زبان‌های برنامه‌سازی

سهیل محمدخانی، علیرضا کاویانی، سپهر رمضانی
گزارش فاز ۱

فهرست مطالب

۱ تغییرات گرامری ۲

۲ دیتاتایپ‌ها ۳
۱.۲ دیتاتایپ‌های اصلی ۳
۲.۲ دیتاتایپ‌های فرعی ۳

۳ تست‌بنچ‌ها ۴
۱.۳ تست‌های ساده ۴
۲.۳ تست‌های سخت ۴
۳.۳ تست‌های ارور هندلینگ: ۴
۴.۳ تست‌های تایپ‌چکر ۴
۵.۳ نحوه نمایش نتایج تست‌بنچ‌ها ۵

۴ محیط اجرا(environment) ۵
۱.۴ variable management ۵
۲.۴ function handling ۵
۳.۴ Lazy Evalutation ۶
۴.۴ Memory Management ۶

۵ ارزیابی عبارات ۶

۶ مدیریت خطا ۶

۷ TypeChecker ۸

۸ نتایج تست‌بنچ ها ۸

۱ تغییرات گرامری

با توجه به عدم آشنایی با چالش‌های فاز ۱، گرامر دستخوش تغییرات اندکی نسبت به فاز قبلی شد، این تغییرات همه افزایشی بودند و هر گرامری که در فاز ۰ گفته شده بوده پیاده‌سازی شده است.
همچنین لازم به ذکر است که همه تغییرات گرامری روی predefined statement ها بوده‌اند، در زبان طراحی شده، predefined statement ها دستوراتی شبیه به تابع هستند که پیاده‌سازی آنها internal است و همه آنها با $ شروع می‌شوند، این دستورات عبارتند از:

- $print
- $input
- $set
- $get
- $push
- $pop
- **$size**
- **$tocharlist**

دو دستور آخر که بولد هم شده‌اند، تفاوت گرامر این فاز نسبت به فاز قبل هستند، چون این دستورات از نوع اینترنال و پریدیفایند هستند حتما باید در گرامر ذکر شوند.، دلیل دستور $tocharlist این است که استرینگ در زبان ما به طول دیفالت به شکل لیست برسی نمی‌شود، برای همین، برای برسی محتوای آن باید دستوری برای کست استرینگ به لیست کاراکترها وجود داشته باشد(نیازی به پریدیفاین کردن کست برعکس نیست چرا که زبان از جمع استرینگ و کاراکتر ساپورت می‌کند و با این عملیات می‌توان کست برعکس را ساخت).
دلیل دستور $size این است که مشابها، اگر ورودی از جنس رشته باشد، برنامه‌نویس هیچ داده‌ای از طول ورودی ندارد، برای همین، حتی با وجود کست به لیست کاراکتر، هیچ راهی برای فهمیدن طول رشته ورودی وجود ندارد برای همین $size لیست نیز باید از قبل در زبان تعریف شده باشد.
طبعا تغییر گرامر موجب تغییر خیلی اندک در پارسر و لکسر زبان هم شد.

۲ دیتاتایپ‌ها

۱.۲ دیتاتایپ‌های اصلی

برای انتخاب دیتاتایپ‌ها مطابق گرامر فاز قبلی عمل شد، دیتاتایپ‌های اصلی این زبان عبارتند از:

۱. int

۲. float

۳. function

۴. string

۵. bool

۶. char

۷. list

همانطور که قبلا هم گفته شد، زبان طراحی شده استرینگ را به چشم یک آبجکت و دیتاتایپ جدا برسی می‌کند، و استرینگ مانند روش مرسوم به چشم یک لیست از کاراکترها برسی نمی‌شود و برای تبدیل آن به لیست کاراکتری، باید از پریدیفایند استیتمنت‌ها استفاده کرد.
در فایل datatype.rkt یک تابع جدا به نام `expval->string-for-print` جز توابع مرسومی که به روش کتاب پیاده‌سازی شده اند نیز وجود دارد که یک داده‌را به ولیو استرینگی برای پرینت کردن آن تبدیل می‌کند، این برای این است که برخلاف برخی زبان‌های سطح پایین، داده‌های غیر استرینگی نیز قابل پرینت کردن باشند، البته می‌شد این موضوع را در سمت اینترپرتر هندل کرد اما ترجیح داده شد تا در سمت دیتاتایپ‌ها این مورد ذکر شود.

۲.۲ دیتاتایپ‌های فرعی

در زبان ذکر شده، جز دیتاتایپ‌های ذکر شده، چند دیتاتایپ دیگر هم اضافه شده که برنامه‌نویس متوجه حضور آنها نیست:

۱. return

۲. break

۳. continue

این دیتاتایپ‌ها در حقیقت دیتاتایپ واقعی نیستند.
در پیاده‌سازی به روشی که کتاب زبان‌ها را برسی می‌کند، پیاده‌سازی زبان‌های فانکشنال کار راحتیست، همچنین زبان‌های سیکونشال نیز با ایده‌های کوچک قابل پیاده‌سازی هستند، اما از لحظه‌ای که دستوراتی که پیاده‌سازی یک بلاک را در زمان غیرقابل پیشبینی و در میان بلاک متوقف می‌کنند که دقیقا ۳ دستور ذکر شده هستند وارد کار می‌شوند، پیاده‌سازی زبان به شدت پیچیده می‌شود و حتی برای پیاده‌سازی این موراد تف‌ها و کارهای کثیفی انجام داده شد :)
وجود این ۳ دیتاتایپ جزئی از کارهای کثیفیست که برای پیاده‌سازی عملکرد این ۳ دستور مجبور به انجام آن شده‌ایم است.

۳ تست‌بنچ‌ها

قبل از توضیح ادامه موارد، نیاز است تا در مورد فرمت تست‌بنچ‌ها توضیح داده شود، نتیجه تست‌بنچ ها هم در فایل‌های جدا و هم عکس‌هایی از بخش‌هایی از آن در انتها ضمیمه شده، اما لازم به ذکر است که برای تست کردن فیچرهای زبان، ۴ تست بنچ اصلی و چند تست بنچ فرعی نوشته شده است، تست‌بنچ های فرعی در کدهای آپلود شده ضمیمه نشده(چون خیلی کثیف بودند و فقط برای دیباگ از آنها استفاده شد) اما ۲ تست بنچ اصلی که اولی بسیار کامل است به شرح زیرند:

۱.۳ تست‌های ساده

این تست‌بنچ در فایل `test-basic.rkt` قرار دارد.
در این تست بنچ، تمام فیچرهای زبان در کدهای کوچک مورد بررسی قرار گرفتند، این تست بنچ شامل ۱۱۵ تست است، اما هر تست آن در حد ۳ یا ۴ خط کد است و بیشتر برای بررسی هر فیچر زبان به صورت جدا و مستقل است.

۲.۳ تست‌های سخت

این تست‌بنچ در فایل `test-hard.rkt` قرار دارد.
در این تست بنچ، سه برنامه بزرگ نوشته شده که به ترتیب به شرح زیرند:

۱. محاسبه ۱۰ فیبوناتچی اول به روش تابع بازگشتی برای تست کردن رفتار تابع بازگشتی در شرایط کمی سخت تر از تست‌های عادی

۲. محاسبه ۱۰ فیبوناتچی اول به کمک دیپی، برای تست کردن رفتار دستورات لیستی در شرایط کمی سخت‌تر از تست‌های عادی

۳. پیاده‌سازی یک برنامه برای تبدیل تمام کاراکترهای کوچک یک رشته به کاراکترهای بزرگ، برای تست کردن رفتار دستورات مرتبط با رشته‌ها در شرایط کمی سخت‌تر از تست‌های عادی.

۳.۳ تست‌های ارور هندلینگ:

این تست‌بنچ در فایل `test-error-handling.rkt` قرار دارد و برای بررسی رفتار زبان در شرایط برخورد با خطای زمان اجرا طراحی‌شده است، لازم به ذکر است برای اجرای این تست‌ها مجبور به استفاده از try-catch شدیم چرا که طبق استاندارد داکیومنتیشن، هنگام برخورد با خطا باید برنامه متوقف شود اما تست بنچ باید همه تست‌ها را اجرا کند.

۴.۳ تست‌های تایپ‌چکر

این تست‌بنچ در فایل `test-typechecker.rkt` قرار دارد.
این تست‌بنچ از اینترپرتر مستقل است چرا که تایپ‌چکر جدا از اینترپرتر پیاده‌سازی شده است.
در این مورد بیشتر در بخش تایپچکر صحبت‌کرده ایم.

۵.۳ نحوه نمایش نتایج تست‌بنچ‌ها

نتایج همه تست‌بنچ‌ها هم به صورت فایل و هم به صورت عکس جزئی در بخش جداگانه ضمیمه شده، همچنین در هر قسمت، نتایج برخی از تست‌بنچ‌هایی که به آن بخش مربوطتند، ضمیمه شده است.

۴ محیط اجرا(environment)

محیط اجرا تقریبا از تمام توابعی که در کتاب ذکر شده، به علاوه چند تابع جزئی دیگر ساپورت می‌کند، همچنین این محیط در فایل `environment.rkt` پیاده‌سازی شده.
همچنین در صورت عدم وجود یک متغیر، برسی‌خطا در همین فایل صورت گرفته.

۱.۴ variable management

از آنجایی که در فاز گرامر استانداردی در خصوص اسکوپینگ مشخص نشد، زبان ما از لحاظ گرامری طوری طراحی شده که همه متغیرها لوکالند و متغیر گلوبالی وجود ندارد، طبعا چنین چیزی در روند اجرا تاثیرگزار نیست، همچنین لازم به ذکر است که برای تغییر گرامر برای ساپورت متغیر گلوبال، باید تغییرات زیادی در فاز ° صورت می‌گرفت و می‌بایستی این مورد در فاز قبل ذکر می‌شد.
ذخیره‌سازی متغیرها از هر نوعی به خصوص از انواع گفته شده در زبان قابل انجام است.
همچنین امکان شدو نیز وجود دارد، چون منابع پس از خارج شدن از هر اسکوپ آزاد می‌شوند، برای ساپورت شدو کافی بود تا تنها آخرین مقدار از یک اسم در انوایرمنت خروجی داده شود.

شکل ۱: بخشی از تست‌بنچ ساده، همچنین لازم به ذکر است مقدار دهی اولیه به لیست در زبان برای ساده‌سازی ممکن نیست و باید با push های متوالی اینکار انجام شود.

۲.۴ function handling

توابع مانند روش کتاب، به چشم یک متغیر دیده می‌شوند، همچنین با استفاده از روشی مانند روش کتاب برای پیاده‌سازی letrec امکان تعریف تابع بازگشتی نیز وجود دارد.

```
********** -> function tests
Testing: "int getValue() {return 42;}; getValue();"
Result: #(struct:num-val 42)

Testing: "int add(int a, int b) {return a + b;}; add(10, 22);"
Result: #(struct:num-val 32)

Testing: "int getValue() { return 10; }; int double(int x) { return x * 2; }; double(getValue());"
Result: #(struct:num-val 20)

Testing: "int fact(int n) {if (n == 0) {return 1;} else {return fact(n - 1) * n;}}; fact(6);"
Result: #(struct:num-val 720)
```

شکل ۲: بخشی از تست‌های ساده برای توابع

۳.۴ Lazy Evalutation

دقیقا استراکچرهایی مانند thunk در زبان پیاده‌سازی نشده، اما سعی شده اگر می‌شد، در محاسبه value-of هر نود AST اگر بچه سمت چپ کافی بود، بچه سمت راست را حساب نکنیم(تنها در آپریشن‌هایی که شبیه سی، به صورت لیزی ایولیویت می‌شوند، طبعا این روش کارایی شبیه thunk را ندارد اما اندکی به بهبود زمان اجرا کمک می‌کند.)

۴.۴ Memory Management

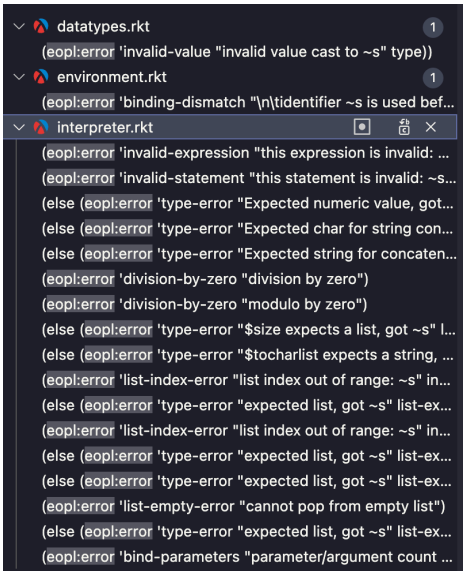
همه منابع داخل اسکوپ تعریف شده و خارج اسکوپ غیر قابل دسترسی هستند.

۵ ارزیابی عبارات

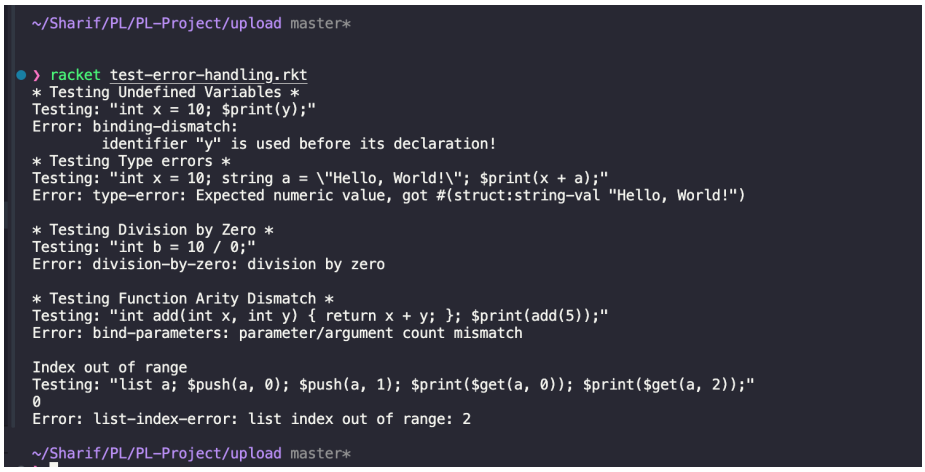
TODO

۶ مدیریت خطا

برای مدیریت خطا، از کتاب‌خانه eopl استفاده شده، همچنین به علت تایپ استریکت بودن زبان پیاده‌سازی شده، مدیریت خطاها کمی آسان‌تر از حالت عادی است.
برای مدیریت خطا، جز ۲ مورد(یکی در برای گرفتن مقدار وریبلی که در انوایرمنت نیست و دیگری و دیگری برای کستینگ ناموفق در دیتاتایپ‌ها) همه خطاها مربوط به اکسپرشن ایولیویشن هستند، و به راحتی حین ایولیوت کردن، قابل پیشبینی هستند، برای مثال وقتی valud-of برای اکسپرشن از نوع تقسیم پیاده‌سازی می‌شود، کافیست تا اگر ولیوآف سمت راست ۰ بود، توسط eopl::error خطای دیویژن بای زیرو داده شود.
هنگام بروز خطا نوع خطا و همچنین پیام مرتبط با آن نمایش‌داده می‌شود و اجرای برنامه متوقف می‌شود، نتایج تست بنچ ارور هندلینگ در زیر آمده که نشان می‌دهد همه خطاهایی که در داک ذکر شده بود برسی شده‌اند، همچنین، خطاهای بیشتری فرای آنچه در داک گفته شده، پیاده‌سازی شده‌اند.



شکل ۳: تمام خطاهایی که در زبان از آنها ساپورت می‌شود



شکل ۴: نتیجه تست بنچ ارور هندلینگ، دقت کنید اجرای هر برنامه در این تست بنچ در یک ترای‌کچ قرار دارد تا در صورت بروز خطا، برنامه متوقف نشود و همه تست‌ها اجرا شوند.

۷ TypeChecker

TODO

۸ نتایج تست‌بنچ ها