طراحی زبان‌های برنامه‌سازی

نیم‌سال دوم ۱۴۰۳-۱۴۰۴

مدرس: دکتر محمد ایزدی

دانشکده مهندسی کامپیوتر

گزارش پروژه امتیازی درس

نام و نام‌خانوادگی: علی صادقی

شماره دانشجویی: ۹۹۱۰۸۲۲۳

چکیده

در این گزارش، زبان برنامه‌نویسی طراحی‌شده، lexer، parser و گرامر، AST و AST-mapper، سامانه انواع و ارزیابی تنبل، محیط اجرا با حوزه‌بندی lexical، مدیریت حافظه و جمع‌آوری زباله mark-and-sweep GC، ارزیابی عبارات و دستورات، مدیریت خطا، و زیرساخت تست و runnerهای پروژه تشریح می‌شود. ساختار گزارش با الزامات فاز صفر (طراحی گرامر)، فاز یک (طراحی مفسر)، و فاز دو (کدنویسی در خود زبان) منطبق است.

فهرست مطالب

۱ مرور الزامات فازها و انطباق پیاده‌سازی ۳
۱.۱ فاز صفر: طراحی گرامر و ابزارهای نحوی ۳
۲.۱ فاز یک: طراحی مفسر ۳
۳.۱ فاز دو: برنامه‌نویسی در خود زبان ۳

۲ معماری کلی و فایل‌ها ۳

۳ لکسر (**lexer.rkt**) ۳
۱.۳ طراحی توکن‌ها و قواعد ۳

۴ پارسر و گرامر (**parser.rkt**) ۴
۱.۴ فلسفه گرامر ۴
۲.۴ گرامر مربوطه ۴
۳.۴ گزیده‌ای از گرامر (عین کد) ۷

۵ نگاشت **AST** (**ast-mapper.rkt**) ۸
۱.۵ ایده ۸
۲.۵ نمونه نگاشت‌ها ۸

۶ سامانه انواع، مقادیر و ارزیابی تنبل (**datatypes.rkt**) ۸
۱.۶ expval و type ۸
۲.۶ ارزیابی تنبل (call-by-need) ۸

۷ محیط اجرا و حوزه‌بندی (**environment.rkt**) ۹

۸ مدیریت حافظه و GC (**store.rkt**) ۹
۱.۸ ایده ۹

۹ ارزیابی عبارات و دستورات (**interpreter.rkt**) ۹
۱.۹ اعمالگرها و نظام عددی ۹
۲.۹ رشته و لیست: len و اندیس‌گذاری/انتساب اندیسی ۱۰
۳.۹ توابع، بستن‌ها، و تأخیر ارزیابی ۱۰
۴.۹ کنترل جریان: if/while/for، break/continue ۱۰

۱۰ مدیریت خطا ۱۰

۱۱ تایپ: صراحت، بررسی و استنتاج ۱۰
۱.۱۱ صراحت و سازگاری ۱۰
۲.۱۱ استنتاج ۱۰

۱۲ زیرساخت تست و رانرها ۱۱
۱.۱۲ پوشش تست‌ها (tests/) ۱۱
۲.۱۲ حل‌های فاز دو (solutions/) ۱۱

۱۳ راهنمای اجرا برای مصحح ۱۱

۱۴ جمع‌بندی و نکات طراحی ۱۱

۱ مرور الزامات فازها و انطباق پیاده‌سازی

۱.۱ فاز صفر: طراحی گرامر و ابزارهای نحوی

مطابق صورت‌مسأله، باید CFG برای اجزای اصلی (تعریف متغیر، انواع int/float/string/bool/list/nulltype/void، کنترل جریان با if/else، حلقه‌ها while/for، عملگرهای مقایسه و منطق، پرانتزگذاری، و دستور print) تعریف و سپس lexer و parser مبتنی بر Racket parser-tools پیاده شود. در این پروژه، فایل‌های lexer.rkt و parser.rkt دقیقاً همین نقش را ایفا می‌کنند.

۲.۱ فاز یک: طراحی مفسر

الزامات شامل تولید AST از خروجی parser، توابع ارزیابی برای هر نوع گره (expression/statement/function call و غیره)، محیط اجرا (runtime environment) با پشتیبانی از lexical scoping، کنترل جریان اجرا، مدیریت خطا در زمان اجرا، و امکان type checking/strictness/inference بود. این موارد به‌ترتیب در datatypes.rkt، ast-mapper.rkt، environment.rkt، interpreter.rkt و store.rkt پیاده شده‌اند.

۳.۱ فاز دو: برنامه‌نویسی در خود زبان

مطابق صورت‌مسأله، پنج سؤال عملی لازم است در خود زبان حل شوند. برای سهولت ارزیابی، پاسخ‌ها در پوشه solutions با فایل‌های solution1.txt تا solution5.txt قرار گرفته‌اند، و یک solution-runner.rkt برای اجرای انتخابی هر سؤال تعبیه شده است (جزئیات در بخش ۱۲).

۲ معماری کلی و فایل‌ها

- lexer.rkt: تعریف توکن‌ها و lexer با ردیابی شماره خط و پشتیبانی از string escape.
- parser.rkt: گرامر کامل زبان و ساخت AST سطح نحوی (خروجی لیستی).
- datatypes.rkt: تعریف AST تایپ‌شده، expval، type و توابع force/convert/extract.
- ast-mapper.rkt: نگاشت AST نحوی به AST تایپ‌شده‌ی ماژول datatypes.
- environment.rkt: محیط اجرا و lexical scoping بر مبنای EoPL datatypes.
- store.rkt: حافظه برداری، free-list، و mark-and-sweep GC.
- interpreter.rkt: ارزیابی program/statement/expression، اعمالگرها، call-by-need با thunk و مدیریت حلقه‌ها/پرش‌ها.
- پوشه tests: شامل test##.txt که هریک بخشی از الزامات را می‌پوشانند، به‌همراه tst-runner.rkt.
- پوشه solutions: شامل solution1..5.txt و solution-runner.rkt برای فاز دو.

۳ لکسر (lexer.rkt)

۱.۳ طراحی توکن‌ها و قواعد

```
(define-tokens value-tokens
  (INT FLOAT STRING ID TRUE FALSE NULL EOF))

(define-empty-tokens keyword-tokens
  (VAR FUNC IF ELSE WHILE FOR PRINT RETURN BREAK CONTINUE LEN
```

```
    INTTYPE FLOATTYPE STRINGTYPE BOOLTYPE LISTTYPE NULLTYPE VOID))

(define-empty-tokens symbol-tokens
  (SEMICOLON COLON COMMA OP CP OB CB OCB CCB
   ASSIGNMENT PLUS MINUS TIMES DIVIDE INT-DIVIDE MODULO POWER
   LT GT LET GET EQUALS NOTEQUALS AND OR NOT))
```

پشتیبانی از اعداد صحیح و اعشاری، شناسه‌ها، کلیدواژه‌ها، عملگرهای حسابی/منطقی/مقایسه‌ای و نشانه‌گذارها فراهم است. رشته‌ها با escapeهای \n، \"، و \\ پردازش می‌شوند. شمارنده خط برای پیام‌های خطا به‌روز می‌شود.

۴ پارسر و گرامر (parser.rkt)

۱.۴ فلسفه گرامر

گرامر به‌صورت بالا به پایین و نزدیک به ساختارهای آشنا (C-like) طراحی شده است؛ Program شامل StatementList، و Statementها شامل تعریف/انتساب/بلوک/فراخوانی/چاپ/بازگشت/پرش/شرط/حلقه هستند. انواع صریح Type و نیز لیست‌های پارامتریک list<T> پوشش داده شده‌اند.

۲.۴ گرامر مربوطه

```
        Program ::= StatementList

StatementList ::= Statement StatementList
                | lambda

Statement ::= OtherStatement
            | IfStatement

OtherStatement ::= VariableDeclaration
                 | FunctionDeclaration
                 | Assignment
                 | LoopStatement
                 | Block
                 | FunctionCallStmt
                 | PrintStatement
                 | ReturnStatement
                 | BreakStatement
                 | ContinueStatement

BreakStatement ::= 'break' ';'

ContinueStatement ::= 'continue' ';'

Block ::= '{' StatementList '}'

VariableDeclaration ::= 'var' Identifier ':' Type '=' Expression ';'

Type ::= 'int'
       | 'float'
       | 'string'
       | 'bool'
       | 'list' '[' Type ']'
```

```
        | 'void'
        | 'nulltype'

FunctionDeclaration ::= 'func' Identifier '(' ParameterList ')' ':' Type
    Block

ParameterList ::= ParameterListNonEmpty
                | lambda

ParameterListNonEmpty ::= Parameter
                        | Parameter ',' ParameterListNonEmpty

Parameter ::= Identifier ':' Type

Assignment ::= Identifier '=' Expression ';'

IfStatement ::= MatchedIf
              | UnmatchedIf

MatchedIf ::= 'if' '(' Expression ')' Block 'else' Block

UnmatchedIf ::= 'if' '(' Expression ')' Block
              | 'if' '(' Expression ')' MatchedIf 'else' UnmatchedIf

LoopStatement ::= WhileLoop
                | ForLoop

WhileLoop ::= 'while' '(' Expression ')' Block

ForLoop ::= 'for' '(' ForAssignment ';' Expression ';' ForAssignment ')'
    Block

ForAssignment ::= Identifier '=' Expression

FunctionCallExpr ::= Identifier '(' ArgumentList ')'

FunctionCallStmt ::= FunctionCallExpr ';'

ArgumentList ::= ArgumentListNonEmpty
               | lambda

ArgumentListNonEmpty ::= Expression
                       | Expression ',' ArgumentListNonEmpty

PrintStatement ::= 'print' '(' Expression ')' ';'

ReturnStatement ::= 'return' Expression ';'
                  | 'return' ';'
                  | lambda

Expression ::= LogicalExpression

LogicalExpression ::= ComparativeExpression
```

```
                    | LogicalExpression LogicalOperator
                      ComparativeExpression
                    | 'not' ComparativeExpression

ComparativeExpression ::= ArithmeticExpression
                        | ArithmeticExpression ComparativeOperator
                          ArithmeticExpression

ArithmeticExpression ::= Term
                       | ArithmeticExpression AdditiveOperator Term

Term ::= Power
       | Term MultiplicativeOperator Power

Power ::= Factor
        | Factor '^' Power

Factor ::= Identifier
         | Literal
         | '(' Expression ')'
         | FunctionCallExpr
         | UnaryOperator Factor
         | 'NULL'

Literal ::= IntegerLiteral
         | FloatLiteral
         | StringLiteral
         | ListLiteral
         | 'true'
         | 'false'

ListLiteral ::= '[' ExpressionList ']'

ExpressionList ::= ExpressionListNonEmpty
                 | lambda

ExpressionListNonEmpty ::= Expression
                         | Expression ',' ExpressionListNonEmpty

ComparativeOperator ::= '<' | '>' | '<=' | '>=' | '==' | '!='

LogicalOperator ::= '&&' | '||' | 'and' | 'or'

AdditiveOperator ::= '+' | '-'

MultiplicativeOperator ::= '*' | '/' | '%'

UnaryOperator ::= '-' | '!' | 'not'

IntegerLiteral ::= Digit+
FloatLiteral ::= Digit+ '.' Digit+
StringLiteral ::= '"' (Character)* '"'
```

۳.۴ گزیده‌ای از گرامر (عین کد)

```
(start Program) (end EOF)
(tokens value-tokens keyword-tokens symbol-tokens)

[Program         [(StatementList) (list 'Program $1)]]
[StatementList   [(Statement StatementList) (cons $1 $2)]
                 [() '()]]

[Statement [(OtherStatement) $1] [(IfStatement) $1]
           [(LoopStatement) $1] [(IndexAssignment) $1]]

[VariableDeclaration
  [(VAR ID COLON Type ASSIGNMENT Expression SEMICOLON)
   (list 'VarDecl $2 $4 $6)]
  [(VAR ID ASSIGNMENT Expression SEMICOLON)
   (list 'VarDeclInfer $2 $4)]]

[Type [(INTTYPE) "int"] [(FLOATTYPE) "float"] [(STRINGTYPE) "string"]
      [(BOOLTYPE) "bool"]
      [(LISTTYPE OB Type CB) (list 'list $3)]
      [(LISTTYPE LT Type GT) (list 'list $3)]
      [(NULLTYPE) "nulltype"] [(VOID) "void"]]

[FunctionDeclaration
  [(FUNC ID OP ParameterList CP COLON Type Block)
   (list 'FuncDecl $2 $4 $7 $8)]]

[IfStatement
  [(IF OP Expression CP Block ELSE Block) (list 'If $3 $5 $7)]
  [(IF OP Expression CP Block) (list 'If $3 $5 'None)]]

[WhileLoop [(WHILE OP Expression CP Block) (list 'While $3 $5)]]

[ForLoop
 [(FOR OP ForAssignment SEMICOLON Expression SEMICOLON ForAssignment CP
    Block)
  (list 'For $3 $5 $7 $9)]]

[FunctionCallExpr
  [(ID OP ArgumentList CP) (list 'Call $1 $3)]
  [(LEN OP Expression CP) (list 'Len $3)]]

[ComparativeOperator
  [(LT) '<] [(GT) '>] [(LET) '<=] [(GET) '>=] [(EQUALS) '==] [(NOTEQUALS)
      '!=']]

[AdditiveOperator [(PLUS) '+] [(MINUS) '-]]
[MultiplicativeOperator [(TIMES) '*] [(DIVIDE) '/] [(MODULO) '%] [(INT-
   DIVIDE) '//]]
```

نکته طراحی: برای دسترسی اندیسی PostfixExpr OB Expression CB و IndexAssignment پیش‌بینی شده تا هم a[i] و هم انتساب a[i]=... پوشش داده شوند.

۵ نگاشت AST (ast-mapper.rkt)

۱.۵ ایده

خروجی parser یک AST نحویِ سبک‌وزن (لیستی) است. ماژول ast-mapper آن را به AST تایپ‌شده (datatypes.rkt) نگاشت می‌کند تا ارزیابی و type checking تمیز و توسعه‌پذیر باشد.

۲.۵ نمونه نگاشت‌ها

```
[(list 'VarDecl id type expr)
 (var-decl-stmt (symbol->string id) (convert-type type) (ast->expr expr))
    ]

[(list 'Call fname args)
 (call-exp (symbol->string fname) (map ast->expr args))]

[(list 'Index base idx)
 (index-exp (ast->expr base) (ast->expr idx))]

[(list 'If cond then else)
 (if-stmt (ast->expr cond)
          (ast->stmt then)
          (if (or (eq? else #f) (eq? else 'None)) #f (ast->stmt else)))]
```

۶ سامانه انواع، مقادیر و ارزیابی تنبل (datatypes.rkt)

۱.۶ expval و type

```
(define-datatype expval expval?
  (num-val (num number?))
  (float-val (float number?))
  (bool-val (bool boolean?))
  (string-val (string string?))
  (list-val (vals (list-of expval?)))
  (null-val) (void-val)
  (thunk-val (thunk procedure?) (memo box?))
  (closure-val (params (list-of param?))
               (body statement?) (env environment?)))

(define-datatype type type?
  (int-type) (float-type) (string-type) (bool-type)
  (list-type (element-type type?)) (null-type) (void-type))
```

۲.۶ ارزیابی تنبل (call-by-need)

آرگومان‌های توابع به‌صورت thunk ذخیره و در مصرف، با force-expval ارزیابی و memoize می‌شوند. این کار سربار محاسبات غیرضروری را کاهش می‌دهد و با طراحی len/index نیز سازگار است.

```
(define (force-expval v)
  (cases expval v
```

```
    (thunk-val (thunk memo)
      (let ((cached (unbox memo)))
        (if cached cached (let ((res (thunk))) (set-box! memo res) res)))
        )
    (else v)))
```

۷ محیط اجرا و حوزه‌بندی (environment.rkt)

محیط اجرا با داده‌نوع empty-environment | extend-environment و apply-env پیاده شده و lexical scoping را تضمین می‌کند؛ بستن تابع (closure) محیط تعریف را در خود نگه می‌دارد.

```
(cases environment env
  (empty-environment () #f)
  (extend-environment (saved-var val saved-env)
    (if (equal? var saved-var) val (apply-env var saved-env))))
```

۸ مدیریت حافظه و GC (store.rkt)

۱.۸ ایده

حافظه به‌صورت برداری از entry = (cons value marked?) نگهداری می‌شود، با free-list برای بازیابی خانه‌های آزاد. gc! با الگوی mark-and-sweep پیاده شده: ابتدا reset-marks!، سپس mark-env و mark-expval برای علامت‌گذاری دسترس‌پذیرها، و نهایتاً sweep! برای آزادسازی بقیه.

```
(define (gc! root-env)
  (when (and (vector? the-store) (> store-length 0))
    (reset-marks!)    ; mark bit := #f
    (mark-env root-env)
    (sweep!)
    #t))
```

نکته‌ی طراحی: closureها محیط را علامت می‌زنند و thunkها در صورت memoized شدن، مقدار ذخیره‌شده را علامت می‌زنند؛ بدین‌ترتیب چرخه‌های قابل‌رسیدن حفظ و بقیه جمع‌آوری می‌شوند.

۹ ارزیابی عبارات و دستورات (interpreter.rkt)

۱.۹ اعمالگرها و نظام عددی

```
(case op
  [(+) (num-val (+ l r))]
  [(/) (if (zero? r) (eopl:error 'division-by-zero "Division by zero")
           (float-val (/ (exact->inexact l) r)))]
  [(//) (num-val (quotient l r))]
  [(%) (num-val (remainder l r))]
  [(==) (bool-val (= l r))] ... )
```

تقسیم / خروجی float-val و تقسیم صحیح // خروجی num-val می‌دهد. سازگاری نوعی بین int/float با types-compatible? لحاظ شده است (ارتقای ضمنی).

۲.۹ رشته و لیست: **len** و اندیس‌گذاری/انتساب اندیسی

```
(len-exp ...) ;      list   string
)index-exp ...) ;
)index-assign-stmt ...) ;
```

انتساب اندیسی برای list با بازسازی عنصر هدف و برای string با جایگزینی کاراکتر (از اولین نویسه‌ی رشته‌ی جایگزین) پیاده شده است.

۳.۹ توابع، بستن‌ها، و تأخیر ارزیابی

```
)closure-val (params body env))
;        thunk        env    bind    :
)map (lambda (arg) (make-thunk (lambda () (eval-expression arg env))))
     args)
```

۴.۹ کنترل جریان: **break/continue**، **if/while/for**

با سه پرچم ?continue-flag/break-flag/in-loop، اجرای while/for مدیریت و پرش‌ها به‌شکل امن هندل می‌شوند.

۱۰ مدیریت خطا

پیام‌های eopl:error معنادار و همراه با نوع و (در صورت نیاز) شماره خط هستند؛ از جمله: unbound-variable، type-error، division-by-zero، index-out-of-range، arity-mismatch، not-a-function، و خطای break/continue خارج از حلقه. این با رهنمودهای فاز یک کاملاً منطبق است.

۱۱ تایپ: صراحت، بررسی و استنتاج

۱.۱۱ صراحت و سازگاری

تعریف متغیرها می‌تواند با نوع صریح (var x: int = ...) یا با استنتاج (var x = ...) باشد. تابع check-type در var-decl-stmt اعمال می‌شود. سازگاری ضمنی int/float در محاسبات پشتیبانی می‌شود.

۲.۱۱ استنتاج

```
(define (infer-type val)
  (cases expval val
    (num-val ...) (float-val ...) (bool-val ...) (string-val ...)
    (list-val (vals) (if (null? vals) (list-type (int-type))
                         (list-type (infer-type (car vals))))) ...))
```

برای لیست تهی، یک سیاست محافظه‌کارانه اتخاذ شده است (به‌صورت پیش‌فرض list<int>) تا اجرای تست‌ها ساده و قابل‌پیش‌بینی بماند.[1]

[1] در نسخه‌های بعدی می‌توان list<_> تهی را با نوع عنصری «نامقیّد» مدل کرد.

۱۲ زیرساخت تست و رانرها

۱.۱۲ پوشش تست‌ها (tests/)

پوشه tests شامل فایل‌های test#.txt است که هر کدام بخشی از الزامات پروژه را پوشش می‌دهند (از گرامر تا ارزیابی و خطا). برای اجرای انتخابی هر تست، فایل tst-runner.rkt در ریشه پروژه تعبیه شده است؛ مصحح کافی‌ست فقط شماره را عوض کند:

```
;; in tst-runner.rkt
(define test-file "tests/test10.txt") ;
```

۲.۱۲ حل‌های فاز دو (solutions/)

برای سؤالات فاز دو، پاسخ‌ها در پوشه solutions با نام‌های solution1.txt تا solution5.txt قرار دارند. یک رانر مجزا برای آن‌ها در ریشه پروژه آمده است:

```
;; in solution-runner.rkt
(define test-file "solutions/solution1.txt") ;
```

الگوی فوق، ارزیابی گزینشی را برای هر سؤال (شماره ۱ تا ۵) تسهیل می‌کند.

۱۳ راهنمای اجرا برای مصحح

۱. یکی از تست‌ها را در tst-runner.rkt با تغییر خط (define test-file "...") انتخاب کنید و اجرا بگیرید.

۲. برای فاز دو، به‌طور مشابه در solution-runner.rkt شماره solution را تغییر دهید.

۳. پیام‌های خطا و خروجی print طبق قراردادهای فاز یک/دو تولید می‌شوند.

۱۴ جمع‌بندی و نکات طراحی

- سادگی نحو با قدرت بیان کافی: گرامر نزدیک به زبان‌های آشنا و در عین حال دارای list<T> و توابع با نوع بازگشتی صریح است.
- ارزیابی تنبل: call-by-need برای آرگومان‌های توابع با thunk+memo پیاده شده تا کارایی بهبود یابد.
- محیط و GC ماژولار: environment/store جداگانه اجازهٔ توسعهٔ آسان (مثلاً افزودن objects/records) را می‌دهد.
- توسعه‌های آینده (ایده):
  - pattern matching برای ساختارهای داده،
  - option/result types برای خطاهای قابل‌مدیریت،
  - بهبود سیاست نوع لیست تهی با نوع عنصری نامقیّد،
  - بهینه‌سازی GC با generational GC (در نسخه فعلی mark-and-sweep کاملاً کافی‌ست).

ضمیمه‌ها (گزیده کدها)

تعریف انواع و پارامتر تابع

```
)define-datatype param param?
  (a-param (name string?) (param-type type?)))
```

الگوی انتساب اندیسی تو در تو

```
(define (assign-index-rec lvalue-expr idx new-val env)
  (cases expression lvalue-expr
    (var-exp (var) ... )
    (index-exp (base-expr inner-idx-expr)
      (let* ((i (expval->num (eval-expression inner-idx-expr env)))
             (base-val (force-expval (eval-expression base-expr env))))
        (cond
          [(list-val? base-val) ...]
          [(string-val? base-val) ...]
          [else (eopl:error 'type-error "Left-hand side is not indexable
            ")])))))
```

هستهٔ GC

```
(define (mark-expval v)
  (cases expval v
    (list-val (vals) (for-each mark-expval vals))
    (closure-val (params body env) (mark-env env))
    (thunk-val (thunk memo) (let ((c (unbox memo))) (when c (mark-expval
      c))))
    (else #f)))
```

گزیده‌ای از گرامر (چاپ و بازگشت)

```
[PrintStatement   [(PRINT OP ExpressionList CP SEMICOLON) (list 'Print $3)
   ]]
[ReturnStatement [(RETURN Expression SEMICOLON) (list 'Return $2)]
                 [(RETURN SEMICOLON) (list 'Return 'None)]]
```

سخن آخر: تقریباً تمام بخش‌های خواسته‌شده مطابق فازها پیاده‌سازی شده‌اند و پوشش تستی گسترده در tests/ قرار دارد. در مواردی که بنا به سادگی نسخهٔ فعلی، سیاست‌های محافظه‌کارانه اتخاذ شده (مثلاً نوع لیست تهی)، مسیر توسعهٔ آینده روشن و در گزارش تشریح شد تا معیارهای ارزیابی پروژه به بهترین نحو برآورده شوند.