طراحی زبان‌های برنامه‌سازی

استاد: محمد ایزدی

پروژه بهار ۱۴۰۱

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

نیم‌سال دوم ۱۴۰۱-۱۴۰۰

مهلت ارسال:
۲۵ تیرماه ۱۴۰۱
ساعت ۲۳:۵۹

به نکات زیر توجه کنید:

* پروژه تحویل حضوری (مجازی) خواهد داشت.
* نمره‌ی اصلی پروژه از ۱۰۰ حساب می‌شود و ۳۵ نمره هم به عنوان نمره‌ی امتیازی قرار داده شده است.
* پروژه را در گروه‌های ۲ یا ۳ نفره انجام دهید.
* در نهایت تمام فایل‌های خود را در یک فایل زیپ با نام $P1_StudentID$ قرار دهید و در کوئرا آپلود کنید. آپلود یکی از اعضای گروه کافیست.
* دقت کنید که در صورتی که print را پیاده‌سازی نکنید، باید یک ابزار نظارتی برای بررسی صحت اجرای برنامه‌ها داشته باشید.
* لطفا پروژه را خودتان انجام دهید و از دیگران کپی نکنید. در صورت وقوع چنین مواردی مطابق با سیاست درس رفتار می‌شود.
* در صورتی که سوال یا ابهامی درباره‌ی این پروژه داشتید پرسش خود را زیر پست مربوطه در کوئرا مطرح کنید.

۱ تعریف گرامر

در این فاز ما قصد داریم یک مفسر برای یک زبان ساده طراحی کنیم. گرامر این زبان به شکل زیر است:

$$1.\ Program \rightarrow Statements\ EOF$$

$$2.\ Statements \rightarrow Statement\ `;`\ |\ Statements\ Statement\ `;`$$

$$3.\ Statement \rightarrow Compound_stmt\ |\ Simple_stmt$$

$$4.\ Simple_stmt \rightarrow Assignment\ |\ Global_stmt\ |\ Return_stmt$$
$$|\ `pass`\ |\ `break`\ |\ `continue`$$

$$5.\ Compound_stmt \rightarrow Function_def\ |\ If_stmt\ |\ For_stmt$$

$$6.\ Assignment \rightarrow ID\ `=`\ Expression$$

$$7.\ Return_stmt \rightarrow `return`\ |\ `return`\ Expression$$

$$8.\ Global_stmt \rightarrow `global`\ ID$$

$$9.\ Function_def \rightarrow `def`\ ID\ `(`\ Params\ `)`\ `:`\ Statements$$
$$|`def`\ ID\ `()`\ :\ `\ Statements$$

$$10.\ Params \rightarrow Param_with_default\ |\ Params\ `,`\ Param_with_default$$

$$11.\ Param_with_default \rightarrow ID\ `=`\ Expression$$

$$12.\ If_stmt \rightarrow `if`\ Expression\ `:`\ Statements\ Else_block$$

$$13.\ Else_block \rightarrow `else`\ `:`\ Statements$$

$$14.\ For_stmt \rightarrow `for`\ ID\ `in`\ Expression\ `:`\ Statements$$

$$15.\ Expression \rightarrow Disjunction$$

$$16.\ Disjunction \rightarrow Conjunction\ |\ Disjunction\ `or`\ Conjunction$$

$$17.\ Conjunction \rightarrow Inversion\ |\ Conjunction\ `and`\ Inversion$$

$$18.\ Inversion \rightarrow `not`\ Inversion\ |\ Comparison$$

$$19.\ Comparison \rightarrow Eq_Sum\ |\ Lt_Sum\ |\ Gt_Sum\ |\ Sum$$

$$20.\ Eq_Sum \rightarrow Sum\ `==`\ Sum$$

$$21.\ Lt_Sum \rightarrow Sum\ `<`\ Sum$$

$$22.\ Gt_Sum \rightarrow Sum\ `>`\ Sum$$

$$23.\ Sum \rightarrow Sum\ `+`\ Term \mid Sum\ `-`\ Term \mid Term$$

$$24.\ Term \rightarrow Term\ `*`\ Factor \mid Term\ `/`\ Factor \mid Factor$$

$$25.\ Factor \rightarrow `+`\ Power \mid `-`\ Power \mid Power$$

$$26.\ Power \rightarrow Atom\ `**`\ Factor \mid Primary$$

$$27.\ Primary \rightarrow Atom \mid Primary\ `[`\ Expression\ `]` \mid Primary\ `()` \mid Primary\ `(`\ Arguments\ `)`$$

$$28.\ Arguments \rightarrow Expression \mid Arguments\ `,`\ Expression$$

$$29.\ Atom \rightarrow ID \mid `True` \mid `False` \mid `None` \mid NUMBER \mid List$$

$$30.\ List \rightarrow `[`\ Expressions\ `]` \mid `[]`$$

$$31.\ Expressions \rightarrow Expressions\ `,`\ Expression \mid Expression$$

همانطور که در گرامر این زبان مشخص است برنامه‌های این زبان شامل تعدادی statement هستند که با ; از هم جدا شده اند. این گرامر، یک نسخه‌ی ساده‌شده از گرامر زبان Python است و شکل کلی برنامه مشابه پایتون خواهد بود.

در ادامه چند نکته درباره‌ی این گرامر آمده‌اند و چند نکته‌ی دیگر هم در بخش ۳ نوشته شده‌اند.

- این گرامر (۱) LALR است بنابراین برای پیاده‌سازی‌های بعدی نیاز به هیچ تغییری در آن نخواهیم داشت. (فقط برای اطلاع، اگر نمی‌دانید LALR چیست اهمیتی ندارد.)
- یکی از موارد که در این گرامر ساده‌شده وجود دارند و آن را از گرامر پایتون متمایز می‌کنند، آن است که در این پروژه بین هر دو statement یک ; خواهد آمد و در مقابل نیازی به رعایت INDENT نیست.
- توجه کنید که NUMBER نماینده‌ی اعداد مثبت صحیح و یا اعشاری است.

- موارد درون “ ، terminal هایی هستند که به همین شکل در زبان ظاهر خواهند شد. مواردی که همه حروف آن‌ها بزرگ است مانند $NUMBER$ نیز terminal هستند با این تفاوت که می‌توانند مقادیر مختلفی بگیرند و مقدار مورد نظر برنامه‌نویس به جای آن‌ها می‌آید. کلماتی هم که با حروف بزرگ شروع می‌شوند nonterminal هستند.

۲ پیاده‌سازی اسکنر و پارسر (۱۰ نمره)

مرحله‌ی اول پیاده سازی هر مفسری، پیاده‌سازی لکسر و پارسر مربوط به آن است. این دو ماژول با گرفتن رشته‌ی برنامه‌ی ورودی، درخت مربوط به آن برنامه را برمی‌گردانند. به طور مثال در صفحه‌ی ۶۴ کتاب مشاهده می‌کنید که برنامه‌ی ورودی به شکل زیر نوشته شده است.

```
<<-(-(x,3),-(v,i))>>
```

اما همان‌طور که می‌دانید برنامه‌ی ورودی مفسر، یک رشته است. پس باید به طریقی این رشته را به شکلی مشابه عبارت بالا تبدیل کنیم. دو ماژول لکسر و پارسر این کار را برایمان انجام می‌دهند. حال به اختصار وظیفه‌ی هرکدام از این دو ماژول را بیان می‌کنیم:

* لکسر: این ماژول رشته‌ی ورودی را تکه‌تکه کرده و به کلمات (توکن‌های) اصلی برنامه می‌شکند. به طور مثال برنامه‌ی "a=2" به کلمات a و = و ۲ شکسته می‌شود. برای مشاهده‌ی بهتر کارکرد این ماژول مثال آورده‌شده به زبان رکت lexer.rkt را اجرا کرده و خروجی را مشاهده کنید.

* پارسر: این ماژول وظیفه‌ی ساخت درخت را از روی کلمات خروجی لکسر و از روی گرامر دارد. به طور مثال گرامر ساده‌ی $Exp \rightarrow Exp + NUMBER \mid NUMBER$ را در نظر بگیرید. حال فرض کنید رشته‌ی ورودیمان "1 + 2 + 3 + 4" است. می‌دانیم لکسر این رشته را تبدیل به کلمات ۱ و + و ۲ و + و ۳ و + و ۴ می‌کند. حال انتظار ما از پارسر این است که با این کلمات، درخت $(+\ (+\ (+\ (1)\ 2)\ 3)\ 4)$ را بسازد. برای نمونه کد parser.rkt را اجرا کنید و خروجی آن را ببینید.

شما در این بخش باید کلمات و گرامر پروژه را برای لکسر و پارسر تعریف کنید و به عبارتی پس از انجام این بخش، باید یک تابع parse داشته‌باشید که یک رشته شامل برنامه‌ای در زبان پایتون ساده‌شده‌ی این پروژه را به عنوان ورودی بگیرد و درخت پارس‌شده‌ی آن را به عنوان خروجی برگرداند.

برای ساخت لکسر و پارسر راه‌های مختلفی وجود دارد. یک راه این است که کد تمام بخش‌ها را از پایه خودمان طراحی کنیم اما این کار ممکن است زمان‌بر باشد و موضوع این درس هم نیست (این موضوعات و جزئیات دیگری از گرامرها که در این‌جا به آن‌ها نیازی نداریم در درس‌هایی مانند طراحی کامپایلرها بررسی می‌شوند).

یک راه دیگر، این است که از ابزارهای سازنده‌ی لکسر و پارسر استفاده کنیم. با این ابزارها می‌توانیم بدون درگیر شدن با جزئیات پیاده‌سازی، توصیف گرامر مطلوب را به لکسر و پارسر تبدیل کنیم. زبان رکت ابزارهای مناسبی برای این کار در اختیار ما قرار می‌دهد. در این پروژه هم برای پیاده‌سازی این بخش از ابزارهای موجود در این زبان استفاده می‌کنیم. برای مطالعه‌ی بیشتر می‌توانید به منبع زیر نگاه کنید.

https://docs.racket-lang.org/parser-tools/index.html

دقت کنید که در فایل‌های نمونه‌ی lexer.rkt و parser.rkt هم از این ابزارها استفاده شده است. با مطالعه‌ی آن‌ها و توجه به این موضوع که گرامر مورد استفاده در این پارسر نمونه همان گرامر ساده‌ی $Exp \rightarrow Exp + NUMBER \mid NUMBER$ است، می‌توانید دید بهتری از روند کار به دست بیاورید.

۳ پیاده‌سازی اولیه‌ی مترجم (۶۰ نمره)

در این بخش شما باید به کمک آموخته‌های خود از این درس، یک مفسر ساده برای این زبان پیاده‌سازی کنید. دقت کنید که در تست‌های نهایی، هیچ برنامه‌ای خطای نحوی نخواهد داشت و بنابراین نیازی به پیاده‌سازی Error Handler نیست.

در ادامه توضیحات بیشتری درباره‌ی چند مورد از گرامر آمده است تا دید بهتری از آن به دست بیاورید.

- $pass$ دستوری است که هیچ کاری انجام نمی‌دهد.
- خط ۸: عبارت global برای آن استفاده می‌شود که یک متغیر موجود در بلوک بیرونی یک تابع را درون تابع استفاده کنیم. دقت کنید اگر متغیری را بعد از تعریف تابع نیز تعریف کنید، می‌توانید از آن استفاده کنید. به طور مثال کد زیر:

```
1  def f():
2      global a;
3      a = a + 1;
4      ;
5  a = 2;
6  print(a);
7  b = f();
8  print(a);
```

مقدار ۲ و ۳ را پرینت می‌کند ولی کد زیر

```
def f():
    global a;
    a = a + 1;
    ;
b = f();
```

مشکل دارد و نیازی نیست آن را در نظر بگیرید. در صورتی که یک متغیر را گلوبال کنیم, تغییرات آن را درون تابع به بیرون تابع نیز اعمال خواهد شد. همچنین در صورتی که درون یک تابع یک متغیر را گلوبال تعریف نکنیم، به هیچ‌وجه نمی‌توان از آن استفاده کرد(این مورد با پایتون تفاوت دارد.)

- خط ۹: در هنگام تعریف تابع باید به تمام متغیرهای آن مقدار اولیه بدهید.
- اعداد در زبان ما به دو شکل صحیح و اعشاری هستند. هنگام تعریف درصورتی که عدد تعریف شده نقطه (.) نداشت، صحیح و در غیر این صورت اعشاری است.
- خط ۱۵: جلوی عبارت in تنها یک لیست می‌تواند بیاید. (این مورد بر خلاف پایتون است.)
- عبارات شرطی که در if و for می‌آیند تنها باید از نوع boolean باشند، یعنی باید پس از evaluate شدن در نهایت به '$True$' یا '$False$' برسند و نمی‌توانیم از بقیه‌ی موارد، مثل اعداد، برای این موضوع استفاده کنیم. (این مورد بر خلاف پایتون است.)
- خط ۲۳ و ۲۴ و ۲۶ گرامر: عملگرهای ریاضی تنها روی مقادیر هم‌نوع اجرا می‌شوند.
- خط ۲۳ و ۲۴ و ۲۶ گرامر: برای نوع داده‌ی list، تنها عملگر $+$ به معنای $concat$ (به هم چسباندن دو لیست) قابل اعمال است.
- خط ۲۵ گرامر: دو عملگر $+$ و $-$ تنها بر روی اعداد قابل استفاده هستند.
- خط ۲۷: فراخوانی تابع به شکل call by value است.
- در نهایت در هر قسمتی از پیاده‌سازی نیازی به فرض خاصی داشتید، آن را در یک داک نوشته و به همراه پروژه ارسال کنید. (نیاز به تهیه‌ی داک در حالت کلی نیست.)

۴ خاصیت بازگشتی توابع (۱۵ نمره)

برای این بخش شما باید ذخیره‌ی تابع را به نحوی انجام دهید که بتوانیم درون یک تابع، خودش را صدا بزنیم. به مثال زیر توجه کنید.

```
def f():
    a = print(2);
    a = f();
    ;

a = f();
```

۵ خواندن کد از فایل (۱۰ نمره)

در این بخش باید یک تابع evaluate بنویسید که به عنوان ورودی، آدرس کد موردنظر را بگیرد و آن را اجرا کند و خروجی را نمایش دهد. به طور مثال می‌توانیم کد موجود در فایل *example.txt* در کنار پروژه را به صورت زیر evaluate کنیم.

```
(evaluate "a.txt")
```

۶ تابع print (۵ نمره)

در این بخش باید یک تابع print را به زبان اضافه کنید به نحوی که با اجرای این تابع ورودی آن (که می‌تواند لیست، عدد و یا رشته باشد) نمایش داده شود. نحوه‌ی نمایش اهمیتی ندارد و این بخش صرفا جهت تست نهایی است. همچنین جهت راحتی فرض کنید ورودی این تابع نیازی به evaluate شدن ندارد و ورودی Atom یا لیستی از Atom هاست. این تابع را می‌توانید به گرامر اضافه کرده و یا به شکل یک تابع از پیش آماده در envinroment بگذارید.

۷ Lazy Evaluation (۲۰ نمره)

در این بخش شما باید Lazy Evaluation را پیاده‌سازی کنید که در زبان ما شامل موارد زیر می‌شود:

- در هنگام * کردن، در صورتی که سمت چپ ضرب ۰ بود، سمت راست محاسبه نشده و مقدار ۰ برگردانده می‌شود.
- یک متغیر که در Assignment مقدار دهی می‌شود، تا وقتی که از آن استفاده نشود، محاسبه نمی‌شود.
- تا وقتی از یک ورودی تابع استفاده نشده است، آن مقدار محاسبه نمی‌شود.

۸ Types (۱۵ نمره)

برای این بخش، از شما می‌خواهیم تا type-checking را به صورت محدود برای این زبان پیاده‌سازی کنید. برای این کار، به برنامه‌نویس این اختیار را می‌دهیم که در زمان Assignment برای متغیر یک نوع تعریف کند. همچنین این امکان را فراهم می‌کنیم تا برای هر تابع که تعریف می‌شود، نوع خروجی و ورودی قابل تعیین باشد.

با این قابلیت، مفسر شما باید پیش از اجرای برنامه در حدی که در کلاس گفته شده است امن بودن اجرای آن برنامه را بررسی کند و در صورتی که مشکلی وجود داشت، به کاربر اطلاع بدهد. به طور خاص در این پروژه دو مورد زیر مدنظر هستند.

- هر متغیری که استفاده می‌شود باید bind شده باشد.
- مقداری که هر متغیر می‌گیرد باید مطابق با نوع مشخص شده باشد.

در صورت پیاده‌سازی این بخش، قواعد زیر به گرامر اضافه می‌شوند.

$$32.\ Type \rightarrow \text{‘}int\text{‘} \mid \text{‘}float\text{‘} \mid \text{‘}bool\text{‘} \mid \text{‘}list\text{‘} \mid \text{‘}None\text{‘}$$

$$33.\ Assignment_lhs \rightarrow ID \mid ID\ \text{‘}:\text{‘}\ Type$$

$$34.\ Return_type \rightarrow \text{‘}:\text{‘} \mid \text{‘}->\text{‘}\ Type\ \text{‘}:\text{‘}$$

علاوه بر این، خط‌های ۱ و ۶ و ۹ و ۱۱ گرامر به صورت زیر تغییر خواهند کرد.

$$1.\ Program \rightarrow Statements\ EOF \mid \text{‘}checked\text{‘}\ Statements\ EOF$$

$$6.\ Assignment \rightarrow Assignment_lhs\ \text{‘}=\text{‘}\ Expression$$

$$9.\ Function_def \rightarrow \text{`}def\text{`}\ ID\ \text{`(`}\ Params\ \text{`)`}\ Return_type\ Statements$$
$$|\ \text{`}def\text{`}\ ID\ \text{`()`}\ Return_type\ Statements$$
$$11.\ Param_with_default \rightarrow Assignment_lhs\ \text{`}=\text{`}\ Expression$$

همان‌طور که مشاهده می‌کنید، در این‌جا (برای سادگی بیشتر) هنگامی که بخواهیم برنامه type-check شود، در ابتدای برنامه کلیدواژه‌ی '$checked$' نوشته می‌شود. زمانی که این کلیدواژه غایب باشد مفسر به تایپ‌های نوشته شده اهمیتی نمی‌دهد و اگر تایپ‌ها نوشته نشده باشند هم مشکلی به وجود نمی‌آید.

در مقابل، زمانی که این کلیدواژه مشاهده شود برنامه type-check می‌شود. در این صورت اگر برنامه مشکلی در این زمینه داشت این مشکلات گزارش می‌شوند و برنامه اجرا نمی‌شود. در غیر این صورت، برنامه مانند روند عادی اجرا می‌شود و روند اجرا پس از بررسی با چیزی که پیش‌تر گفتیم تفاوتی ندارد.

برای بررسی نوع مقدارها دقت کنید که در این گرامر ۵ نوع ممکن برای مقدارها در نظر گرفته شده است. اعداد می‌توانند از نوع 'int' یا '$float$' باشند و برای زمانی که مقدار موردنظر '$None$' باشد، نوع '$None$' را در نظر می‌گیریم. اگر هیچ نوعی برای یک متغیر نوشته نشود فرض می‌کنیم تمام ۵ نوع برای آن متغیر ممکن هستند و اشکالی برای آن به وجود نمی‌آید.

زمانی که در $Assignment$ یکی از این نوع‌ها همراه ' : ' در جلوی نام متغیر نوشته می‌شود، مفسر متوجه می‌شود که مقدار آن متغیر باید از آن نوع باشد. نوع خروجی تابع هم پس از تعریف تابع به همراه '$->$' می‌تواند مشخص شود. این سینتکس، مشابه معادلش در زبان پایتون است. برای اینکه دید بهتری از این سینتکس به دست بیاورید، به نمونه‌ی زیر توجه کنید.

```
checked

def add_one(n: int) -> int:
  return n + 1;
;

def f(n: int) -> None:
  a: int = add_one(n);
  b = print(a);
;
```

```

c: None = f(7);
```

موفق باشید : —)