

به نام خدا

طراحی کامپایلر

آرش شفیعی



- کامپایلرها: اصول، روش‌ها، و ابزارها از آهو و همکاران¹
- طراحی کامپایلرهای مدرن از گرون و همکاران²

¹ Compilers: Principles, Techniques, and Tools, by Aho et al.

² Modern Compiler Design, by Grune et al.

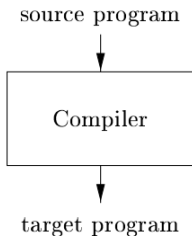
مقدمه

- زبان برنامه‌نویسی یک روش نشانه‌گذاری¹ برای توصیف محاسبات است.
- برای این که محاسبات توصیف شده توسط یک ماشین قابل فهم و اجرا باشد، محاسبات توصیف شده توسط یک زبان برنامه‌نویسی باید به زبان ماشین ترجمه شود.
- برنامه‌ای که محاسبات توصیف شده در یک زبان برنامه‌نویسی را به زبان ماشین ترجمه می‌کند کامپایلر² نامیده می‌شود.
- در مطالعه و پیاده‌سازی کامپایلرها بسیاری از مباحث زبان‌های برنامه‌نویسی، معماری ماشین‌ها، نظریه زبان‌ها و ماشین‌ها، الگوریتم‌ها و مهندسی نرم‌افزار مورد استفاده قرار می‌گیرند.
- در این فصل به معرفی اجمالی کامپایلرها و ساختار کلی آنها می‌پردازیم.

¹ notation

² compile

- کامپایلر برنامه‌ای است که یک برنامه در یک زبان برنامه‌نویسی را که زبان مبدأ¹ نامیده می‌شود، دریافت می‌کند و آن را به معادل آن در یک زبان دیگر که زبان مقصد² نامیده می‌شود ترجمه می‌کند.

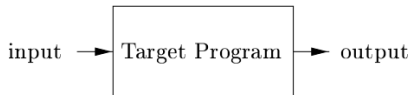


- کامپایلر همچنین خطاهایی را که در برنامه مبدأ وجود دارند را در حین ترجمه تشخیص می‌دهد.

¹ source language

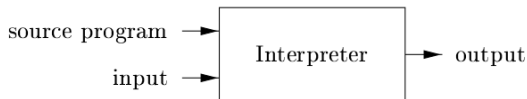
² target language

- اگر زبان مقصد برنامه‌ای به زبان ماشین باشد، آنگاه کاربر می‌تواند ورودی خود را جهت انجام محاسبات به برنامه مقصد ارسال کند و نتیجه محاسبات را دریافت کند.



پردازشگرهای زبان

- یک مفسر¹ نوع دیگری از پردازشگرهای زبان است. یک مفسر به جای تولید برنامه در زبان مقصد، مستقیماً برنامه مبدأ و ورودی کاربر را دریافت می‌کند و نتیجه محاسبات را به عنوان خروجی تولید می‌کند.



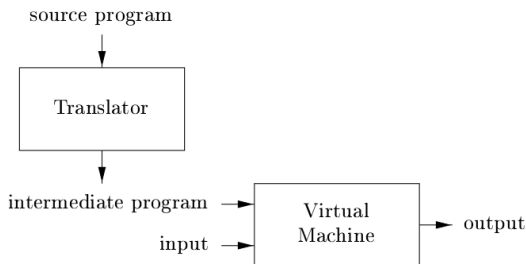
- برنامه تولید شده در زبان ماشین توسط کامپایلر معمولاً سریع‌تر از مفسر محاسبات را انجام می‌دهد.
- از طرف دیگر مزیت مفسر این است که کد نوشته شده بر روی هر ماشینی قابل اجراست و همچنین بهتر از کامپایلر می‌تواند به تشخیص خطاها توسط برنامه نویس کمک کند زیرا دستورات در آن یک‌به‌یک اجرا می‌شوند.

¹ interpreter

- مشکل اصلی پیاده‌سازی یک زبان توسط کامپایلر این است که کد تولید شده تنها بر روی ماشینی که برای آن کامپایل شده است قابل اجرا است. مزیت آن این است که کد کامپایل شده سرعت اجرای بالایی دارد. از طرف دیگر مشکل اصلی مفسر سرعت اجرای بالای آن است و مزیت آن این است که کد نوشته شده بر روی هر ماشینی قابل اجراست.

پردازشگرهای زبان

- پردازشگرهای زبان جاوا کامپایل و تفسیر را ترکیب می‌کنند. یک برنامه جاوا ابتدا به یک برنامه میانی به نام بایت‌کد¹ کامپایل می‌شود. سپس بایت‌کد تولید شده توسط یک مفسر به نام ماشین مجازی جاوا² تفسیر می‌شود.



¹ bytecode

² Java virtual machine

- مزیت این روش این است که بایت‌کد کامپایل شده بر روی یک ماشین می‌تواند بر روی یک ماشین دیگر تفسیر شود و در عین حال سرعت اجرای آن نیز بهتر از روش تفسیر است.
- از آنجایی که اجرای برنامه توسط مفسر نسبت به اجرای برنامه به زبان ماشین سرعت پایین‌تری دارد، کامپایلرهایی به نام کامپایلرهای درجا¹ وجود دارند که بایت‌کد را به کد زبان ماشین ترجمه می‌کنند.

¹ just-in-time

- برای تولید برنامه قابل اجرا در زبان مقصد، علاوه بر یک کامپایلر، به برنامه‌های دیگری نیز نیاز است.
- یک برنامه مبدأ می‌تواند به ماژول‌های مختلف در چند فایل جداگانه ذخیره شده باشد. وظیفه جمع‌آوری برنامه مبدأ از فایل‌های مختلف به عهده پیش‌پردازنده¹ است. همچنین یک پیش‌پردازنده می‌تواند دستوراتی را که به عنوان مخفف به جای دستورات اصلی به کار رفته‌اند (ماکروها²) را با دستورات اصلی جایگزین کند.
- پس از پردازش پیش‌پردازنده، برنامه تبدیل شده به کامپایلر ارسال می‌شود. کامپایلر، یک برنامه به زبان اسمبلی³ به عنوان خروجی تولید می‌کند، زیرا تولید کد اسمبلی ساده‌تر از تولید کد به زبان ماشین است.
- برنامه اسمبلی به یک اسمبلر⁴ داده می‌شود و کد برنامه به زبان ماشین تولید می‌شود.

¹ preprocessor

² macros

³ assembly

⁴ assembler

- برنامه‌های بزرگ معمولاً به صورت قطعات جداگانه کامپایل می‌شوند. به همین جهت نیاز است قطعات کامپایل شده با یکدیگر پیوند داده شوند. پیوند برنامه‌ها توسط پیوند دهنده یا لینکر¹ انجام می‌شود.
- در پایان برنامه اجرایی تولید شده توسط لینکر به وسیله لودر² در حافظه قرار می‌گیرد.

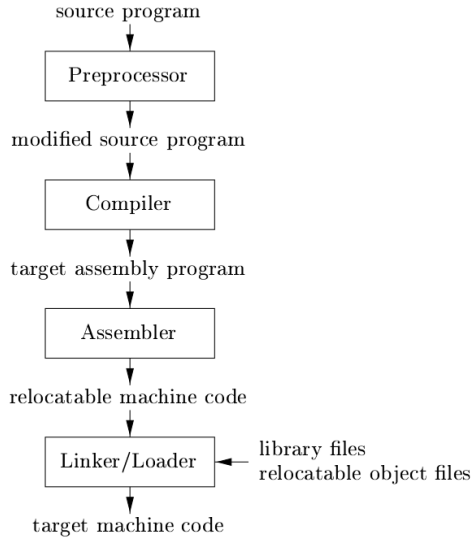
¹ linker

² loader

- در صورتی که یک کتابخانه ایستا¹ باشد، لینکر آن را به برنامه پیوند می‌دهد و کد اجرایی تولید شده از الحاق برنامه کامپایل شده و کتابخانه کامپایل شده به دست می‌آید. در صورتی که کتابخانه پویا² باشد، لودر در زمان اجرای برنامه آن را در حافظه قرار می‌دهد.
- مزیت کتابخانه ایستا این است که کد برنامه می‌تواند به صورت مستقل اجرا شود و عیب آن این است که کد اجرایی تولید شده حجم بیشتری اشغال می‌کند. مزیت کتابخانه پویا این است که اگر کتابخانه به طور مجزا به روز رسانی شود، نیاز به کامپایل مجدد برنامه وجود ندارد. همچنین کد کامپایل شده حجم کمتری اشغال می‌کند. همچنین کتابخانه‌های پویا را می‌توان بین چند برنامه به اشتراک گذاشت.

¹ static library

² dynamic library



- ساختار داخلی کامپایلر را می‌توان به دو بخش تحلیل یا آنالیز¹ و بخش ترکیب یا سنتز² تقسیم کرد.

- بخش تحلیل برنامه مبدأ را بر اساس گرامر آن ساختار بندی و تقسیم بندی می‌کند. سپس از این تحلیل برای تولید یک کد میانی معادل برنامه مبدأ استفاده می‌کند. اگر در تحلیل کد مبدأ، کامپایلر یک خطای نحوی³ یا معنایی⁴ تشخیص دهد، باید پیام‌های خطای مناسب تولید کند تا کاربر برنامه تصحیح کند. در بخش تحلیل کامپایلر، یک جدول علائم⁵ نیز تهیه می‌شود که ساختار داده‌ای برای نگهداری اطلاعات درمورد برنامه است. این جدول علائم نیز به بخش سنتز کامپایلر ارسال می‌شود.

¹ analysis

² synthesis

³ syntax

⁴ semantic

⁵ symbol table

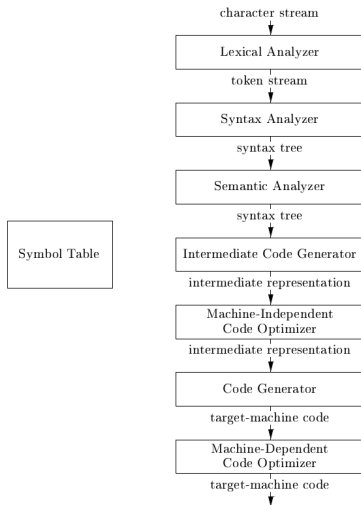
- بخش سنتز با استفاده از اطلاعات دریافت شده از بخش تحلیل، یعنی نمایش میانی ساختار برنامه¹ (کد میانی) و جدول علائم برنامه مقصد را تولید می‌کند.
- بخش تحلیل یا آنالیز بخش پیشین² کامپایلر و بخش ترکیب یا سنتز بخش پسین³ کامپایلر نیز نامیده می‌شود.
- یک کامپایلر واحدی به نام بهینه‌ساز کد را نیز شامل می‌شود که ساختار دریافت شده از بخش تحلیل را دریافت کرده و آن را بهینه می‌کند تا کد مقصد بهتری تولید شود.

¹ intermediate representation

² front end

³ back end

- یک کامپایلر از قسمت‌های زیر تشکیل شده است. جدول علائم با همه قسمت‌ها در ارتباط است.



- اولین مرحله در فرایند کامپایل تحلیل لغوی یا واژگانی¹ یا اسکنر² نامیده می‌شود.
- تحلیل‌گر لغوی جریانی از کاراکترها را که برنامه مبدأ را تشکیل می‌دهند دریافت می‌کند و آنها را به دنباله‌های معناداری به نام واژه³ تبدیل می‌کند.
- به ازای هر واژه، تحلیل‌گر لغوی، یک توکن⁴ به صورت یک زوج شامل نام توکن⁵ و مقدار ویژگی⁶ آن تولید می‌کند. `<token-name, attribute-value>`
- نام توکن در واقع مفهوم انتزاعی یا نوع واژه است و ویژگی توکن می‌تواند خود واژه باشد و یا به مکان واژه در جدول علائم اشاره کند. توکن‌ها به واحد بعدی کامپایلر یعنی تحلیل‌گر نحوی ارسال می‌شوند.

¹ lexical analysis

² scanner

³ lexeme

⁴ token

⁵ token name

⁶ attribute value

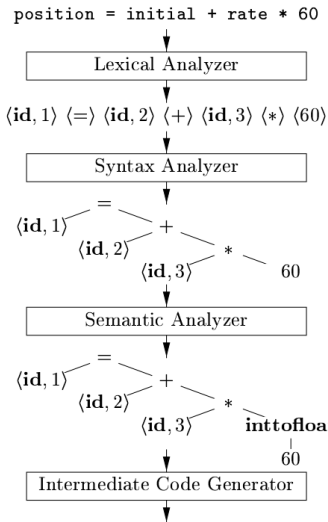
- برای مثال فرض کنید برنامه مبدأ یک دستور انتساب مقدار به صورت زیر باشد.
$$\text{position} = \text{initial} + \text{rate} * 60$$
- حروف یا کاراکترهای این دستور را می‌توان دسته‌بندی کرد و واژه‌های زیر را تولید کرد.
- ۱- واژه `position` به توکن $\langle \text{id}, 1 \rangle$ نگاشت می‌شود، جایی که `id` نوعی انتزاعی برای مفهوم شناسه¹ است و 1 به مکان واژه `position` در جدول علائم اشاره می‌کند. در جدول علائم در مکان واژه `position`، نوع و نام این متغیر ذخیره شده است.
- ۲- نماد انتساب `=` واژه‌ای است که به توکن $\langle = \rangle$ نگاشت می‌شود. این توکن هیچ مقدار ویژگی ندارد. می‌توانستیم به جای $\langle = \rangle$ از توکن $\langle \text{assign} \rangle$ نیز استفاده کنیم ولی برای سهولت نشانه‌گذاری از توکن $\langle = \rangle$ استفاده می‌کنیم.
- ۳- واژه `initial` به توکن $\langle \text{id}, 2 \rangle$ نگاشت می‌شود.

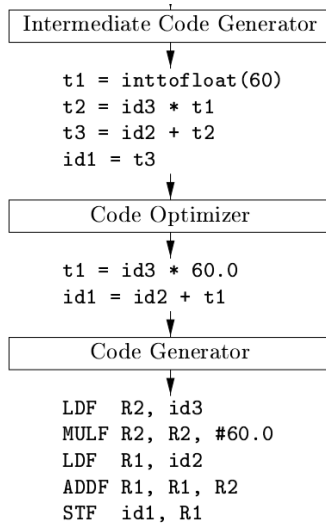
¹ identifier

- ۴- نماد جمع + به توکن $\langle + \rangle$ نگاشت می‌شود.
- ۵- واژه rate به توکن $\langle id, 3 \rangle$ نگاشت می‌شود.
- ۶- نماد ضرب * واژه‌ای است که به توکن $\langle * \rangle$ نگاشت می‌شود.
- ۷- عدد 60 واژه‌ای است که به توکن $\langle 60 \rangle$ نگاشت می‌شود.
- از فاصله‌های خالی بین واژگان در تحلیل‌گر واژگانی چشم پوشی می‌شود.
- بنابراین دستور قبلی پس از تحلیل واژگانی به صورت زیر تبدیل می‌شود.
 $\langle 60 \rangle \langle * \rangle \langle id, 3 \rangle \langle + \rangle \langle id, 2 \rangle \langle = \rangle \langle id, 1 \rangle$

1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE





- دومین گام کامپایلر تحلیل نحوی¹ یا تجزیه² است. تجزیه کننده³ از توکن‌های تهیه شده توسط تحلیل‌گر لغوی استفاده می‌کند و یک نمایش درختی از ساختار گرامری جریان یا رشته ورودی می‌سازد.
- به ساختار تولید شده توسط تحلیل‌گر نحوی، درخت تجزیه⁴ گفته می‌شود.
- گاهی کلمه درخت نحوی⁵ به عنوان مترادف درخت تجزیه به کار برده می‌شود، با اینکه تفاوت‌های اندکی بین درخت تجزیه و درخت نحوی وجود دارد. در درخت تجزیه هنوز معنای یک برنامه بررسی نشده است، اما درخت نحوی پس از تحلیل معنایی ساخته می‌شود.

¹ syntax analysis

² parsing

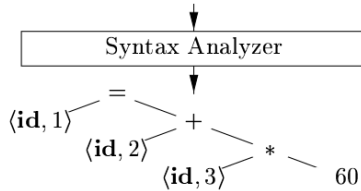
³ parser

⁴ parse tree

⁵ syntax tree

- درخت تجزیه برای عبارت $\text{position} = \text{initial} + \text{rate} * 60$ در زیر نمایش داده شده است. در این درخت تجزیه ترتیب اعمال عملگرها تعیین می‌شود.

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$



- توسط این درخت، عملگر ضرب، سپس عملگر جمع و در پایان عملگر انتساب ارزیابی می‌شوند.

- تحلیل‌گر معنایی¹ درخت تجزیه را به همراه جدول علائم دریافت می‌کند و درستی معنایی برنامه مبدأ را بررسی می‌کند.
- یک قسمت مهم تحلیل معنا، بررسی نوع² است، که توسط آن کامپایلر بررسی می‌کند هر عملگر با عملوندهای آن مطابقت داشته باشند.
- تحلیل‌گر معنایی درخت تجزیه را دریافت و درخت نحوی تولید می‌کند.

¹ semantic analyzer

² type checking

- برای مثال در بسیاری از زبان‌های برنامه‌نویسی لازم است اندیس آرایه‌ها یک عدد صحیح باشد، بنابراین در صورتی که اندیس آرایه عدد صحیح نبود، کامپایلر باید پیام خطا صادر کند. این بررسی در تحلیل‌گر معنایی انجام می‌شود.
- در یک زبان ممکن است برخی از تبدیل نوع‌ها مجاز باشند. برای مثال یک عملگر حسابی دوگانی می‌تواند یک جفت عدد صحیح یا یک جفت عدد اعشاری را به عنوان عملوند دریافت کند. اگر یک عملوند دوگانی یک عدد صحیح و یک عدد اعشاری دریافت کرد، کامپایلر ممکن است عدد صحیح را به اعشاری تبدیل کند. این عملیات نیز در تحلیل‌گر معنایی انجام می‌شود.
- در مثال قبل `rate` یک عدد اعشاری است بنابراین عملوند دوم عملگر ضرب یعنی عدد 60 توسط کامپایلر به یک عدد اعشاری تبدیل می‌شود. برای این تبدیل که به صورت ضمنی انجام می‌شود از عملگر `inttofloat` استفاده شده است.

- در فرایند ترجمه یک برنامه مبدأ به یک برنامه مقصد، کامپایلر ممکن است یک یا چند نمایش از کد مبدأ تولید کند.
- برای مثال درخت نحوی یک نمایش میانی از کد مبدأ است که در تحلیل معنایی استفاده می‌شود.
- پس از تحلیل معنایی، معمولاً کامپایلرها یک نمایش میانی شبیه به کد ماشین تولید می‌کنند که همان برنامه در یک زبان میانی است. این کد میانی باید دو ویژگی داشته باشد. اول اینکه این کد میانی باید به آسانی توسط برنامه مبدأ قابل تولید باشد و همچنین این کد میانی باید بتواند به سهولت به کد مقصد تبدیل شود.

تولید کد میانی

- برای مثال یک کد میانی به نام کد سه آدرسی¹ که شبیه کد اسمبلی است، در مرحله تولید کد میانی تولید می‌شود.

- کد میانی برای دستور قبل به صورت زیر است :

```
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

- در این کد میانی در سمت راست عبارت انتساب حداکثر یک عملگر وجود دارد، بنابراین ترتیب اعمال عملگرها مشخص است. همچنین برای مقادیر میانی نام‌های موقت تعریف می‌شود.

- هر عبارت در این کد میانی حداکثر سه عملوند دارد و به همین دلیل به آن کد سه آدرسی گفته می‌شود.

¹ three-address code

- مرحله بهینه‌سازی مستقل از ماشین کد¹، کد میانی تولید شده را بهینه‌سازی می‌کند تا کد بهتری در پایان برای ماشین مقصد تولید شود. کد بهتر معمولاً کدی است که سریع‌تر اجرا می‌شود، اما می‌توانیم از معیارهای دیگری نیز برای ارزیابی کد بهتر استفاده کنیم. برای مثال معیار ارزیابی می‌تواند کوتاه‌تر بودن کد یا میزان مصرف انرژی کد باشد.

¹ machine-independent code-optimization

- در مثال قبل بهینه‌ساز کد مستقیماً عدد صحیح 60 را به عدد اعشاری 60.0 تبدیل می‌کند و دستور `inttofloat` را حذف می‌کند. علاوه بر آن عبارت انتساب `id1 = t3` تنها یک مقدار را منتقل می‌کند، بنابراین بهینه‌ساز کد متغیر `t3` را حذف می‌کند.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



Code Optimizer



```
t1 = id3 * 60.0
id1 = id2 + t1
```

- برخی از روش‌های بهینه‌سازی کد وجود دارند که با کاستن سرعت کامپایل به میزان اندکی، کد مقصد را به میزان قابل توجهی بهبود می‌دهند.
- برخی دیگر از روش‌های بهینه‌سازی پیچیده‌تر نیز وجود دارند که به زمان بیشتری برای اجرا نیاز دارند که در کامپایلرهای بهینه‌ساز استفاده می‌شوند.

- در مرحله تولید کد¹، کد میانی معادل برنامه ورودی دریافت می‌شود و به یک برنامه در زبان مقصد تبدیل می‌شود. اگر زبان مقصد، زبان ماشین باشد، آنگاه از رجیسترها و قابلیت‌های ماشین مقصد برای تبدیل کد استفاده می‌شود. در مرحله تولید کد، استفاده بهینه از رجیستر اهمیت زیادی پیدا می‌کند.

¹ code generation

- برای مثال با استفاده از رجیسترهای R1 و R2 در مثال قبل کد زیر تولید می‌شود.

```
t1 = id3 * 60.0
id1 = id2 + t1
```



Code Generator



```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

- حرف F در این دستورات نشان می‌دهد که محاسبات بر روی اعداد اعشاری انجام می‌شود. دستور LD برای کپی کردن مقدار از حافظه در رجیستر و ST برای ذخیره کردن مقدار یک رجیستر در یک فضای حافظه استفاده شده است. دستور ADD برای جمع مقدار دو رجیستر و دستور MUL برای ضرب مقدار دو رجیستر استفاده شده است.

- یکی از وظایف کامپایلر نگهداری جدول علائم¹ برای ذخیره‌سازی نام متغیرها و نوع ویژگی‌های دیگر آنهاست که در طول برنامه از آن استفاده می‌شود.
- این ویژگی‌ها اطلاعاتی را نگهداری می‌کنند که همهٔ بخش‌های کامپایلر از آنها استفاده می‌کنند. برای مثال برای یک متغیر، نام، نوع، و حوزه تعریف متغیر اطلاعات مهمی هستند و برای یک تابع، نام، تعداد و نوع پارامترها و نوع فراخوانی پارامترها (فراخوانی با مقدار یا ارجاع یا غیره) و نوع دادهٔ بازگردانده شده از تابع اطلاعاتی هستند که می‌توانند در جدول علائم ذخیره شوند.

¹ symbol table

- مراحل مختلف و اجزای متفاوت یک کامپایلر را می‌توان به دسته‌هایی تقسیم کرد. برای مثال اجزای مقدماتی از جمله تحلیل‌گر لغوی، تحلیل‌گر نحوی، تحلیل‌گر معنایی و تولیدکننده کد میانی را می‌توان در یک گروه قرار داد و بهینه‌ساز کد و تولیدکننده کد را در یک گروه دیگر. بدین ترتیب دو قسمت سمت کاربر (بخش پیشین)¹ و سمت ماشین (بخش پسین)² به وجود می‌آیند. قسمت سمت ماشین بسته به زبان ماشین مقصد متفاوت خواهد بود و قسمت‌های سمت ماشین متفاوتی به ازای ماشین‌های متفاوت وجود خواهند داشت. اما قسمت سمت کاربر به ازای یک زبان ورودی همیشه یکسان است.
- با جداسازی این دو قسمت می‌توان کامپایلری عرضه کرد که قسمت پسین آن برای ماشین‌های جدید قابل تعویض باشد.

¹ front-end

² back-end

- سازندگان کامپایلر مانند همه توسعه‌دهندگان نرم‌افزار از ابزارهای متفاوتی برای تهیه کامپایلر استفاده می‌کنند. برخی از این کامپایلرها به شرح زیر می‌باشند.

۱. تولیدکننده تحلیل‌گر لغوی¹ به صورت خودکار با استفاده از یک توصیف از توکن‌های زبان در زبان منظم، تحلیل‌گر لغوی تهیه می‌کند.
۲. تولیدکننده تجزیه‌کننده² به صورت خودکار یک تحلیل‌گر نحوی از یک توصیف گرامری تهیه می‌کند.
۳. مترجم نحوی³ با استفاده از یک توصیف معنایی از قوانین گرامر، و مجموعه‌ای از قواعد برای پیمایش درخت تجزیه، کد میانی تهیه می‌کند.

¹ scanner generator

² parser generator

³ syntax-directed translator

۴. تولید کنندهٔ تولید کننده کد⁴ : با استفاده از مجموعه‌ای از قوانین ترجمه دستورات کد میانی به کد ماشین، یک تولید کننده کد تهیه می‌کند.
۵. موتور تحلیل جریان داده⁵ : تحلیل جریان داده یکی از بخش‌های مهم بهینه‌ساز کد است. موتور تحلیل جریان داده با جمع‌آوری اطلاعات در مورد اینکه بخش‌های مختلف برنامه چگونه با یکدیگر ارتباط برقرار می‌کنند، کد تولید شده را بهینه‌سازی می‌کند.
۶. ابزار تولید کامپایلر⁶ : مجموعه‌ای از توابع و ابزارها برای ساخت کامپایلر ارائه می‌کند.

⁴ code-generator generator

⁵ data-flow analyzer

⁶ compiler-construction toolkit

- کامپیوترهای الکترونیکی ابتدایی در دههٔ ۱۹۴۰ به وجود آمدند. برنامه‌های آنها به زبان ماشین یعنی با استفاده از صفر و یک‌ها توصیف می‌شدند. با استفاده از ورودی‌های دودویی به کامپیوتر دستور داده می‌شد که عملیات خواندن از حافظه، جمع، ضرب، نوشتن بر روی حافظه و غیره را انجام دهد. نیاز به گفتن نیست که این نوع برنامه‌نویسی بسیار پیچیده است و احتمال ایجاد خطا در آن زیاد است. همچنین هنگامی که برنامه‌ای تولید شد، خواندن و فهمیدن و تغییر دادن آن بسیار مشکل است.

- اولین گام در طراحی زبان‌هایی که به زبان انسان شباهت بیشتری داشته باشند تا زبان ماشین، در طراحی زبان اسمبلی در اوایل دههٔ ۱۹۵۰ صورت گرفت. در زبان اسمبلی دستورات زبان ماشین که با صفر و یک بیان می‌شوند، توسط کلمات معنی‌دار زبان انگلیسی جایگزین شدند.
- در اواخر دهه ۱۹۵۰ زبان فورترن^۱ برای انجام محاسبات علمی، زبان کوپول^۲ برای انجام پردازش داده در کاربردها تجاری، و زبان لیسپ^۳ برای انجام محاسبات بر روی لیست‌ها و کاربردهای هوش مصنوعی توسعه یافتند. دلیل ابداع این زبان‌ها، ایجاد امکان توصیف برنامه‌ها به زبانی بود که به زبان انسان‌ها و زبان ریاضی شباهت بیشتری داشته باشد. برای مثال زبان فورترن برای تبدیل فرمول‌های ریاضی به زبان اسمبلی وجود آمد تا با استفاده از آن برنامه‌نویسان بتوانند فرمول‌ها را شبیه به توصیف ریاضی آنها در برنامه توصیف کنند.

^۱ Fortran

^۲ Cobol

^۳ Lisp

- زبان‌های برنامه‌نویسی را می‌توان براساس رویکرد آنها به توصیف برنامه‌ها دسته‌بندی کرد. یک دسته از زبان‌های برنامه‌نویسی که زبان‌های دستوری¹ نامیده می‌شوند، در توصیف یک برنامه به چگونگی نحوه اجرای برنامه می‌پردازند. به عبارت دیگر یک برنامه به زبان دستوری توصیف می‌کند چگونه محاسبات انجام می‌شوند. دسته‌ای دیگر از زبان‌ها، زبان‌های اعلامی² نامیده می‌شوند. یک برنامه به زبان اعلامی مشخص می‌کند چه محاسباتی باید انجام شود و چگونگی انجام محاسبات غالباً بر عهده کامپایلر است.
- زبان‌های سی، سی++ و جاوا در دسته زبان‌های دستوری قرار می‌گیرند و زبان‌های لیسپ، ام‌ال، هسکل و پرولوگ در دسته زبان‌های اعلامی.

¹ imperative

² declarative

- همچنین می‌توان زبان دستوری را به دو دسته زبان‌های رویه‌ای¹ و زبان‌های شیء‌گرا² تقسیم کرد. یک برنامه در یک زبان رویه‌ای شامل تعداد تابع است که مقادیر خروجی خود را به یکدیگر منتقل می‌کنند و یک برنامه در یک زبان شیء‌گرا از تعدادی شیء تشکیل شده که هر کدام ویژگی و رفتارهایی دارند و از طریق رفتارها با یکدیگر در ارتباط‌اند. سی یک زبان رویه‌ای و سی++ و جاوا مثال‌هایی از زبان‌های شیء‌گرا هستند.
- زبان‌های اعلامی را نیز می‌توان به چند دسته تقسیم کرد. زبان‌های تابعی³ زبان‌هایی هستند که یک برنامه را با استفاده از تعدادی تابع همانند توابع ریاضی توصیف می‌کنند و زبان‌های منطقی⁴ زبان‌هایی هستند که یک برنامه را با استفاده از تعدادی گزاره منطقی توصیف می‌کنند. لیست و اسکیم و هسکل و ام‌ال در دسته زبان‌های تابعی، و پرولوگ در دسته زبان‌های منطقی قرار می‌گیرند.

¹ procedural

² object-oriented

³ functional

⁴ logical

- طراحان کامپایلر باید از طرفی با ویژگی‌های زبان‌های جدید و نیازهای جدید زبان‌های برنامه‌نویسی آشنا باشند و از طرفی دیگر قابلیت‌های ماشین‌ها و سخت‌افزارهای جدید را بشناسند.
- موضوع علم کامپایلر این است که چگونه الگوریتم‌های مناسب را برای پیاده‌سازی یک زبان به کار ببریم به طوری که از طرفی الگوریتم‌های طراحی شده عمومی باشند و همه برنامه‌های ورودی را در بر بگیرد و از طرفی کارامدی بالایی داشته باشد. بنابراین به علم طراحی الگوریتم و ساختمان داده و معماری ماشین نیاز است.
- همچنین از علم نظریه زبان‌های و ماشین‌ها در کامپایلر استفاده می‌شود، چراکه ماشین‌های منظم و گرامرهای منظم برای توصیف واژگان زبان و گرامرهای مستقل از متن برای توصیف ساختار زبان می‌توانند مورد استفاده قرار گیرند.