به نام خدا

طراحی کامپایلر

آرش شفیعی

کتاب‌های مرجع

- کامپایلرها: اصول، روش‌ها، و ابزارها از آهو و همکاران [1]
- طراحی کامپایلر مدرن از گرون و همکاران [2]
- پیاده‌سازی کامپایلر مدرن در جاوا از اپل [3]

[1] Compilers: Principles, Techniques, and Tools, by Aho et al.
[2] Modern Compiler Design, by Grune et al.
[3] Modern Compiler Implementation in Java, by Appel

مقدمه

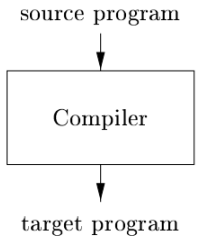
مقدمه

- زبان برنامه‌نویسی یک روش نشانه‌گذاری [1] 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 کامپایلر [2] نامیده می‌شود.
- در مطالعه و پیاده‌سازی کامپایلرها بسیاری از مباحث زبان‌های برنامه‌نویسی، معماری ماشین‌ها، نظریهٔ 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

پردازشگرهای زبان

- کامپایلر برنامه‌ای است که یک برنامه در یک زبان برنامه‌نویسی را که زبان مبدأ [1] نامیده می‌شود، دریافت می‌کند و آن را به معادل آن در یک زبان دیگر که زبان مقصد [2] نامیده می‌شود ترجمه می‌کند.

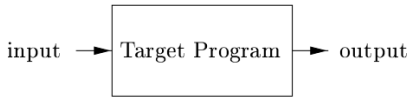


- کامپایلر همچنین خطاهایی را که در برنامه مبدأ وجود دارند را در حین ترجمه تشخیص می‌دهد.

[1] source language
[2] target language

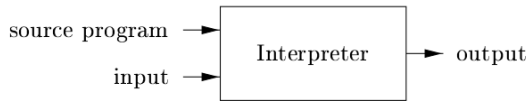
پردازشگرهای زبان

- وقتی 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 مفسر [1] 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.

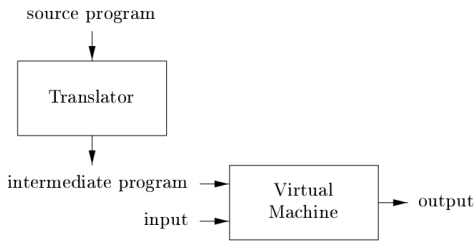
[1] interpreter

پردازشگرهای زبان

- 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 پایین آن است و مزیت آن این است که کد نوشته شده بر روی هر ماشینی قابل اجراست.

پردازشگرهای زبان

- از آنجایی که کامپایل و تفسیر هر کدام مزیت‌های خاص خود را دارند، برخی از پردازشگرهای زبان از یک روش ترکیبی استفاده می‌کنند.
- پردازشگرهای زبان جاوا کامپایل و تفسیر را ترکیب می‌کنند. یک برنامه جاوا ابتدا به یک برنامه میانی به نام بایت‌کد [1] کامپایل می‌شود. سپس بایت کد تولید شده توسط یک مفسر به نام ماشین مجازی جاوا [2] تفسیر می‌شود.



[1] bytecode
[2] Java virtual machine

پردازشگرهای زبان

- مزیت این روش این است که بایت‌کد کامپایل شده برروی یک ماشین می‌تواند برروی یک ماشین دیگر تفسیر شود و در عین حال سرعت اجرای آن نیز بهتر از روش تفسیر است.
- از آنجایی که اجرای برنامه توسط پردازشگرهای ترکیبی زبان نسبت به اجرای برنامه به زبان ماشین سرعت پایین‌تری دارد، کامپایلرهایی به نام کامپایلرهای درجا [1] وجود دارند که بایت‌کد را به کد زبان ماشین ترجمه می‌کنند.

[1] just-in-time

پردازشگرهای زبان

- 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 عهدهٔ پیش‌پردازنده [1] است. همچنین یک پیش‌پردازنده می‌تواند دستوراتی را که به عنوان مخفف به جای دستورات اصلی به کار رفته‌اند (ماکروها [2]) را با دستورات اصلی جایگزین کند.
- پس از پردازش پیش‌پردازنده، برنامه تبدیل شده به کامپایلر ارسال می‌شود. کامپایلر، یک برنامه به زبان اسمبلی [3] به عنوان خروجی تولید می‌کند، زیرا تولید کد اسمبلی ساده‌تر از تولید کد به زبان ماشین است.
- برنامه اسمبلی به یک اسمبلر [4] داده می‌شود و کد برنامه به زبان ماشین تولید می‌شود.

---

[1] preprocessor
[2] macros
[3] assembly
[4] assembler

پردازشگرهای زبان

- 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 لینکر [1] انجام می‌شود.
- در پایان برنامه اجرایی تولید شده توسط لینکر به وسیله لودر [2] در حافظه قرار می‌گیرد.

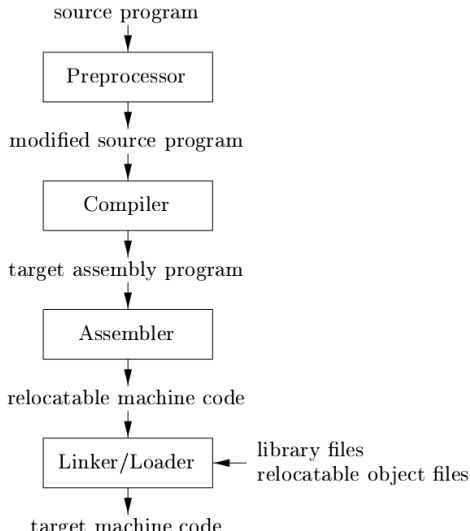
[1] linker
[2] loader

پردازشگرهای زبان

- در صورتی که یک برنامه بخواهد از یک کتابخانه ایستا [1] استفاده کند، لینکر آن را به برنامه پیوند می‌دهد و کد اجرایی تولید شده از الحاق برنامه کامپایل شده و کتابخانه کامپایل شده به دست می‌آید.
- اما در صورتی که یک برنامه بخواهد از یک کتابخانه پویا [2] استفاده کند، لودر در زمان اجرای برنامه کد ماشین کتابخانه پویا را در حافظه قرار می‌دهد.
- مزیت استفاده از 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 استفاده از 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 برنامه حجم کمتری اشغال می‌کند. همچنین کتابخانه‌های پویا را می‌توان بین چند برنامه به اشتراک گذاشت.

[1] static library
[2] dynamic library

پردازشگرهای زبان



ساختار کامپایلر

- ساختار داخلی کامپایلر را می‌توان به دو بخش تحلیل یا آنالیز [1] و بخش ترکیب یا سنتز [2] تقسیم کرد.

- بخش تحلیل برنامه مبدأ را بر اساس گرامر آن ساختاربندی و تقسیم‌بندی می‌کند. سپس از این تحلیل برای تولید یک کد میانی معادل برنامه مبدأ استفاده می‌کند. اگر در تحلیل کد مبدأ، کامپایلر یک خطای نحوی [3] یا معنایی [4] تشخیص دهد، باید پیام‌های خطای مناسب تولید کند تا کاربر، برنامه را تصحیح کند. در بخش تحلیل، یک جدول علائم [5] نیز تهیه می‌شود که ساختار داده‌ای برای نگهداری اطلاعات درمورد برنامه است. این جدول علائم نیز به بخش سنتز کامپایلر ارسال می‌شود.

- بخش سنتز با استفاده از اطلاعات دریافت شده از بخش تحلیل، یعنی نمایش میانی ساختار برنامه [6] (کد میانی) و جدول علائم، برنامه مقصد را تولید می‌کند.

---

[1] analysis
[2] synthesis
[3] syntax error
[4] semantic error
[5] symbol table
[6] intermediate representation

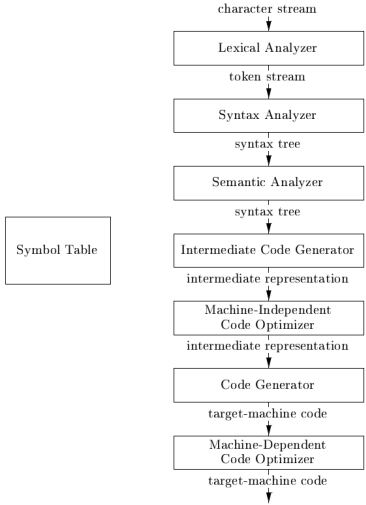
ساختار کامپایلر

- بخش تحلیل یا آنالیز، بخش روساخت [1] یا سمت کاربر کامپایلر و بخش ترکیب یا سنتز، بخش زیرساخت [2] یا سمت ماشین 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] front end
[2] back end

ساختار کامپایلر

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



تحلیل لغوی

- اولین مرحله در فرایند کامپایل تحلیل لغوی یا واژگانی [1] یا اسکنر [2] نامیده می‌شود.
- تحلیل‌گر لغوی جریانی از کاراکترها را که برنامه مبدأ را تشکیل می‌دهند دریافت می‌کند و آنها را به دنباله‌های معناداری به نام واژه [3] تبدیل می‌کند.
- به ازای هر واژه، تحلیل‌گر لغوی، یک توکن [4] به صورت یک زوج شامل نام توکن [5] و مقدار ویژگی [6] 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

تحلیل لغوی

- برای مثال فرض کنید برنامه مبدأ یک دستور انتساب مقدار به صورت زیر باشد.
  `position = initial + rate * 60`

- حروف یا کاراکترهای این دستور را می‌توان دسته‌بندی کرد و واژه‌های زیر را تولید کرد.

۱- واژهٔ `position` به توکن ⟨`id,1`⟩ نگاشت می‌شود، جایی که `id` نوعی انتزاعی برای مفهوم شناسه [1] است و 1 به مکان واژهٔ `position` در جدول علائم اشاره می‌کند. در جدول علائم در مکان واژهٔ `position` ، نوع و نام این متغیر ذخیره شده است.

۲- نماد انتساب = واژه‌ای است که به توکن ⟨=⟩ نگاشت می‌شود. این توکن هیچ مقدار ویژگی ندارد. می‌توانستیم به جای ⟨=⟩ از توکن ⟨`assign`⟩ نیز استفاده کنیم ولی برای سهولت نشانه‌گذاری از توکن ⟨=⟩ استفاده می‌کنیم.

۳- واژهٔ `initial` به توکن ⟨`id,2`⟩ نگاشت می‌شود.

[1] identifier

تحلیل لغوی

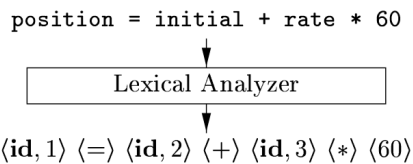
۴- نماد جمع + به توکن ⟨+⟩ نگاشت می‌شود.

۵- واژهٔ rate به توکن ⟨id,3⟩ نگاشت می‌شود.

۶- نماد ضرب * واژه‌ای است که به توکن ⟨*⟩ نگاشت می‌شود.

۷- عدد 60 واژه‌ای است که به توکن ⟨60⟩ 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.
  ⟨id,1⟩ ⟨=⟩ ⟨id,2⟩ ⟨+⟩ ⟨id,3⟩ ⟨*⟩ ⟨60⟩



تحلیل نحوی

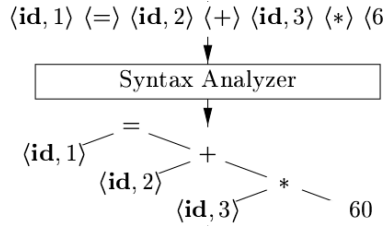
- دومین گام کامپایلر تحلیل نحوی [1] یا تجزیه [2] است. تجزیه کننده [3] از توکن‌های تهیه شده توسط تحلیل‌گر لغوی استفاده می‌کند و یک نمایش درختی از ساختار گرامری جریان یا رشتهٔ ورودی می‌سازد.
- به ساختار تولید شده توسط تحلیل‌گر نحوی، درخت تجزیه [4] گفته می‌شود.
- گاهی کلمهٔ درخت نحوی [5] به عنوان مترادف درخت تجزیه به کار برده می‌شود، با اینکه تفاوت‌های اندکی بین درخت تجزیه و درخت نحوی وجود دارد.

---

[1] syntax analysis
[2] parsing
[3] parser
[4] parse tree
[5] syntax tree

تحلیل نحوی

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



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

تحلیل معنایی

- تحلیل‌گر معنایی [1] درخت تجزیه را به همراه جدول علائم دریافت می‌کند و درستی معنایی برنامهٔ مبدأ را بررسی می‌کند.
- یک قسمت مهم تحلیل معنا، بررسی نوع [2] است، که توسط آن کامپایلر بررسی می‌کند هر عملگر با عملوندهای آن مطابقت داشته باشند.
- تحلیل‌گر معنایی درخت تجزیه یا درخت نحوی را دریافت می‌کند و بر روی آن عملیات تحلیل معنا انجام می‌دهد و یک درخت نحوی دیگر تولید می‌کند.

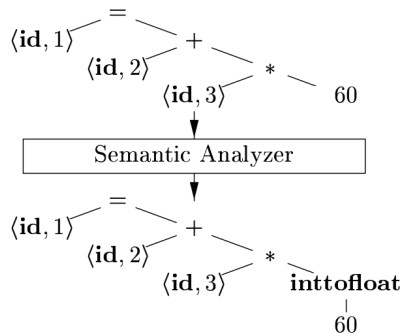
[1] semantic analyzer
[2] type checking

تحلیل معنایی

- برای مثال اگر در یک عبارت انتساب مقدار، نوع متغیرها در دو طرف عملگر همخوانی نداشته باشند، 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.

تحلیل معنایی

- 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.

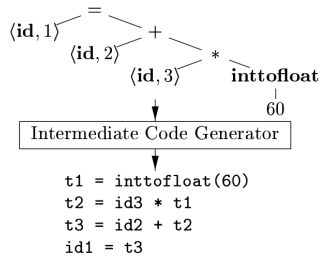


تولید کد میانی

- 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 تولید 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 کد میانی باید به آسانی توسط برنامهٔ مبدأ قابل تولید باشد و دوم اینکه کد میانی باید بتواند به سهولت به کد مقصد تبدیل شود.

تولید کد میانی

- برای مثال یک کد میانی به نام کد سه آدرسی [1] که شبیه کد اسمبلی است، در مرحله تولید کد میانی تولید می‌شود. کد میانی برای دستور قبل به صورت زیر است :



- 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

بهینه‌سازی کد

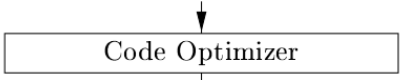
- مرحلهٔ بهینه‌سازی مستقل از کد ماشین [1] ، کد میانی تولید شده را بهینه‌سازی می‌کند تا کد بهتری در پایان برای ماشین مقصد تولید شود. کد بهتر معمولا کدی است که سریع‌تر اجرا می‌شود، اما می‌توانیم از معیارهای دیگری نیز برای ارزیابی کد بهتر استفاده کنیم. برای مثال معیار ارزیابی می‌تواند کوتاهتر بودن کد یا میزان مصرف انرژی کد باشد.

[1] machine-independent code-optimization

بهینه‌سازی کد

- 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
```




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

بهینه‌سازی کد

- 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.

تولید کد

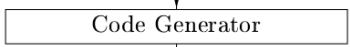
- در مرحله تولید کد [1] ، کد میانی معادل برنامه ورودی دریافت می‌شود و به یک برنامه در زبان مقصد تبدیل می‌شود. اگر زبان مقصد، زبان اسمبلی مختص به یک ماشین باشد، آنگاه از رجیسترها و قابلیت‌های ماشین مقصد برای تبدیل کد استفاده می‌شود. در مرحله تولید کد، استفادهٔ بهینه از رجیستر اهمیت زیادی پیدا می‌کند.

[1] code generation

تولید کد

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

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




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

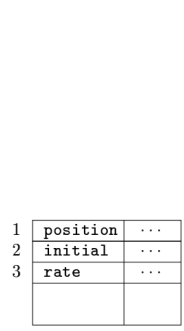
- حرف F در این دستورات نشان می‌دهد که محاسبات برروی اعداد اعشاری انجام می‌شود. دستور LD برای کپی کردن مقدار از حافظه در رجیستر و ST برای ذخیره کردن مقدار یک رجیستر در یک فضای حافظه استفاده شده است. دستور ADD برای جمع مقدار دو رجیستر و دستور MUL برای ضرب مقدار دو رجیستر استفاده شده است.

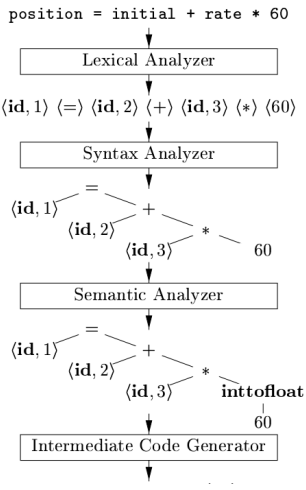
مدیریت جدول علائم

- یکی از وظایف کامپایلر نگهداری جدول علائم [1] 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.

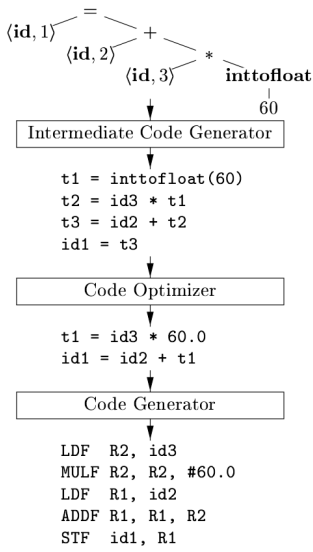
[1] symbol table

اجزای کامپایلر





اجزای کامپایلر



دسته‌بندی اجزای کامپایلر

- مراحل مختلف و اجزای متفاوت یک کامپایلر را می‌توان به دسته‌هایی تقسیم کرد. برای مثال می‌توان 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 یک کامپایلر به طور کلی از دو قسمت سمت کاربر یا بخش روساخت [1] و سمت ماشین یا بخش زیرساخت [2] تشکیل شده است.
- بخش زیرساخت بسته به زبان ماشین مقصد متفاوت خواهد بود و بنابراین به ازای ماشین‌های متفاوت بخش‌های زیرساخت متنوعی وجود خواهند داشت. اما بخش روساخت به ازای یک زبان ورودی همیشه یکسان است.
- با جداسازی این دو بخش می‌توان کامپایلری عرضه کرد که بخش زیرساخت آن برای ماشین‌های جدید قابل تعویض باشد.

[1] front-end
[2] back-end

ابزارهای تولید کامپایلر

- سازندگان کامپایلر مانند همهٔ توسعه‌دهندگان نرم‌افزار از ابزارهای متفاوتی برای تهیه کامپایلر استفاده می‌کنند. برخی از این ابزارها به شرح زیر اند.

۱. تولید کنندهٔ تحلیل‌گر لغوی [1] به صورت خودکار با استفاده از یک توصیف از توکن‌های زبان در زبان منظم، تحلیل‌گر لغوی تهیه می‌کند.

۲. تولید کنندهٔ تجزیه کننده [2] به صورت خودکار یک تحلیل‌گر نحوی از یک توصیف گرامری تهیه می‌کند.

۳. مترجم نحوی [3] با استفاده از یک توصیف معنایی از قوانین گرامر، و مجموعه‌ای از قواعد برای پیمایش درخت تجزیه، یک تحلیل‌گر معنایی و تولیدکنندهٔ کد میانی تهیه می‌کند.

[1] scanner generator
[2] parser generator
[3] syntax-directed translator

ابزارهای تولید کامپایلر

۴. تولید کنندهٔ تولید کننده کد [4] : با استفاده از مجموعه‌ای از قوانین ترجمه دستورات کد میانی به کد ماشین، یک تولید کنندهٔ کد تهیه می‌کند.

۵. موتور تحلیل جریان داده [5] : تحلیل جریان داده یکی از بخش‌های مهم بهینه‌ساز کد است. موتور تحلیل جریان‌داده با جمع‌آوری اطلاعات در مورد اینکه بخش‌های مختلف برنامه چگونه با یکدیگر ارتباط برقرار می‌کنند، کد تولید شده را بهینه‌سازی می‌کند.

۶. ابزار تولید کامپایلر [6] : مجموعه‌ای از توابع و ابزارها برای ساخت کامپایلر ارائه می‌کند.

[4] code-generator generator
[5] data-flow analyzer
[6] compiler-construction toolkit

تکامل زبان‌های برنامه‌نویسی

- کامپیوترهای الکترونیکی ابتدایی در دهۀ 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 برروی 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.

زبان‌های سطح بالا

- اولین گام در طراحی زبان‌هایی که به زبان انسان شباهت بیشتری داشته باشند تا زبان ماشین، در طراحی زبان اسمبلی در اوایل دهۀ 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 [1] برای انجام محاسبات علمی، زبان کوبول [2] برای انجام پردازش داده در کاربردها تجاری، و زبان لیسپ [3] برای انجام محاسبات بر روی لیست‌ها و کاربردهای هوش‌مصنوعی 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.

[1] Fortran
[2] Cobon
[3] Lisp

دسته‌بندی زبان‌ها

- زبان‌های برنامه‌نویسی را می‌توان براساس رویکرد آنها به توصیف برنامه‌ها دسته‌بندی کرد. یک دسته از زبان‌های برنامه‌نویسی که زبان‌های دستوری [1] 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 اعلامی [2] نامیده می‌شوند. یک برنامه به زبان اعلامی مشخص می‌کند چه محاسباتی باید انجام شود و چگونگی انجام محاسبات غالبا بر عهده کامپایلر است.
- زبان‌های سی، سی++ و جاوا در دسته زبان‌های دستوری قرار می‌گیرند و زبان‌های لیسپ، ام‌ال، هسکل و پرولوگ در دستهٔ زبان‌های اعلامی.

[1] imperative
[2] declarative

دسته‌بندی زبان‌ها

- همچنین می‌توان زبان دستوری را به دو دسته زبان‌های رویه‌ای [1] و زبان‌های شیءگرا [2] 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 تابعی [3] زبان‌هایی هستند که یک برنامه را با استفاده از تعدادی تابع همانند توابع ریاضی توصیف می‌کنند و زبان‌های منطقی [4] زبان‌هایی هستند که یک برنامه را با استفاده از تعدادی گزاره منطقی توصیف می‌کنند. لیسپ 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

علم طراحی کامپایلر

- 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 بشناشند.
- موضوع علم کامپایلر این است که چگونه الگوریتم‌های مناسب را برای پیاده‌سازی کامپایلر یک زبان به‌کار ببریم به طوری‌که از طرفی الگوریتم‌های طراحی شده عمومی باشند و همهٔ برنامه‌های ورودی را در بربگیرند و از طرفی کارامدی بالایی داشته باشد. بنابراین در طراحی کامپایلر به علم طراحی الگوریتم و ساختمان داده و معماری ماشین نیاز است.
- همچنین از علم نظریهٔ زبان‌های و ماشین‌ها در کامپایلر استفاده می‌شود، چراکه ماشین‌های متناهی و گرامرهای منظم برای توصیف واژگان زبان و گرامرهای مستقل از متن و الگوریتم‌های مربوط به آنها برای توصیف ساختار زبان و پیاده‌سازی آن مورد استفاده قرار گیرند. در پیاده‌سازی یک کامپایلر اصول و قوانین طراحی نرم‌افزار نیز باید رعایت شوند، بنابراین از علم طراحی نرم‌افزار نیز استفاده می‌شود.