

زبان‌های برنامه‌سازی - گزارش فاز ۱

سهیل محمدخانی، علیرضا کاویانی، سپهر رضانی

فهرست مطالب

۲	۱	تغییرات گرامری
۳	۲	دیتاتایپ‌ها
۳	۱.۲	دیتاتایپ‌های اصلی
۳	۲.۲	دیتاتایپ‌های فرعی
۴	۳	تست‌بنج‌ها
۴	۱.۳	تست‌های ساده
۴	۲.۳	تست‌های سخت
۴	۳.۳	تست‌های ارور هندلینگ:
۴	۴.۳	تست‌های تایپ‌چکر
۵	۵.۳	نحوه نمایش نتایج تست‌بنج‌ها
۵	۴	محیط اجرا (environment)
۵	۱.۴	variable management
۵	۲.۴	function handling
۶	۳.۴	Lazy Evaluation
۶	۴.۴	Memory Management
۶	۵	ارزیابی عبارات
۶	۱.۵	توابع مهم در پیاده‌سازی اینترپرت
۷	۲.۵	عبارات پایه
۷	۳.۵	ساختار کنترلی
۸	۴.۵	توابع
۹	۵.۵	داده‌ساختارها (لیست)
۹	۶.۵	ورودی، خروجی، و سایر پریدیفایند استیتمنت‌ها
۱۰	۶	مدیریت خطا
۱۱	۷	TypeChecker
۱۱	۱.۷	محیط اجرای تایپ چکر
۱۲	۲.۷	تایپ‌چکر
۱۲	۳.۷	تست‌بنج تایپ‌چکر
۱۲	۸	نتایج تست‌بنج‌ها

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 بررسی نمی‌شود، برای همین، برای بررسی 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 مشابهها، 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 قبلاً هم گفته شد، زبان طراحی شده استرینگ را به چشم یک آبجکت و دیتاتایپ جدا بررسی می‌کند، و استرینگ مانند روش مرسوم به چشم یک لیست از کاراکترها بررسی 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 مرسوم می‌کند 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 بررسی می‌کند، پیاده‌سازی زبان‌های فانکشنال کار راحت‌تر است، 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 دقیقاً ۳ دستور ذکر شده هستند وارد کار می‌شوند، پیاده‌سازی زبان به شدت پیچیده می‌شود و حتی برای پیاده‌سازی این موارد تف‌ها و کارهای کثیفی انجام داده شد (:)

وجود این ۳ دیتاتایپ جزئی از کارهای کثیف‌ترین 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 تست‌بنچ‌ها 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، تست‌بنچ‌های 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 طراحی شده است، لازم به ذکر است برای اجرای این تست‌ها مجبور به استفاده از `try-catch` شدیم چرا که طبق استاندارد داکيومنتیشن، 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 صحبت کرده 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 تقریباً 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، بررسی‌خطا در همین فایل صورت گرفته.

۱.۴ variable management

از آنجایی که در فاز گرامر استاندارد در خصوص اسکوپینگ مشخص نشد، زبان ما از لحاظ گرامری طوری طراحی شده که همه متغیرها لوکالند و متغیر گلوبالی وجود ندارد، طبعاً چنین چیزی در روند اجرا تاثیرگذار 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.

```
Testing: "true || false;"
Result: #(struct:bool-val #t)

***** -> var declaration tests
Testing: "int x = 5;"
Result: #(struct:num-val 5)

Testing: "int y;"
Result: #(struct:num-val 0)

Testing: "bool b = false;"
Result: #(struct:bool-val #f)

Testing: "string s = \"salam\";"
Result: #(struct:string-val "salam")

Testing: "char c = 'a';"
Result: #(struct:char-val "a")

Testing: "list emp;"
Result: #(struct:list-val ())

Testing: "int getVal(){return 42;};"
Result: #(struct:function-val () (scope ((simple-stament (return-statement (return-statement (exp6 (exp5 (exp4 (exp3 (exp2 (exp1 (exp0 (atom (value-atom (int-val 42)))))))))))))) #(struct:extend-environment "getVal" #(struct:num-val 0) #(struct:empty-environment))))
```

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 Evaluation

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

تمام عبارات در گرامر پیاده‌سازی شده‌اند، این عبارات همه عبارات گفته شده در داکيومنتیشن و حتی مقدار خیلی بیشتری را تشکیل می‌دهند، تنها استثنا، عبارت for است که چون در اسپسیفیکیشن فاز قبل، تنها ذکر شده بود که نیاز است از یک نوع حلقه فارغ از نوع آن ساپورت کنیم، هم در فاز گرامر و هم در این فاز از حلقه فور صرف نظر کردیم چرا که پیاده‌سازی آن بسیار پیچیدگی به کد اضافه می‌کند و همچنین صرفا با داشتن while می‌توان تمام عملیات‌های فور را انجام داد.

۱.۵ توابع مهم در پیاده‌سازی اینترپرت

صرفا لازم به ذکره جاهایی که میبینیم نوشته شده tof سر اینه که سر پیاده‌سازی وایل مشکلاهی خیلی عجیبی خوردیم و مجبور شدیم تقریبا کل توابعمونو یکم عوض کنیم که اوکی باشند، سر همین تو ورژن قبل پاک کردن تابعای به درد نخور هر تابع ۲ تا ورژن داشت وگرنه دلیل دیگه‌ای نداره

- value-of-x: این تابع مانند کاریست که کتاب در طراحی زبان let انجام داد، این تابع، یک نود AST به تایپ x را روی محیط p ایولیویت کرده و نتیجه را بر می‌گرداند.
- exec-statement-list-tof: این تابع یک لیست از دستورات را گرفته و آنها را به ترتیب و به صورت سیکونشال اجرا کرده و در صورت نیاز تغییراتی روی انوایرمنت می‌دهد، این تابع دقیقا نقطه ایست که پیاده‌سازی تقریبا غیرخطی کتاب را تبدیل به یک پیاده‌سازی خطی و سیکونشال می‌کند (مانند تمرینی که در آن خواسته شده بود تا begin را پیاده کنیم)
- get-default-value: مقادیر دیفالت برای تایپ‌های متفاوت را خروجی می‌دهد

- `extract-var-name`: نام متغیر از نود AST مشخص می‌شود.
- `exec-function-declaration-unified`: تقریباً مشابه عملکرد `letrec`
- `bind-parameters`: لیست پارامترهای یک تابع را گرفته و با نگاه به انوایرمنت، بایندینگ‌های مورد نظر را انجام می‌دهد

همچنین جدای از پیاده‌سازی برای `valud-of-x` برای گرامرهایی که ساپورت کرده‌ایم، برای پریدیفایند استیمنت‌ها هم هر یک پیاده‌سازی مشخصی انجام شده تا زبان یک تورینگ کاملیت شود.

۲.۵ عبارات پایه

پیاده‌سازی عبارات پایه چلنج خاصی نداشت، برای عبارات ریاضی با توجه به گرامر ترتیب عملیات‌ها به درستی انجام می‌شود و در اینترپتر صرفاً باید عملیات مورد نظر را انجام دهیم که کار ساده‌ایست،

```

47 (define value-of-expression
48   (lambda (expr env)
49     (cond
50       [(and (pair? expr) (eq? (car expr) '+))
51        (let ((val1 (value-of-expression (cadr expr) env))
52              (val2 (value-of-expression (caddr expr) env)))
53          (let ((result (+ val1 val2)))
54            (update-environment val1 val2)))]
55       [(and (pair? expr) (eq? (car expr) '*))
56        (let ((val1 (value-of-expression (cadr expr) env))
57              (val2 (value-of-expression (caddr expr) env)))
58          (let ((result (* val1 val2)))
59            (update-environment val1 val2)))]
60       [(and (pair? expr) (eq? (car expr) '/))
61        (let ((val1 (value-of-expression (cadr expr) env))
62              (val2 (value-of-expression (caddr expr) env)))
63          (let ((result (/ val1 val2)))
64            (update-environment val1 val2)))]
65       [(and (pair? expr) (eq? (car expr) 'sqrt))
66        (let ((val1 (value-of-expression (cadr expr) env))
67              (val2 (value-of-expression (caddr expr) env)))
68          (let ((result (sqrt val1)))
69            (update-environment val1 val2)))]
70       [else
71        (error "Unknown operator: " (car expr)))]
72     ))

```

شکل ۳: نمونه‌ای از پیاده‌سازی چند آپریشن از آپریشن‌های ذکر شده:

```

3000 (define (eval-expr expr env)
3001   (cond
3002     [(and (pair? expr) (eq? (car expr) '+))
3003      (let ((val1 (eval-expr (cadr expr) env))
3004            (val2 (eval-expr (caddr expr) env)))
3005        (let ((result (+ val1 val2)))
3006          (update-environment val1 val2)))]
3007     [(and (pair? expr) (eq? (car expr) '*))
3008      (let ((val1 (eval-expr (cadr expr) env))
3009            (val2 (eval-expr (caddr expr) env)))
3010        (let ((result (* val1 val2)))
3011          (update-environment val1 val2)))]
3012     [(and (pair? expr) (eq? (car expr) '/))
3013      (let ((val1 (eval-expr (cadr expr) env))
3014            (val2 (eval-expr (caddr expr) env)))
3015        (let ((result (/ val1 val2)))
3016          (update-environment val1 val2)))]
3017     [(and (pair? expr) (eq? (car expr) 'sqrt))
3018      (let ((val1 (eval-expr (cadr expr) env))
3019            (val2 (eval-expr (caddr expr) env)))
3020        (let ((result (sqrt val1)))
3021          (update-environment val1 val2)))]
3022     [else
3023      (error "Unknown operator: " (car expr)))]
3024   ))

```

شکل ۴: بخشی از تست‌های عملیات‌های پایه:

۳.۵ ساختار کنترلی

- `if-else`: برای بلوک ایف خالی یا ایف ال‌اس، با داشتن ولیوآف بچه متناظر با کاندیشن، ادامه کار تنها گرفتن ولیوآف بچه‌ایست که طبق ولیو کاندیشن باید اجرا شود است، به عبارت دیگر، بجز حالت بندی روی نوع بلاک ایف و کثافت کاری‌های اینچنینی، کل کد مربوط به ایف در ۲ خط زیر قابل خلاصه است:

```
(if condition-val
    (exec-statement-tof then-scope env)
    (exec-statement-tof else-scope env))
```

[illegible]

شکل ۵: تست‌های ایف:

- while: برای وایل، از ساختار زیر برای سایورت آن استفاده شد:

```
(define exec-while-loop-tof
  (lambda (condition body-scope env)
    (let ((condition-val (expval->bool (value-of-expression condition env))))
      (if condition-val
          Later: Explained ;
          (cons (num-val 0) env)))))
```

قسمت خالی گذاشته شده، اگر از کانتینو و برک ساپورت نمی‌کردیم، بسیار راحت قابل پیاده‌سازی بود اما دقیقاً بخاطر ساپورت از کانتینو و برک و مشکلاتی که در کنترل فلو ساده برنامه به وجود می‌آمد، این تیکه بسیار پیچیده و طولانی شد.

[illegible]

شکل ۶: بخشی از تست‌های وایل:

٤.٥ توابع

در زبان ما توابع member های درجه اول هستند و در انوایرمنت تفاوتی با متغیرها ندارند، هرچند امکان پاس دادن تابع در زبان تعبیه نشد اما با تغییرات خیلی خیلی کم بخاطر نوع پیاده‌سازی، امکان این امر نیز ممکن می‌شود.

همچنین برای ساپورت از توابع بازگشتی، از تکنیکی مشابه لترک در کتاب استفاده شد

برای پیاده‌سازی توابع، ابتدا تابع `bind-parameters` پیاده‌سازی شد که برای بایند کردن پارامترهای مورد نیاز تابع با مقادیر موجود در انوایرمنت و درست کردن انوایرمنت مخصوص اسکوپ درون تابع است، همچنین، یک تابع دیگر به نام `extract-parameters-names` پیاده شده که نام پارامترهای تابع را خروجی می‌دهد. همچنین دوتابع برای ادامه کار پیاده شد، یکی `exec-function-declaration-to-f` که یک تابع را با تکنیکی مشابه لترک، در انوایرمنت تعریف می‌کند (و اسکوپ و نام و ریترن تایپ و پارامترهای آن را نیز در کنار آن ذخیره می‌کند) و دیگری `value-of-function-call` که تابع را فراخوانی کرده و حاصل فراخوانی را خروجی می‌دهد. تست های مربوط به فانکشن ها قبلا در قسمت‌های قبل (شکل ۲) ضمیمه شده بود و دوباره آن را ضمیمه نمی‌کنیم (همچنین از دایرکتوری `results` نتیجه همه تست‌ها قابل مشاهده است).

۵.۵ داده‌ساختارها (لیست)

تنها داده‌ساختار پیچیده که به صورت تایپ پریدیفاین شده در زبان وجود دارد لیست است تا زبان به یک زبان تورینگ کاملیت تبدیل شود (لیست یا آرایه برای بازی کردن نقش تیپ در یک ماشین تورینگ ضروریست) برای سادگی، در زبان ما نمی‌توان مقدار اولیه به لیست داد و با دستور پریدیفاین `$push` می‌بایستی مقادیر اولیه را پس از تعریف یک لیست تهی، یکی یکی به آرایه پوش کرد، برای تعریف لیست کار عجیبی انجام نشد، دیتاتایپ لیست درون خود به عنوان ولیو یک لیست از جنس لیست رکت نگه می‌دارد و پریدیفاین استیتمنت های `pop`، `push`، `set`، `get` به کد رکت روی لیست درونی تبدیل می‌شوند. به عنوان مثال، کد گت یا همان ایندکسینگ با پاک کردن موارد مربوط به ارور هندلینگ به کد ساده زیر تبدیل می‌شود:

```
(let ((list-expval (apply-env var-name env)))
(index-val (value-of-expression index-expr env)))
```

که `index-val` از توابع اینترنال رکت است.

دقت کنید که ایندکسینگ به صورت اپریتوری در زبان وجود ندارد و بجای آن پریدیفاین استیتمنت گت وجود دارد که نقش ایندکسینگ را ایفا می‌کند.

```
##### -> list tests
Testing: "list emptylist"
Result: #(<struct:List-val 0>)

Testing: "list list123: $push(list123, 2); $push(list123, 3); list123"
Result: #(<struct:List-val 3> #(<struct:num-val 1> #(<struct:num-val 2> #(<struct:num-val 3>)))

Testing: "list listpop: $push(listpop, 3); $push(listpop, 2); $push(listpop, 3); $pop(listpop); listpop"
Result: #(<struct:List-val 3> #(<struct:num-val 1> #(<struct:num-val 2>)))

Testing: "list list123: $push(list123, 1); $push(list123, 2); $push(list123, 3); $set(list123, 1);"
Result: #(<struct:num-val 3>)

Testing: "list list123: $push(list123, 3); $push(list123, 2); $push(list123, 1); $set(list123, 1, 7); list123"
Result: #(<struct:List-val 3> #(<struct:num-val 1> #(<struct:num-val 2> #(<struct:num-val 3>)))
```

شکل ۷: تست‌های ساده مربوط به لیست

۶.۵ ورودی، خروجی، و سایر پریدیفاین استیتمنت‌ها

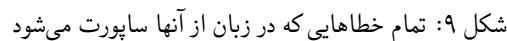
پیاده‌سازی پریدیفاین استیتمنت‌ها صرفاً معادل این بود که یک کد رکت برای برخی از توابع بزیم تا بتوان سایر کدها را به کدهای مرتبط به زبان ترجمه کرد، به عبارت دیگر، این توابع تنها برای تورینگ کاملیت کردن زبان بودند.

برای پیاده‌سازی پرینت، می‌بایستی تابعی می‌زدیم که یک دیتاتایپ که در پیاده‌سازی رکتی همیشه از جنس استراکت است را تبدیل به استرینگ می‌کردیم که در قسمت دیتاتایپ‌ها توضیح داده شد، چالش دیگری در این قسمت نبود، برای مثال، یکی از این پیاده‌سازی‌ها را ضمیمه کرده‌ایم:

شکل ۸: پیاده‌سازی دستور tocharlist که یک استرینگ را به لیست کاراکتر تبدیل می‌کند (چون استرینگ در زبان ما از جنس لیست نیست)

برای مدیریت خطا، از کتابخانه `error` 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 وریبل که در انوایرمنت نیست و دیگری و دیگری برای کستینگ ناموفق در دیتانایپ‌ها) همه خطاها مربوط به اکسپرسن ایلویشن هستند، و به راحتی حین ایلویشن کردن، قابل پیشبینی هستند، برای مثال وقتی `valud-of` برای اکسپرسن از نوع تقسیم پیاده‌سازی می‌شود، کافیت تا اگر ولوآف سمت راست ۰ بود، توسط `error::error` خطای دیویژن بای زیرو داده شود. هنگام بروز خطا نوع خطا و همچنین پیام مرتبط با آن نمایش داده می‌شود و اجرای برنامه متوقف می‌شود، نتایج تست پنج ارور هندلینگ در زیر آمده که نشان می‌دهد همه خطاهایی که در داک ذکر شده بود بررسی شده‌اند، همچنین، خطاهای بیشتری فرای آنچه در داک گفته شده، پیاده‌سازی شده‌اند.



```
~/Sharif/PL/PL-Project/upload master*
● > racket test-error-handling.rkt
* Testing Undefined Variables *
Testing: "int x = 10; $print(y);"
Error: binding-mismatch:
      identifier "y" is used before its declaration!
* Testing Type errors *
Testing: "int x = 10; string a = \"Hello, World!\"; $print(x + a);"
Error: type-error: Expected numeric value, got #(struct:string-val "Hello, World!")

* Testing Division by Zero *
Testing: "int b = 10 / 0;"
Error: division-by-zero: division by zero

* Testing Function Arity Dismatch *
Testing: "int add(int x, int y) { return x + y; }; $print(add(5));"
Error: bind-parameters: parameter/argument count mismatch

Index out of range
Testing: "list a; $push(a, 0); $push(a, 1); $print($get(a, 0)); $print($get(a, 2));"
0
Error: list-index-error: list index out of range: 2

~/Sharif/PL/PL-Project/upload master*
```

شکل ۱۰: نتیجه تست پنج ارور هندلینگ، دقت کنید اجرای هر برنامه در این تست پنج در یک برای یک قرار دارد تا در صورت بروز خطا، برنامه متوقف نشود و همه تست‌ها اجرا شوند.

TypeChecker V

دقت کنید که برای سادگی، یک محیط برای تایپ چک کردن و اینترپرت بعد آن ساخته نشده و این دو مستقل از هم عمل می‌کنند، برای همین برای اجرای کد، ابتدا باید یک بار به صورت دستی تایپ چکر را روی آن در صورت نیاز از اطمینان از لحاظ تایپ‌ها ران کنیم و سپس اینترپتر را روی آن اجرا کنیم، البته که تایپ‌ارورها به طول جداگانه در اینترپتر جلوی بسیاری از خطاهای تایپی را بدون اجرای تایپ‌چکر می‌گیرند. همانطور که قبلاً هم گفته شد، زبان طراحی شده زبان تایپ استریکت است و تمام تایپ‌ها به خصوص در ورودی و خروجی توابع، تایپ‌ها کاملاً مشخص شده اند و گرامر به طوری طراحی شده که زبان تایپ استریکت باشد. برای همین، نیازی به پشتیبانی از type inference نیست و تمام متغیرها بدون نیاز به استنتاج ثانویه، از همان اول تایپ مشخص دارند. پس می‌توان صرفاً با تایپ چکینگ عادی بدون حل یک دستگاه معادله یا کار پیچیده‌ای برای تایپ چکینگ، تایپ‌چکر زبان را نوشت.

۱.۷ محیط اجرای تایپ چکر

طبعاً تایپ چکر همانطور که در کتاب هم گفته شده بود، نیاز به یک انوایرمنت جدا دارد که در آن هر متغیر با تایپ آن بایند شود، این محیط اجرا مانند محیط اجرای اصلی با ۳ تابع زیر قابل تعریف می‌باشد:

```
(define empty-type-env '())

(define (env-lookup name env)
  (let loop ((lst env))
    (cond
      [(null? lst) #f]
```

```

[[equal? (caar lst) name) (cdar lst)]
[else (loop (cdr lst))]])))

(define (env-extend name typ env)
  (cons (cons name typ) env))

```

۲.۷ تایپ چکر

توابع مهم تایپ چکر و وظایف هر یک از آنها

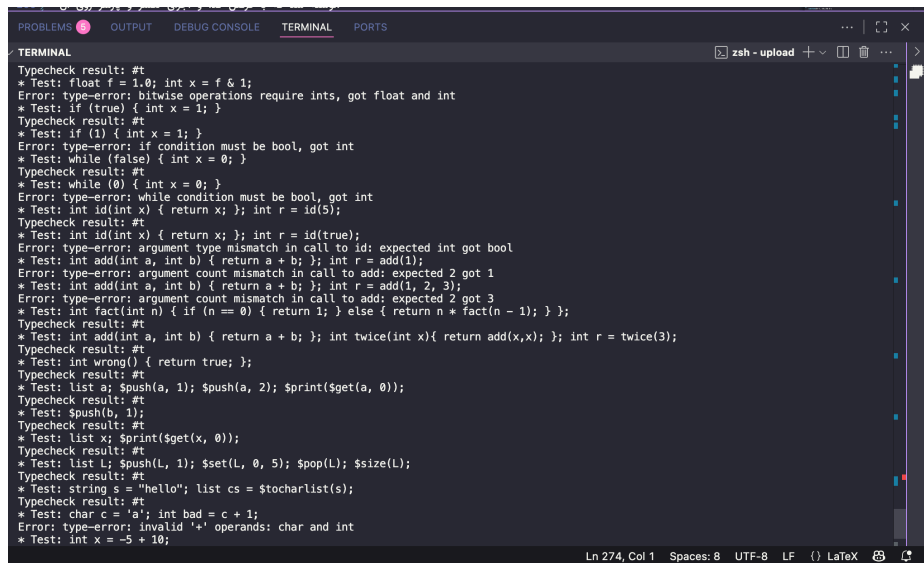
- `types-equal?`: چک کردن اینکه ۲ تایپ برابرند یا نه، اگر تایپ اول به دومی اتمات کست می‌شد، برابر محسوب می‌شوند (مانند اینت و فلوئت)
 - `type-of-expression`: این تابع مشابه آنچه در کتاب نوشته شده، تایپ یک اکسپرشن را تعیین می‌کند
 - `typecheck-smth`: این توابع هر یک با تایپ چک کردن یک نود خاص از *AST* و انجام آن به صورت بازگشتی روی بچه‌های آن، تایپ‌چکینگ را به طول کامل انجام می‌دهند.
- سعی شد تا جای ممکن پیاده‌سازی این توابع مانند چیزی که در کلاس و کتاب گفته شده بود انجام شود.

۳.۷ تست بنچ تایپ چکر

برای تست تایپ چکر، یک تابع به نام `test-parse-and-typecheck` نوشته شد که با گرفتن کد، و اجرای لکسر و پارسر روی آن، *AST* را گرفته و به تایپ چکر می‌دهد. چند تست متفاوت که تقریباً همه حالات درگیری تایپ چکر را مورد بررسی قرار می‌دهند، در فایل `test-typechecker.rkt` نوشته شده است.

۸ نتایج تست بنچ ها

نتایج تست بنچ ها در دایرکتوری `results` به صورت کامل قابل دسترسیست (همچنین خودتون هم میتونین اجراش کنین طبع :))
عکس از بخش‌هایی از تست بنچ‌ها نیز در زیر ضمیمه شده:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Typecheck result: #t
* Test: float f = 1.0; int x = f & 1;
Error: type-error: bitwise operations require ints, got float and int
* Test: if (true) { int x = 1; }
Typecheck result: #t
* Test: if (1) { int x = 1; }
Error: type-error: if condition must be bool, got int
* Test: while (false) { int x = 0; }
Typecheck result: #t
* Test: while (0) { int x = 0; }
Error: type-error: while condition must be bool, got int
* Test: int id(int x) { return x; }; int r = id(5);
Typecheck result: #t
* Test: int id(int x) { return x; }; int r = id(true);
Error: type-error: argument type mismatch in call to id: expected int got bool
* Test: int add(int a, int b) { return a + b; }; int r = add(1);
Error: type-error: argument count mismatch in call to add: expected 2 got 1
* Test: int add(int a, int b) { return a + b; }; int r = add(1, 2, 3);
Error: type-error: argument count mismatch in call to add: expected 2 got 3
* Test: int fact(int n) { if (n == 0) { return 1; } else { return n * fact(n - 1); } };
Typecheck result: #t
* Test: int add(int a, int b) { return a + b; }; int twice(int x){ return add(x,x); }; int r = twice(3);
Typecheck result: #t
* Test: int wrong() { return true; };
Typecheck result: #t
* Test: list a; $push(a, 1); $push(a, 2); $print($get(a, 0));
Typecheck result: #t
* Test: $push(b, 1);
Typecheck result: #t
* Test: list x; $print($get(x, 0));
Typecheck result: #t
* Test: list L; $push(L, 1); $set(L, 0, 5); $pop(L); $size(L);
Typecheck result: #t
* Test: string s = "hello"; list cs = $tocharlist(s);
Typecheck result: #t
* Test: char c = 'a'; int bad = c + 1;
Error: type-error: invalid '+' operands: char and int
* Test: int x = -5 + 10;
```

شکل ۱۱: بخشی از خروجی تست‌بنج تایپ‌چکر



```
Testing: "int count = 5; while (count > 0) { $print(count); count = count - 1; }"
5
4
3
2
1
Result: #(struct:num-val 0)

Testing: "int x = 10; while (x < 5) { $print(\"this should not print!\"); }"
Result: #(struct:num-val 0)

Testing: "bool running = true; int counter = 0; while (running) { $print(counter); counter = counter + 1; if (counter >= 3) { running = false; } }"
0
1
2
Result: #(struct:num-val 0)

Testing: "int outer = 0; while (outer < 2) { int inner = 0; while (inner < 2) { $print(outer * 10 + inner); inner = inner + 1; } outer = outer + 1; }"
0
1
10
11
Result: #(struct:num-val 0)

Testing: "int a = 1; int b = 10; while (a < b && a < 5) { $print(a); a = a + 1; }"
1
2
3
4
Result: #(struct:num-val 0)

Testing: "int sum = 0; int n = 1; while (n <= 4) { sum = sum + n; n = n + 1; } $print(sum);"
10
Result: #(struct:num-val 10)

***** -> control flow tests
Testing: "int i = 0; while (i < 5) { if (i % 2 == 0) { $print(\"even: \"); $print(i); } i = i + 1; }"
even:
0
even:
2
even:
4
Result: #(struct:num-val 0)

Testing: "bool condition = true; if (condition) { int j = 0; while (j < 3) { $print(j); j = j + 1; } }"
0
1
2
```

شکل ۱۲: بخشی از خروجی تست‌بنج ساده

```

Result: #(struct:num-val 0)
(num-val 0)
----- fib DP printer -----
Testing: "\n      int n = 10;\n      list dp;\n      $push(dp, 1);\n      $push(dp, 1);\n\n      int idx = 2;\n      int n = 10;\n      while (idx <= n) {\n      int ans = $get(dp, (idx - 1)) + $get(dp, (idx - 2));\n      $print(idx);\n      $print(\" : \");\n      $print(ans);\n      $print(\"-----\n      -----\");\n\n      idx = idx + 1;\n      $push(dp, ans);\n      }\n"
2
2
-----
3
:
3
-----
4
:
5
-----
5
:
8
-----
6
:
13
-----
7
:
21
-----
8
:
34
-----
9
:
55
-----
10
:
89
-----

```

شکل ۱۳: خروجی بخش ۱۰ فیوناتچی اول به روش بازگشتی تست‌های سخت

```

Result: #(struct:num-val 0)
(num-val 0)
----- fib DP printer -----
Testing: "\n      int n = 10;\n      list dp;\n      $push(dp, 1);\n      $push(dp, 1);\n\n      int idx = 2;\n      int n = 10;\n      while (idx <= n) {\n      int ans = $get(dp, (idx - 1)) + $get(dp, (idx - 2));\n      $print(idx);\n      $print(\" : \");\n      $print(ans);\n      $print(\"-----\n      -----\");\n\n      idx = idx + 1;\n      $push(dp, ans);\n      }\n"
2
2
-----
3
:
3
-----
4
:
5
-----
5
:
8
-----
6
:
13
-----
7
:
21
-----
8
:
34
-----
9
:
55
-----
10
:
89
-----

```

شکل ۱۴: خروجی بخش ۱۰ فیوناتچی اول به روش دیپی تست‌های سخت

[illegible]

شکل ۱۵: خروجی بخش تبدیل به آپریکس تست‌های سخت

```
Error: type-error: type mismatch in declaration of b: declared bool but initializer has int
# Test: $print(u);
Error: type-error: unbound identifier u
# Test: int x = 1; $print(y + u);
Error: type-error: unbound identifier y
# Test: int x = 5; x = x + 1;
Typecheck result: #t
# Test: int x = 5; x = "hello";
Error: type-error: assignment type mismatch for x: expected int got string
# Test: int a = 5; int b = 2; int c = a % b;
Typecheck result: #t
# Test: int a = 5; float b = 2.0; float c = a / b;
Typecheck result: #t
# Test: int a = 5; string s = "a"; int z = a + s;
Error: type-error: invalid '+' operands: int and string
# Test: int x = 1; bool t = x < 3;
Typecheck result: #t
# Test: string s = "a"; bool t = s == "b";
Typecheck result: #t
# Test: int x = 1; bool bad = x && true;
Error: type-error: && requires bools, got int and bool
# Test: int a = 3; int b = 4; int d = a & b;
Typecheck result: #t
# Test: float f = 1.0; int x = f & 1;
Error: type-error: bitwise operations require ints, got float and int
# Test: if (true) { int x = 1; }
Typecheck result: #t
# Test: if (1) { int x = 1; }
Error: type-error: if condition must be bool, got int
# Test: while (false) { int x = 0; }
Typecheck result: #t
# Test: while (0) { int x = 0; }
Error: type-error: while condition must be bool, got int
# Test: int id(int x) { return x; }; int r = id(5);
Typecheck result: #t
# Test: int id(int x) { return x; }; int r = id(true);
Error: type-error: argument type mismatch in call to id: expected int got bool
# Test: int add(int a, int b) { return a + b; }; int r = add(1);
Error: type-error: argument count mismatch in call to add: expected 2 got 1
# Test: int add(int a, int b) { return a + b; }; int r = add(1, 2, 3);
Error: type-error: argument count mismatch in call to add: expected 2 got 3
# Test: int fact(int n) { if (n == 0) { return 1; } else { return n * fact(n - 1); } };
Typecheck result: #t
# Test: int add(int a, int b) { return a + b; }; int twice(int x) { return add(x,x); }; int r = twice(3);
Typecheck result: #t
# Test: int wrong() { return true; };
Typecheck result: #t
# Test: list a; $push(a, 1); $push(a, 2); $print($get(a, 0));
Typecheck result: #t
# Test: $push(b, 1);
```

شکل ۱۶: بخشی از خروجی تست بنچ تایپ چکر