به نام خدا

Compiler Design

مهدی حق‌وردی

دانشگاه اصفهان

فهرست مطالب

مقدمه

مقدمه

- در این ارائه به بررسی طراحی کامپایلر می‌پردازیم.
- در این ارائه سعی شده است که با نشان دادن مثال و کدهای واقعی فهم قسمت‌های مختلف یک کامپایلر (یا مفسر) برای شما ساده شود.
- در ابتدا به معرفی و بررسی قسمت‌های مختلف یک کامپایلر پرداخته می‌شود.
- در حین معرفی و توضیح، مثال‌هایی از آن قسمت نشان داده می‌شود،
- و در پایان دو کتاب برای مطالعه‌ی عمیق روی کامپایلرها و مفسرها معرفی می‌شوند.

مقدمه

- در این ارائه به بررسی طراحی کامپایلر می‌پردازیم.
- در این ارائه سعی شده است که با نشان دادن مثال و کدهای واقعی فهم قسمت‌های مختلف یک کامپایلر (یا مفسر) برای شما ساده شود.
- در ابتدا به معرفی و بررسی قسمت‌های مختلف یک کامپایلر پرداخته می‌شود.
- در حین معرفی و توضیح، مثال‌هایی از آن قسمت نشان داده می‌شود،
- و در پایان دو کتاب برای مطالعه‌ی عمیق روی کامپایلرها و مفسرها معرفی می‌شوند.

مقدمه

- در این ارائه به بررسی طراحی کامپایلر می‌پردازیم.
- در این ارائه سعی شده است که با نشان دادن مثال و کدهای واقعی فهم قسمت‌های مختلف یک کامپایلر (یا مفسر) برای شما ساده شود.
- در ابتدا به معرفی و بررسی قسمت‌های مختلف یک کامپایلر پرداخته می‌شود.
- در حین معرفی و توضیح، مثال‌هایی از آن قسمت نشان داده می‌شود،
- و در پایان دو کتاب برای مطالعه‌ی عمیق روی کامپایلرها و مفسرها معرفی می‌شوند.

مقدمه

- در این ارائه به بررسی طراحی کامپایلر می‌پردازیم.
- در این ارائه سعی شده است که با نشان دادن مثال و کدهای واقعی فهم قسمت‌های مختلف یک کامپایلر (یا مفسر) برای شما ساده شود.
- در ابتدا به معرفی و بررسی قسمت‌های مختلف یک کامپایلر پرداخته می‌شود.
- در حین معرفی و توضیح، مثال‌هایی از آن قسمت نشان داده می‌شود،
- و در پایان دو کتاب برای مطالعه‌ی عمیق روی کامپایلرها و مفسرها معرفی می‌شوند.

مقدمه

- در این ارائه به بررسی طراحی کامپایلر می‌پردازیم.
- در این ارائه سعی شده است که با نشان دادن مثال و کدهای واقعی فهم قسمت‌های مختلف یک کامپایلر (یا مفسر) برای شما ساده شود.
- در ابتدا به معرفی و بررسی قسمت‌های مختلف یک کامپایلر پرداخته می‌شود.
- در حین معرفی و توضیح، مثال‌هایی از آن قسمت نشان داده می‌شود،
- و در پایان دو کتاب برای مطالعه‌ی عمیق روی کامپایلرها و مفسرها معرفی می‌شوند.

معرفی

معرفی

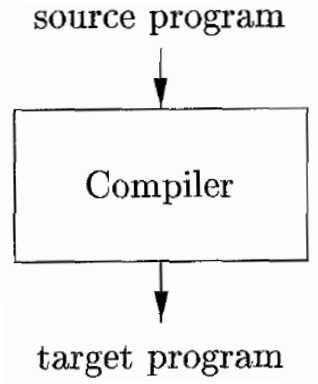
- زبان‌های برنامه‌نویسی، نمادها و ابزاری برای توصیف محاسبات برای انسان‌ها و کامپیوترها هستند.
- جهانی که ما می‌شناسیم به زبان‌های برنامه‌نویسی وابستگی بسیار زیادی دارد، چون تمام نرم‌افزارهای روی کامپیوترها دنیا با زبانی نوشته شده‌اند.
- اما قبل از اینکه بتوانیم آنها را **اجرا (run)** کنیم، باید بتوانیم آنها را به حالتی تبدیل کنیم که پردازنده‌های ما بتوانند آن‌ها را اجرا کنند.

نرم‌افزاری که این کار را برای ما انجام می‌دهد، **کامپایلر** نام دارد.

پردازشگرهای زبان

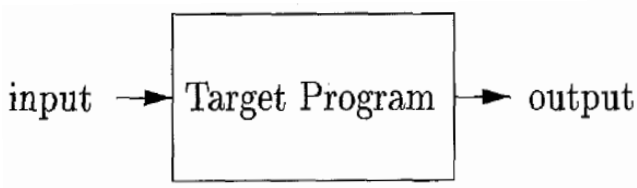
پردازش‌گرهای زبان

- به بیان ساده، کامپایلر برنامه‌ایست که می‌تواند یک برنامه را با یک زبان (*source* language) بخواند، و معادل آن را به زبانی دیگر ترجمه کند (*targer* language.)



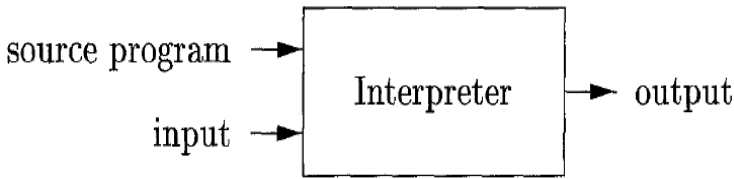
برنامه‌ی ترجمه شده

- اگر برنامه‌ی تولید شده، یک executable machine-language program باشد، می‌توان آن را مستقیما توسط پردازنده اجرا کرد.

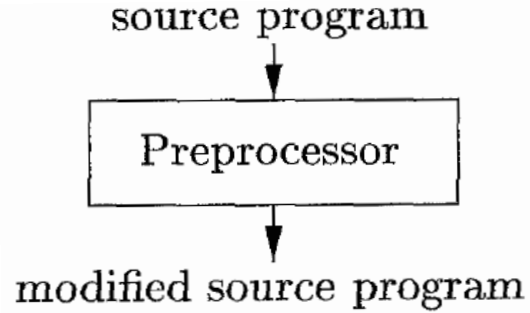


مفسر

- نوع دیگری از پردازش‌گرهای زبانی، مفسرها هستند که بجای تبدیل زبان به زبان دیگر ($target$)، خود مستقیما مسئول اجرای زبان اول ($source$) می‌شوند.



یک سیستم پردازشگر زبان - Preprocessor



Preprocessor

- زبان‌های کهنی مثل C و C++ سیستم moduling و ساختاربندی منظمی برای جدا کردن source codeهایشان نداشتند.
- اما زبان‌های جدیدتر مثل Python چنین سیستمی را دارند.

```
1 import math
2 from csv import reader, writer
```

- زبان‌های کهنی مثل C و C++ سیستم moduling و ساختاربندی منظمی برای جدا کردن source codeهایشان نداشتند.
- اما زبان‌های جدیدتر مثل Python چنین سیستمی را دارند.

```
1 import math
2 from csv import reader, writer
```

Preprocessor

- به همین دلیل، آنها نیاز داشتند که وقتی برنامه‌ای که نوشته‌اند را به چندین فایل تقسیم کرده‌اند، کامپایل کنند، برنامه‌ای تمام source codeهایشان را به یک source code واحد تبدیل کرده و آن را به کامپایلر بدهند، که بخاطر این نیاز، نرم‌افزاری به نام Preprocessor نوشته شد.

main cpython / Python / ceval.c

markshannon GH-111485: Generate instruction and uop metadata (GH-113287)

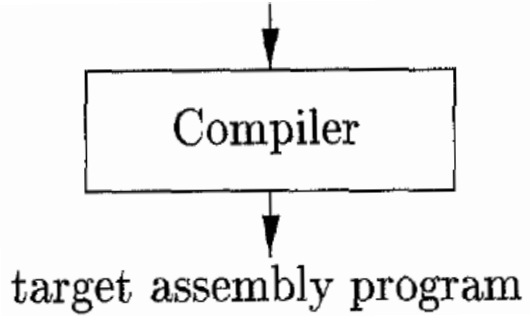
Code Blame 2903 lines (2644 loc) · 87.6 KB

```c
/* Execute compiled code */

#define _PY_INTERPRETER

#include "Python.h"
#include "pycore_abstract.h"      // _PyIndex_Check()
#include "pycore_call.h"          // _PyObject_CallNoArgs()
#include "pycore_ceval.h"         // _PyEval_SignalAsyncExc()
#include "pycore_code.h"
#include "pycore_emscripten_signal.h"  // _Py_CHECK_EMSCRIPTEN_SIGNALS
#include "pycore_function.h"
#include "pycore_instruments.h"
#include "pycore_intrinsics.h"
#include "pycore_long.h"          // _PyLong_GetZero()
```

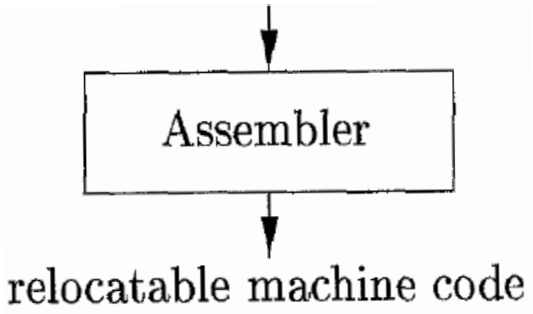
- با Preprocessorها می‌توان ماکرو به زبان اضافه کرد.

```
73  #define Py_XDECREF(arg) \
74      do { \
75          PyObject *xop = _PyObject_CAST(arg); \
76          if (xop != NULL) { \
77              Py_DECREF(xop); \
78          } \
79      } while (0)
```

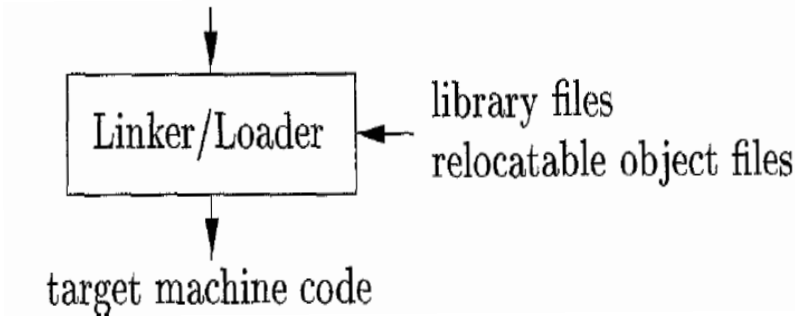
یک سیستم پردازشگر زبان - Compiler



یک سیستم پردازشگر زبان - Assembler



یک سیستم پردازشگر زبان - Linker/Loader



ساختار یک کامپایلر

ساختار کلی

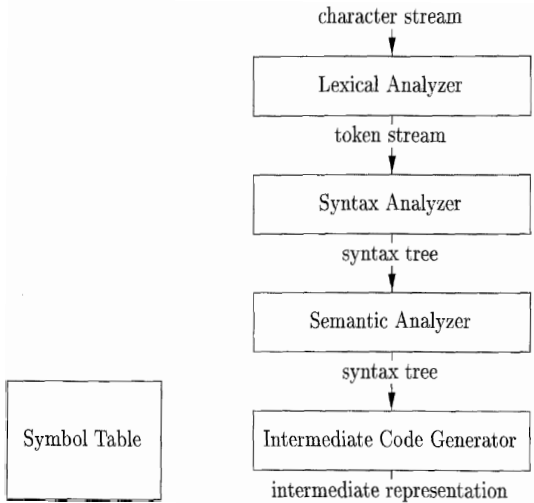
- تا کنون به کامپایلر به عنوان یک جعبه دارای ورودی خروجی نگاه می‌کردیم،
- اما اگر این جعبه را باز کنیم، با دو قسمت اصلی در کامپایلرها مواجه می‌شویم:
  ۱. آنالیز
  ۲. سنتز

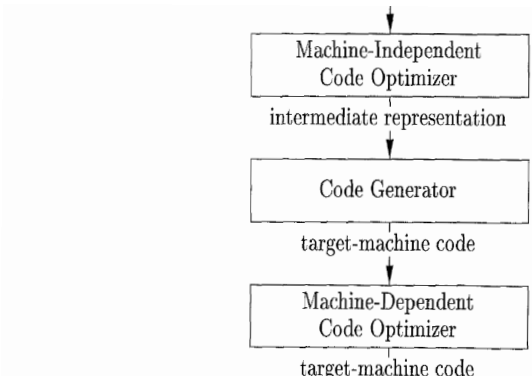
آنالیز

- این قسمت، source code را به قسمت‌های مختلفی می‌شکند،
- و قواعد گرامی را به قسمت‌های مختلف تحمیل می‌کند.
- این قسمت، اگر قسمتی را مخالف قوانین و گرامر زبان پیدا کند، پیام‌های مطلع کننده‌ای به کار نشان می‌دهد که ورودی را اصلاح کند.
- و در پایان این قسمت، داده ساختاری به نام *symbol table* را تولید می‌کند که تقریبا در تمامی گام‌های کامپایل (که در ادامه به آنها پرداخته می‌شود،) استفاده می‌شود.

سنتز

- قسمت سنتز، بعد از رد شدن از تمامی مراحل قسمت آنالیز، خروجی مورد نیاز ما را تولید می‌کند.
- به قسمت آنالیز front-end و قسمت سنتز back-end هم گفته می‌شود.


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Symbol Table
Intermediate Code Generator
intermediate representation


Machine-Independent
Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent
Code Optimizer
target-machine code

تحلیل واژگانی (Lexical Analysis)

- اولین کار کامپایلر اسکن کردن فایل داده شده و شکستن آن به توکن‌های کوچک‌تریست که در آن زبان تعریف شده.
- چیزی که تحلیل‌گر واژگان تولید می‌کند، دنباله‌ای از چنین ترکیبی‌ست:

$$\langle \text{token-name}, \text{attribute-value} \rangle$$

تحلیل واژگانی (Lexical Analysis)

- فرض کنید چنین عبارتی را می‌خواهیم آنالیز کنیم:

```
position = initial + rate * 60
```

- چیزی که تحلیل‌گر واژگان تولید می‌کند:

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

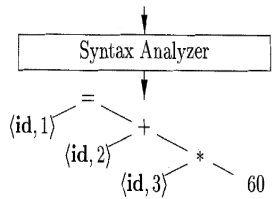
- که **id** مخفف identifier و عدد روبه‌روی آن اندیس symbol table است.

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

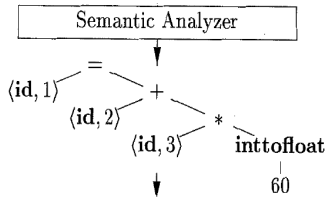
تحلیل نحوی (Syntax Analysis)

- سپس توکن به قسمت بعدی که تحلیل‌گر نحوی است فرستاده می‌شوند.
- در این قسمت با توجه به گرامر زبان، یک ساختار درختی از توکن‌ها تولید می‌شود که با قواعد زبان سازگار است (یکی از قواعد تقدم عملگرهای ریاضیست)



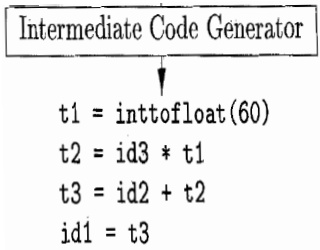
تحلیل معنایی (Semantic Analysis)

- درخت تولید شده در مرحله‌ی قبل، به این قسمت فرستاده می‌شود.
- در این قسمت، درخت تولید شده با اطلاعاتی دیگری همچون type متغیرها بررسی می‌شود، و اگر تداخلی با زبان داشت، گزارش می‌شود. برای مثال یک زبان اجازه جمع شدن یک `float` با یک `int` را نمی‌دهد که در این مرحله این اشتباه کشف و گزارش می‌شود.
- همینطور، این قسمت اگر بتواند type case مناسبی را انجام دهد، این کار را می‌کند. در مثال ما، متغیرهای `position`، `initial` و `rate` از نوع `float` تعریف شده‌اند و عدد 60 هم باید به `float` تبدیل می‌شود.



تولید کد میانی (Intermediate Code Generation)

- کامپایلرهای غالبا بعد از طی کردن مراحل قبل، کدی که بتوان آنرا بسادگی تولید و بسادگی ترجمه کرد را تولید می‌کنند تا سپس بتوانند آن را در مرحله‌ی بعدی بهینه کنند.



بهینه‌سازی کد (Code Optimization)

- یکی از قسمت‌های بسیار مهم در کامپایلرهای قسمت بهینه‌سازی کد است. کامپایلرهای کنونی بسیار باهوش هستند و می‌توانند روند کدی که نوشته شده را بررسی با توجه به ISA معماری‌ای که برای آن می‌خواهند کد ماشین تولید کنند، بهترین دستورات را انتخاب کنند تا برنامه سریع‌تر اجرا شود.
- به نظر شما یک کد اسمبلی که یک برنامه‌نویس آن را نوشته سریع‌تر است یا یک کدی که به زبان C نوشته شده و توسط یک کامپایلر مثل GCC یا Clang به اسمبلی ترجمه شده؟