GPP Interpreter'ın Analizi ve Kullanımı

Bu rapor, sağlanan Lisp kodunun genel yapısını, işleyiş mantığını ve örnek giriş-çıkış senaryolarını açıklamaktadır. Kod, GPP adlı özel bir programlama dili için bir yorumlayıcı (interpreter) işlevi görmektedir. Aşağıda, kodun bileşenleri, işleyişi ve kullanımına dair detaylı bir analiz sunulmaktadır.

İçindekiler

Giriş

Sağlanan Lisp kodu, GPP adlı özel bir programlama dilini işlemek üzere tasarlanmış bir yorumlayıcıdır. Bu yorumlayıcı, GPP kodlarını okuyarak tokenize eder, sözdizimini analiz eder, bir soyut sözdizim ağacı (AST) oluşturur ve ardından bu AST'yi değerlendirerek sonuç

üretir. Ayrıca, hata yönetimi ve değişken/fonksiyon tabloları gibi önemli bileşenleri de içermektedir.

Genel Yapı

Kod, üç ana bileşenden oluşmaktadır:

1. **Tokenizasyon (Lexical Analysis):** Giriş metnini alır ve anlamlı bileşenlere (tokenlara) ayırır.
2. **Sözdizimi Analizi (Parsing):** Tokenları kullanarak bir AST oluşturur.
3. **Değerlendirme (Evaluation):** AST'yi değerlendirir ve sonuç üretir.

Ayrıca, global durum tabloları, hata yönetimi ve yardımcı fonksiyonlar da bu yapının bir parçasıdır.

Bileşenlerin Detaylı Açıklaması

Global Durum ve Tablo Tanımlamaları

lisp
Copy code
```
;; Global parser state
(defvar *current-token-index* 0)
(defvar *tokens* '())
(defvar *values* '())
(defvar *identifiers* '())

;; Ortam (environment) ve fonksiyon tabloları:
(defvar *variables* (make-hash-table :test 'equal))
(defvar *functions* (make-hash-table :test 'equal))
```

- ***current-token-index*:** Token listesi üzerinde ilerlemek için kullanılan indeks.
- ***tokens*, *values*, *identifiers*:** Tokenizasyon sürecinde elde edilen tokenlar, değerler ve tanımlayıcılar.
- ***variables*, *functions*:** Değişkenler ve fonksiyonlar için hash tabloları, ortam olarak görev yapar.

Hata Yönetimi

lisp
Copy code
(define-condition syntax-error (error)
  ((message :initarg :message :initform "Syntax error occurred." :accessor syntax-error-message)))

- **syntax-error:** Sözdizimi hatalarını temsil eden özel bir hata koşulu tanımlanmıştır.

Tokenizasyon

Tokenizasyon, giriş metnini anlamlı bileşenlere ayırma sürecidir.

getTokenList Fonksiyonu

lisp
Copy code
(defun getTokenList (line)
  (setq tokenList '())
  (setq valueList '())
  (setq identifierList '())
  (setq input '())
  (split-line line)
  ;; Liste sıralarını düzelt
  (setq tokenList (reverse tokenList))
  (setq valueList (reverse valueList))
  (setq identifierList (reverse identifierList))
  (setq input (reverse input))
)

- **İşlevi:** Giriş satırını alır, boş listeleri sıfırlar ve split-line fonksiyonunu çağırarak satırı tokenlara ayırır.
- **Liste Sıraları:** split-line fonksiyonu sırasında listelere eleman eklerken ters sırada eklenir, bu yüzden sonunda listeler ters çevrilir.

split-line ve check-tokens Fonksiyonları

lisp
Copy code
```
(defun split-line (string)
 "Split given string from spaces into words"
 (setq string (custom-left-trim string)) ; Remove leading spaces
 (let ((words '()) (start nil))
  (loop for i from 0 to (1- (length string))
      do (if (char= (char string i) #\Space)
          (if start
            (progn
             (push (subseq string start i) words)
             (setf start nil)))
          (unless start (setf start i)))
      finally (if start (push (subseq string start) words)))

  (setq words (reverse words))

  (dolist (word words)
   (if (equal (check-tokens word) 1)
     (return)))
  'done))
```

- **İşlevi:** Giriş string'ini boşluklara göre kelimelere böler ve her kelimeyi check-tokens fonksiyonuna gönderir.
- **check-tokens:** Her kelimeyi analiz ederek uygun tokenları belirler ve ilgili listelere ekler.

Sözdizimi Analizi (Parsing)

Parsing aşaması, token listesini alır ve dilin sözdizimine uygun olup olmadığını kontrol eder, ardından bir AST oluşturur.

parse-start ve İlgili Fonksiyonlar

lisp
Copy code

```
(defun parse-start ()
  "Start rule: Parse an EXPLIST"
  (parse-explist))

(defun parse-explist ()
  "EXPLIST -> $EXP $EXPLIST | $EXP"
  (let ((first (parse-exp)))
    (if first
        (let ((rest (parse-explist)))
          (if rest
              (cons first rest)
            (list first)))
      nil)))
```

- **parse-start:** Parsing işleminin başlangıç noktasıdır, EXPLIST'i parse eder.
- **parse-explist:** Birden fazla ifade (EXP) listesini parse eder.

parse-exp ve parse-paren-expression

```
lisp
Copy code
(defun parse-exp ()
  "EXP -> (OPERATOR EXP EXP) | (if EXPB EXPLIST) | ... | IDENTIFIER | VALUEF | FCALL"
  (let ((tk (next-token)))
    (cond
     ((string= tk "OP_OP")
      (advance-token) ;; '('
      (parse-paren-expression))

     ((string= tk "IDENTIFIER")
      (advance-token)
      (let ((id (car *identifiers*)))
        (setq *identifiers* (cdr *identifiers*))
        id))

     ((string= tk "VALUEF")
      (advance-token)
      (let ((val (car *values*)))
```

```
    (setq *values* (cdr *values*))
    val))

  (t nil))))
```

- **parse-exp:** Bir ifadeyi parse eder. İfade bir parantezli ifade, tanımlayıcı, değer veya fonksiyon çağrısı olabilir.
- **parse-paren-expression:** Parantez içinde yer alan ifadeleri ayrıntılı olarak parse eder (aritmetik operatörler, if, while, for, set, defvar, deffun, fonksiyon çağrıları vb.).

Soyut Sözdizim Ağacı (AST) Değerlendirme

AST, parse edilen ifadelerin hiyerarşik temsilidir ve değerlendirme aşamasında kullanılır.

evaluate-ast Fonksiyonu

```
lisp
Copy code
(defun evaluate-ast (ast)
 (cond
  ((stringp ast)
   (if (find #\: ast)
     ast
    (lookup-variable ast)))

  ((consp ast)
   (case (car ast)
    (DEFFUN
     (define-function (cadr ast) (caddr ast) (cadddr ast))
     "0:1")

    (FCALL
     (apply-function (cadr ast) (caddr ast)))

    ;; Diğer durumlar...
    )))

  ;; Hatalar ve diğer durumlar...
```

))

- **İşlevi:** AST'yi değerlendirir. AST'nin türüne (string veya list) bağlı olarak değişkenleri lookup eder veya fonksiyonları uygular.
- **Durumlar:** DEFFUN, FCALL, SET, DEFVAR, IF, WHILE, FOR, EQUAL, LESS, OP_PLUS, OP_MINUS, OP_MULT, OP_DIV gibi işlemleri destekler.

Diğer Değerlendirme Fonksiyonları

- **evaluate-explist:** Bir ifade listesini sırayla değerlendirir.
- **evaluate-boolean:** Boolean ifadeleri değerlendirir (TRUE, FALSE, EQUAL, LESS).
- **Aritmetik Operatör Fonksiyonları:** op-plus-fn, op-minus-fn, op-mult-fn, op-div-fn gibi fonksiyonlar, kesirli değerler üzerinde aritmetik işlemler yapar.

Yardımcı Fonksiyonlar

- **fraction-to-float ve split-fraction:** Kesirli değerleri işlemek için kullanılır.
- **lookup-variable, set-variable, defvar-variable:** Değişkenleri yönetir.
- **define-function, apply-function:** Fonksiyon tanımlama ve uygulama işlemlerini gerçekleştirir.
- **while-loop, for-loop:** Döngü yapıları için değerlendirme sağlar.

İşleyiş Mantığı

Yorumlayıcının işleyişi, genellikle şu adımlarla gerçekleşir:

1. **Giriş Alma (Read):** Kullanıcıdan bir satır girişi alınır.
2. **Tokenizasyon (Tokenize):** Giriş satırı tokenlara ayrılır.
3. **Sözdizimi Analizi (Parse):** Tokenlar, dilin sözdizimine uygun olarak analiz edilir ve AST oluşturulur.
4. **Değerlendirme (Evaluate):** AST, yorumlayıcı tarafından değerlendirilir ve sonuç üretilir.
5. **Çıkış (Print):** Sonuç kullanıcıya gösterilir.
6. **Döngü (Loop):** Kullanıcı "exit" yazana kadar süreç tekrarlanır.

Read-Eval-Print Döngüsü

lisp

```
Copy code
(defun gppinterpreter ()
  "READ-EVAL-PRINT loop"
  (loop
   (format t "~%>>> ")
   (let ((line (read-line *standard-input* nil)))
     (if (or (null line) (string= "exit" line))
         (return)
       (progn
         (getTokenList line)
         (format t "Tokens: ~a~%" tokenList)
         (format t "Values: ~a~%" valueList)
         (format t "Identifiers: ~a~%" identifierList)

         (let ((syntax-check (check-function-syntax tokenList)))
           (format t "~a~%" syntax-check))

         (setq *tokens* tokenList)
         (setq *values* valueList)
         (setq *identifiers* identifierList)
         (setq *current-token-index* 0)

         (handler-case
             (let ((ast (parse-start)))
               (if ast
                   (progn
                     (format t "Grammar is correct.~%")
                     (let ((result (evaluate-explist ast)))
                       (format t "RESULT: ~a~%" result)))
                 (format t "No expressions found.~%")))
           (syntax-error (e)
             (format t "Syntax Error: ~a~%" (syntax-error-message e))
             (format t "RESULT: NIL~%")))))))))
```

- **Giriş Alımı:** Kullanıcıdan bir satır girişi alınır.
- **Tokenizasyon:** getTokenList ile giriş satırı tokenize edilir.
- **Sözdizimi Kontrolü:** check-function-syntax ile temel sözdizimi kontrolü yapılır.
- **Parsing ve AST Oluşturma:** parse-start ile AST oluşturulur.

- **Değerlendirme:** evaluate-explist ile AST değerlendirilir ve sonuç üretilir.
- **Hata Yönetimi:** Sözdizimi hataları handler-case ile yakalanır ve raporlanır.
- **Döngü:** Kullanıcı "exit" yazana kadar süreç tekrarlanır.

Tokenizasyon Süreci

Giriş satırı, boşluklara göre kelimelere ayrılır ve her kelime, belirli kurallara göre tokenlara dönüştürülür. Örneğin:

- **Operatörler:** + → OP_PLUS, - → OP_MINUS, * → OP_MULT, / → OP_DIV
- **Parantezler:** (→ OP_OP,) → OP_CP
- **Anahtar Kelimeler:** if → KW_IF, while → KW_WHILE, for → KW_FOR, deffun → KW_DEFFUN vb.
- **Değerler:** Kesirli değerler N:D formatında, örneğin 3:4 → VALUEF
- **Tanımlayıcılar:** Harf veya _ ile başlayan kelimeler → IDENTIFIER

Sözdizimi Analizi ve AST Oluşturma

Parse aşamasında, token listesi üzerinde ileri geri hareket edilerek ifadelerin yapısı kontrol edilir ve uygun şekilde AST oluşturulur. Örneğin:

- **Aritmetik İfade:** (+ 3:4 5:6) → (OP_PLUS "3:4" "5:6")
- **Fonksiyon Tanımlama:** (deffun add (x y) (+ x y)) → (DEFFUN "add" ("x" "y") ((OP_PLUS "x" "y")))
- **If İfadesi:** (if (equal x y) (print x) (print y)) → (IF (EQUAL "x" "y") (FCALL "print" ("x")) (FCALL "print" ("y")))

AST Değerlendirme ve Sonuç Üretme

Değerlendirme aşamasında, oluşturulan AST'ye göre işlemler gerçekleştirilir:

- **Aritmetik İşlemler:** OP_PLUS, OP_MINUS, OP_MULT, OP_DIV gibi işlemler, kesirli değerler üzerinde hesaplamalar yapar.
- **Fonksiyonlar:** DEFFUN ile fonksiyon tanımlanır, FCALL ile fonksiyonlar çağrılır.
- **Kontrol Yapıları:** IF, WHILE, FOR gibi yapılar koşullara bağlı olarak ifadeleri değerlendirir.
- **Değişken Yönetimi:** SET, DEFVAR ile değişkenler tanımlanır ve değerler atanır.

Örnek Giriş ve Çıkış Senaryoları

Aşağıda, GPP yorumlayıcısında çalıştırılabilecek bazı örnek girişler ve beklenen çıktılar sunulmuştur.

Değişken Tanımlama ve Kullanma

Giriş:

scss
Copy code
(defvar x 3:4)

Beklenen Çıkış:

vbnet
Copy code
>>> (defvar x 3:4)
Tokens: (KW_DEFVAR IDENTIFIER VALUEF OP_CP)
Values: (3:4)
Identifiers: (x)
Syntax Error: Function definition syntax is valid.
Grammar is correct.
RESULT: 0:1

Açıklama:

- defvar ile x adlı bir değişken tanımlanır ve değeri 3:4 olarak atanır.
- Değişken başarılı bir şekilde tanımlandığı için sonuç 0:1 olarak döner.

Giriş:

arduino
Copy code
(set x 5:6)

Beklenen Çıkış:

vbnet
Copy code
>>> (set x 5:6)
Tokens: (KW_SET IDENTIFIER VALUEF OP_CP)
Values: (5:6)
Identifiers: (x)
Syntax Error: Function call syntax is valid.
Grammar is correct.
RESULT: 5:6

Açıklama:

- set ile x değişkeninin değeri 5:6 olarak güncellenir.
- Güncelleme başarılı olduğu için sonuç 5:6 olarak döner.

Fonksiyon Tanımlama ve Çağırma

Giriş:

css
Copy code
(deffun add (a b) (+ a b))

Beklenen Çıkış:

vbnet
Copy code
>>> (deffun add (a b) (+ a b))
Tokens: (KW_DEFFUN IDENTIFIER OP_OP IDENTIFIER IDENTIFIER OP_CP OP_OP OP_PLUS IDENTIFIER IDENTIFIER OP_CP OP_CP)
Values: ()
Identifiers: (add a b + a b)
Syntax Error: Function definition syntax is valid.
Grammar is correct.
RESULT: 0:1

Açıklama:

- deffun ile add adlı bir fonksiyon tanımlanır, iki parametre alır (a, b) ve bu parametrelerin toplamını döner.
- Fonksiyon tanımlaması başarılı olduğu için sonuç 0:1 olarak döner.

Giriş:

csharp
Copy code
(add 3:4 5:6)

Beklenen Çıkış:

vbnet
Copy code
>>> (add 3:4 5:6)
Tokens: (OP_OP IDENTIFIER VALUEF VALUEF OP_CP)
Values: (3:4 5:6)
Identifiers: (add)
Syntax Error: Function call syntax is valid.
Grammar is correct.
RESULT: 8:10

Açıklama:

- add fonksiyonu çağrılır ve 3:4 ile 5:6 kesirleri toplanır.
- Sonuç olarak 8:10 döner.

Kontrol Yapıları (if, while, for)

If İfadesi

Giriş:

scss
Copy code
(if (equal x 5:6) (print x) (print 0:1))

Beklenen Çıkış:

vbnet
Copy code
>>> (if (equal x 5:6) (print x) (print 0:1))
Tokens: (KW_IF OP_OP KW_EQUAL IDENTIFIER VALUEF OP_CP OP_OP IDENTIFIER OP_CP OP_OP IDENTIFIER OP_CP OP_CP)
Values: (5:6 0:1)
Identifiers: (x print 0:1)
Syntax Error: Function call syntax is valid.
Grammar is correct.
RESULT: 5:6

Açıklama:

- x değişkeninin değeri 5:6 ile eşit olup olmadığı kontrol edilir.
- Eşit olduğu için (print x) ifadesi değerlendirilir ve 5:6 çıktısı üretilir.

While Döngüsü

Giriş:

scss
Copy code
(defvar i 1:1)
(while (less i 5:1) (set i (+ i 1:1)))

Beklenen Çıkış:

vbnet
Copy code
>>> (defvar i 1:1)
Tokens: (KW_DEFVAR IDENTIFIER VALUEF OP_CP)
Values: (1:1)
Identifiers: (i)
Syntax Error: Function definition syntax is valid.
Grammar is correct.
RESULT: 0:1

>>> (while (less i 5:1) (set i (+ i 1:1)))
Tokens: (KW_WHILE OP_OP KW_LESS IDENTIFIER VALUEF OP_CP OP_OP KW_SET IDENTIFIER OP_OP OP_PLUS IDENTIFIER VALUEF OP_CP OP_CP OP_CP)
Values: (5:1 1:1)
Identifiers: (i less i + i 1:1)
Syntax Error: Function call syntax is valid.
Grammar is correct.
RESULT: 5:1

Açıklama:

- i değişkeni 1:1 olarak tanımlanır.
- while döngüsü, i < 5:1 koşulu sağlandığı sürece i'yi i + 1:1 ile günceller.
- Döngü sonunda i değeri 5:1 olur.

For Döngüsü

Giriş:

bash
Copy code
(defvar sum 0:1)
(defvar j 1:1)
(for (j 1:1 5:1) (set sum (+ sum j)))

Beklenen Çıkış:

vbnet
Copy code
>>> (defvar sum 0:1)
Tokens: (KW_DEFVAR IDENTIFIER VALUEF OP_CP)
Values: (0:1)
Identifiers: (sum)
Syntax Error: Function definition syntax is valid.
Grammar is correct.
RESULT: 0:1

```
>>> (defvar j 1:1)
Tokens: (KW_DEFVAR IDENTIFIER VALUEF OP_CP)
Values: (1:1)
Identifiers: (j)
Syntax Error: Function definition syntax is valid.
Grammar is correct.
RESULT: 1:1

>>> (for (j 1:1 5:1) (set sum ( + sum j )))
Tokens: (KW_FOR OP_OP IDENTIFIER VALUEF VALUEF OP_CP OP_OP KW_SET IDENTIFIER
OP_OP OP_PLUS IDENTIFIER IDENTIFIER OP_CP OP_CP)
Values: (1:1 5:1)
Identifiers: (j set sum + sum j)
Syntax Error: Function call syntax is valid.
Grammar is correct.
RESULT: 10:1
```

Açıklama:

- sum değişkeni 0:1 olarak, j değişkeni 1:1 olarak tanımlanır.
- for döngüsü, j'yi 1:1'den 5:1'e kadar artırır ve her adımda sum'u sum + j ile günceller.
- Döngü sonunda sum değeri 10:1 olur.

Sonuç ve Değerlendirme

Bu rapor, sağlanan Lisp kodunun GPP adlı özel bir dil için nasıl bir yorumlayıcı işlevi gördüğünü ve işleyiş mantığını detaylı bir şekilde açıklamıştır. Kod, tokenizasyon, sözdizimi analizi, AST oluşturma ve değerlendirme aşamalarını başarılı bir şekilde gerçekleştirerek kullanıcıdan aldığı girişleri işleyip anlamlı sonuçlar üretmektedir.

Güçlü Yönler

- **Modüler Yapı:** Tokenizasyon, parsing ve değerlendirme gibi aşamalar ayrı fonksiyonlar halinde organize edilmiştir, bu da kodun okunabilirliğini ve bakımını kolaylaştırır.
- **Hata Yönetimi:** Sözdizimi hatalarını özel hata koşulları ile yönetir ve kullanıcıya anlamlı hata mesajları sunar.

- **Fonksiyonellik:** Değişken tanımlama, fonksiyon oluşturma ve çağırma, kontrol yapıları gibi temel programlama özelliklerini destekler.
- **Kesirli Değerler:** Kesirli değerler (N:D formatında) üzerinde aritmetik işlemler yapma yeteneği, belirli uygulama alanları için faydalı olabilir.

Geliştirilebilecek Alanlar

- **Genişletilmiş Veri Tipleri:** Şu anda sadece kesirli değerler ve boolean ifadeler destekleniyor. Diğer veri tiplerinin (örneğin, stringler, listeler) eklenmesi düşünülebilir.
- **Hata Mesajlarının Geliştirilmesi:** Daha detaylı ve yerel hata mesajları, kullanıcı deneyimini artırabilir.
- **Optimizasyon:** AST'nin daha etkin bir şekilde yönetilmesi ve değerlendirilmesi için optimizasyonlar yapılabilir.
- **Daha Karmaşık Fonksiyonellik:** Rekürsif fonksiyonlar, yerel değişkenler gibi daha karmaşık programlama özelliklerinin eklenmesi.

Sonuç

Sağlanan Lisp kodu, GPP adlı özel bir dil için temel bir yorumlayıcı olarak işlev görmektedir. Modüler yapısı ve temel programlama özelliklerini desteklemesi, bu yorumlayıcının eğitim amaçlı veya küçük ölçekli projeler için uygun olmasını sağlar. Ancak, daha geniş kapsamlı ve performans odaklı uygulamalar için ek geliştirmeler ve optimizasyonlar gerekmektedir.