل کد میانی بود.

بخش بندی کامپایلر

- به ۴ مرحله اول و بخش عمده از مرحله بهینه سازی کد میانی که مستقل از زبان ماشین و سخت افزار است و وابستگی به زبان مبدا دارد به صورت یک مرحله در نظر گرفته می شود که به آن front endهم می گویند و به بخش باقی مانده بهینه سازی کد میانی و تولید کد نهایی و بهینه سازی کد نهایی بخش backend گفته می شود که از زبان مبدا مستقل بوده و وابسته به زبان ماشین و سخت افزار است.

بخش بندی کامپایلر

front end(گروه فازهاي متوالي وابسته به زبان مبدا)

۱- تحليل لغوي
۲- تحليل نحوي
۳- تحليل معنايي

backend(گروه فازهاي متولي وابسته به زبان مقصد)

۴- توليد كد مياني
۵- بهينه سازي كد
۶- توليد كد نهايي

جدول نمادها

- ساختمان داده ای است که در طی کامپایل برای نگهداری اطلاعات راجع به اسامی که در برنامه نویسی به کار رفته استفاده می شود. ساده ترین ساختمان داده برای نگهداری این اطلاعات یک آرایه ۲ بعدی است.
- این مشخصات می تواند شامل اسم، آدرس حافظه تخصیص داده شده به آن، محلی از برنامه که متغییر در آن تعریف شده است. خطوطی از برنامه که متغییر در آن استفاده شده است. نوع متغییر. و در مورد تابع ها نوع ورودی و تعداد ورودی و نوع خروجی و ... می تواند باشد.
- در مرحله تحلیل لغوی کلیه شناسه های موجود در برنامه ورودی وارد جدول نمادها می شوند.
- در مراحل دیگر کامپایل، اطلاعات مربوط به این شناسه ها به جدول اضافه شده و در مراحل کامپایل هر جایی که نیاز باشد از این اطلاعات استفاده خواهد شد.

خطا پرداز

- انوع خطاها:

۱. لغوی(املایی)

۲. نحوی

۳. معنایی

۴. منطقی

- **نکات:**

- **خطاهای لغوی تماما در زمان کامپایل توسط اسکنر شناسایی می شود.**
- **خطاهای نحوی تماما در زمان کامپایل توسط تجزیه گر شناسایی می شوند.**
- **خطاهای معنایی می توانند در زمان کامپایل یا زمان اجرا شناسایی شوند.**
- **خطاهای منطقی در زمان اجرا شناسایی می شوند.**

خطاپرداز

- خطاهای منطقی در زمان اجرا رخ می دهند. مثال هایی از خطاهای منطقی می تواند تقسیم بر صفر سر ریزی ایجاد حلقه نامحدود، فراخوانی توابع بازگشتی که شرط خاتمه ندارد و ارجاع به حافظه از طریق ادرس پوچ باشد.
- **اگر از تقسیم بندی بخش های پسین و پیشین استفاده نکنیم برای تولید کامپایلر های nزبان و m معماری ماشین مقصد متفاوت نیاز به طراحی و ساخت چند کامپایلر مجزا است؟؟؟؟؟**
- **N*m**
- **با توجه به تقسیم بندی فوق، اگر n زبان برنامه نویسی متفاوت و m معماری خاص داشته باشیم. تعداد کامپایلر هایی که طراحی و پیاده سازی می شوند چقدر است؟**
- **N+m**

مثال:

Area:= Pos + Rate * 50