Düzce Üniversitesi Bilim ve Teknoloji Dergisi

Araştırma Makalesi

Sayısal Çözümleme Yöntemlerinin Programlanması ve Yorumlanması

Hüseyin PEHLİVAN [a,*]

[a] *Bilgisayar Mühendisliği Bölümü, Mühendislik Fakültesi, Karadeniz Teknik Üniversitesi, Trabzon, TÜRKİYE*
** Corresponding author's e-mail address: pehlivan@ktu.edu.tr*

ÖZET

Sayısal hesaplama ya da diğer adıyla nümerik analiz, uygulamalı matematiğin önemli bir dalıdır. Matematiğin analitik çözüm üretemediği veya üretilen çözümün uygulama açısından yüksek hesaplama karmaşıklığına sahip olduğu durumlarda sayısal yöntemler kullanılır. Bu makalede, sayısal yöntemlerin programlanabilmesini sağlayan bir programlama dilinin tasarımı yapılmış ve bu dilde yazılan kaynak kodun değerlendirmesini yapabilen bir yorumlayıcı geliştirilmiştir. Dilin sözdizimi BNF (Backus Naur Form) notasyonunda tanımlanan bir LL(k) grameri ile temsil edilmiştir. Yorumlayıcı, her biri kodun farklı türden yorumlamasını yapan ayrıştırıcı, anlamsal denetleyici, simgesel türev alıcı ve kod değerlendirici gibi birkaç temel bileşenden oluşmaktadır. Dilin sözdizim analizi için kullanılan LL(k) ayrıştırıcı bileşeni, otomatik kod üretim aracı olan JavaCC yardımıyla üretilmiştir. Diğer bileşenler bu ayrıştırıcının oluşturduğu soyut sözdizim ağacı üzerinde çalışmaktadır. Dilin kullanımına yönelik olarak, birkaç sayısal kök bulma yönteminin programlaması ve yorumlaması gösterilmiştir. Bazı popüler diller ile dil metriklerine dayalı bir karşılaştırma yapılmış ve koşum zamanları değerlendirilmiştir.

Anahtar Kelimeler: *Sayısal yöntemler, programlama dilleri, biçimsel gramerler, ayrıştırıcılar, yorumlayıcılar*

Programming and Interpretation of Numerical Analysis Methods

ABSTRACT

Numerical calculation, or, in other words, numerical analysis, is an important branch of applied mathematics. Numerical methods are used where mathematics can not produce an analytical solution or the generated solution has high computational complexity. In this article, a programming language that allows numerical methods to be programmed is designed and an interpreter, which can evaluate the source code written in this language, is developed. The language syntax is represented by an LL(k) grammar defined in the BNF (Backus Naur Form) notation. The interpreter consists of several basic components such as parser, semantic controller, symbolic derivator and code evaluator, each of which makes a different kind of code interpretation. The LL(k) parser component used for the syntactic analysis of the language is generated via JavaCC, an automatic code generation tool. The other components work on the abstract syntactic tree that this parser generates. For the use of the language, the programming and interpretation of several numerical root finding methods are demonstrated. A comparison is made with some populer languages based on language metrics and the running time is evaluated.

Keywords: *Numerical methods, programming languages, formal grammars, parsers, interpreters*

Geliş: 07/01/2019, Düzeltme: 18/01/2019, Kabul: 25/01/2019

I. GİRİŞ

Mühendislik problemlerinin bilgisayar ortamları aracılığıyla çözülebilmesinde matematiksel modellerin önemli bir rolü vardır. Bu modellerin gerektirdiği karmaşık hesaplamaların bilgisayar programlarıyla yürütülmesine yönelik birçok sayısal ve simgesel yöntem geliştirilmiştir. Sayısal yöntemler, bir matematiksel problemin çözümünü belirli bir hata payı ile gerçekleştirir. Simgesel yöntemler ise, matematiksel problemlerin tam (hata içermeyen) çözümünü bulmaya çalışır. Simgesel hesaplama sistemlerinde, öncelikle problemlerin matematiksel denklemlerle ifade edilmesine ve programlanabilen algoritmalara dönüştürülmesine ihtiyaç duyulur [1]. Daha sonra denklemler kümesini çözmek üzere tasarlanan algoritmalar, özel bir programlama dili kullanılarak kodlanır.

Sayısal çözümleme yöntemlerini içerisinde barındıran Matlab [2], Maple [3], Mathematica [4] ve Macsyma [5] gibi birçok genel amaçlı hesaplama araçları geliştirilmiştir. Bu yazılım araçlarının yanında, belirli bir alana ve probleme özel olarak tasarlanmış, AWL, ΣC, MDL ve PObC++ olarak isimlendirilen çeşitli programlama dilleri vardır. AWL dili adaptif web sistemlerinin içerdiği protokol ve bileşenlerin yüksek seviyeli yapılarla tanımlanabilmesini sağlar [6]. ΣC dili yüksek performanslı gömülü işlemciler için bir programlama modeli sunar [7]. MDL dili ilaç üretim süreçlerinde kullanılan yazılım araçları için bir işbirliği ortamı hazırlar [8]. PObC++ dili dağıtık bellekli paralel programlama gereksinimlerini karşılar [9].

Programlama dili geliştirme süreçleri, sözdizimi tasarımından yorumlama veya makine dilinde kod üretimine kadar çok sayıda bileşenin kodlanmasını gerektirir. Bu bileşenler arasında, sözdizim analizi yapan dil ayrıştırıcılarını otomatik olarak üreten YACC [10], SableCC [11] ve JavaCC [12] gibi araçlar geliştirilmiştir. Herbir aracın, genellikle formal dil gramerlerinin yapısına benzerlik gösteren, kendine özel sözdizim bileşeni tanımlama biçimi bulunur [13]. Örneğin, Java dilinde ayrıştırıcı kaynak kodu üreten JavaCC aracı, bir programlama dili sözdiziminin içerdiği kelime ve ifadeler için, dil gramerinden uyarlanan farklı tanımlama biçimlerinin kullanılmasına ihtiyaç duyar.

Programlama dilleri, kullanım alanları açısından genel ve özel amaçlı olmak üzere iki gruba ayrılabilir. Genel amaçlı diller (C [15] ve Java [16] gibi), farklı türden problemleri çözebilen bilgisayar uygulamalarının yüksek seviyeli programlama yapıları ile yazılabilmesi için geliştirilmiştir. Sayısal yöntemlerin içerdiği simgesel türev ve fonksiyon dönüşümü [14] gibi matematiksel hesaplamaların programlanmasında kullanılamazlar. Bununla birlikte, hem sayısal hem de simgesel hesaplamalar için tasarlanan Matlab ve Mathematica gibi araçların tedarik ettiği özel amaçlı diller ile sayısal yöntemlerin programlanabilmesi mümkündür. Ancak, bu araçların ticari amaçlarla geliştirilmiş olması, genel amaçlı programlama çevrelerine kolay entegre edilememeleri ve farklı programlama yapılarına sahip olmaları kullanımlarını kısıtlamaktadır. Buna ilaveten, özel amaçlı diller tarafından desteklenen sözdiziminin ve fonksiyon davranışının matematiksel notasyonlardan farklı olması kod geliştirme ve doğrulama süreçlerini zorlaştırır. İki temel sebep olarak, değişken bildirimlerinin kapsamının fonksiyon gövdeleri ile sınırlı olmaması ve fonksiyonların parametrik davranış göstermemesi (başka bir ifadeyle, fonksiyon davranışlarının sadece kendilerine geçilen parametrelere dayandırılmaması) verilebilir. Diğer yandan, fonksiyon kütüphaneleri için ayrıntılı dokümanlar sunulmasına rağmen, dilin sözdizimi yeterince açıklanmamıştır. Genel amaçlı dillerle karşılaştırıldığında, özel amaçlı dillerle geliştirilen kaynak veriler farklı manalar taşıyabilen sözdizim bileşenleri içerebilirler.

Bu makalede, sayısal çözümleme yöntemlerinin kodlama gereksinimlerini karşılamak üzere, SaMP olarak isimlendirilen bir programlama dili geliştirilmiştir. Sözdizimi bir biçimsel gramer ile temsil

edilen SaMP dilinin ana hesaplama yapısını matematiksel ifadeler oluşturmaktadır. Dil için gerçeklenen yorumlayıcının kelime üreteci, ayrıştırıcı, tür denetleyicisi, türev alıcısı ve kod değerlendiricisi gibi birkaç bileşeni vardır. Ayrıştırıcı ile yapılan kelime analizi, sözdizim analizi ve sözdizim ağacının üretimine yönelik işlemlerde JavaCC aracı kullanılmıştır. Tür denetleme, simgesel türev ve kod değerlendirme işlemleri sözdizim ağacı üzerinden gerçekleştirilmiştir. Dilin sözdizimini göstermek için bazı sayısal yöntemler programlanmış ve yorumlanmıştır. Ayrıca, C ve Java gibi modern programlama dilleri ile dil metriklerine ve kod icra zamanlarına dayalı karşılaştırmalar yapılarak sonuçları verilmiştir.

II. SaMP DİLİNİN GENEL YAPISI

SaMP dilinin sözdizimi, sayısal yöntemlerin matematiksel temellerine uygun programlama yapabilmek için, matematiksel ifade gösterim biçimlerine (*notation*) dayandırılmıştır. Bu durum kaynak kod geliştirme ve doğrulama süreçlerinin yönetimini kolaylaştırır. SaMP dilindeki tanımlamalarda hem yordamsal dillerin hem de fonksiyonel dillerin programlama yapılarının kullanımına yer verilmiştir. Özellikle fonksiyonların içinde yapılan ara tanımlamalarda C [15] dilinin sözdizimi, gövde tanımlamalarında ise Haskell [17] dilinin sözdizimi desteklenmiştir.

Fonksiyon tanımlamalarında, parametre biçimlerine dayalı karşılaştırma yapan örüntü eşleme (*pattern matching*) tekniği kullanılmıştır. Bir örüntü, özel ifadelerden, değişkenlerden ya da sayı ve dizge gibi sabit verilerden oluşabilir. Bir fonksiyon tanımlamasında yer alabilecek örüntülerin kümesi, genellikle, fonksiyon parametrelerinin alabileceği bazı özel ya da temel değerlere göre belirlenir. Bir parametre için kullanılacak farklı örüntülere bağlı olarak farklı fonksiyon davranışları ortaya çıkar.
Bir fonksiyonun genel yapısı, aşağıda verildiği gibi, fonksiyon ismi, parametre listesi, ara tanımlamalar ve bir gövdeden oluşur.

```
fonksiyonİsmi(p1, p2, ..., pn) { t1; t2; ...; tn } = fonksiyonGövdesi
```

Bu yapı içerisinde, p1 ve p2 ile temsil edilen parametre alanlarında, parametrenin türü ile uyumlu sabit veri örüntüleri kullanılabilir. Sayısal veri türüne özel olarak $x+k$ (k bir tamsayı) gibi bir örüntüye de yer verilebilir. Diğer yandan, bir blok açılarak belirtilen t1 ve t2 gibi ara tanımlamalar yerine atama deyimleri ve standart çıkış üzerinde veri gösterimi yapan *print* deyimleri yazılabilir. Bu blok, istenirse fonksiyon tanımlamasından çıkarılabilir. Aşağıda sum ve pow isminde iki fonksiyonun tanımlaması gösterilmiştir.

```
sum (x,y+1) = x+y
pow (a, b) { c=a+b } = 2^c
```

Sayısal yöntemlerin programlanmasında bazı önemli matematiksel fonksiyonların tedarik edilmesine ihtiyaç vardır. SaMP dili tarafından kullanımı desteklenen sunulan bazı fonksiyon sözdizimleri Tablo 1'de gösterilmiştir.

Tablo 1. *SaMP dilinin desteklediği fonksiyonlar*

Fonksiyon Türü	**Fonksiyon Sözdizimleri**
Üstel	exp(x), sqrt(x)
Logaritmik	ln(x), log(x)
Trigonometrik	sin(x), cos(x), tan(x)
Standart çıkış	print(x), print(x, y, z)
Türev	drv(f(x), x), drv(f(x), n, x)
Diğer	abs(x), round(x), round(x, n)

Bir fonksiyonun tanımlaması birden fazla denklem ile birkaç kod satırına genişletilebilir. Denklemlerin bir kısmı, fonksiyon parametrelerinin bazı özel değerlerine yönelik örüntüler kullanılarak düzenlenebilir. Fonksiyonlar için bir genel denklem, bir ya da daha fazla sayıda özel denklem tanımlanabilir. Bu denklemlerin kaynak veri içerisindeki sırası önemli değildir. Fonksiyon değerlendirme sürecinde, parametrelerin değer-örüntü eşlemesine göre yönlendirme yapıldığı için, bir fonksiyonun bütün denklemleri yukarıdan aşağıya doğru kontrol edilir. Örüntü eşlemesi yapılabilen bir denklem tanımlanmışsa, ilgili denklemin gövdesi değerlendirilerek geri döndürülür. Eşleme olmaması durumunda, genel denklem değerlendirilmek üzere seçilir.

Aşağıda verilen fib fonksiyonunda, denklemlerin ikisi, x parametresinin özel değerleri olan 0 ve 1 örüntülerini içerdiği için fib(0) ve fib(1) fonksiyon çağrımlarının eşlenmesinde kullanılır. Diğer denklemin seçimi ise sadece n>1 durumunda mümkün olacaktır.

```
fib (0) = 0
fib (1) = 1
fib (n) = fib (n-1) + fib(n-2)
```

SaMP dilinde, emirsel (*imperative*) programlama dillerinin temel yapılarından biri olan döngü ifadelerine yer verilmemiştir. Bir döngü ifadesi ile yapılması gereken hesaplamalar için özyinelemeli (*rekursive*) fonksiyonlar kullanılmalıdır.

Bir fonksiyon denkleminin tanımlaması, her biri "|" koşul işleci ile başlatılan "koşul=gövde" biçiminde parçalı tanımlamalardan oluşabilir. Böyle bir tanımlama örneği yukarıda gösterilen fib fonksiyonu için aşağıdaki gibi yapılabilir.

```
fib (n)
   | n == 0 = 0
   | n == 1 = 1
   | n > 1  = fib (n-1) + fib(n-2)
```

Bu sözdiziminde, tanımlama satırlarının içerdiği koşul ifadeleri belirtildikleri sırada kontrol edilir ve ilgili koşulu doğrulayan ilk tanımlamanın gövdesi değerlendirilerek geri döndürülür. Burada diğer durumları temsil eden n>1 koşulu yerine "otherwise" anahtar kelimesi de yazılabilir.

SaMP dili tarafından desteklenen üç temel veri türü vardır. tamsayılar (*int*), reel sayılar (*double*) ve dizgeler (*string*). Sayısal veriler üzerinde tanımlanan temel aritmetik işlemlerden sadece toplama işlemi dizgeler üzerinde kullanılabilir. Bir değişkene veri ataması yapılmadan önce türünün

bildirilmesine ihtiyaç duyulmaz. Değişkenin türü kendisine ilk atama yapan ifade yardımıyla belirlenir. Değişkene atanan tüm değerler, sadece belirli bir veri türünün elemanları olabilir. Farklı türden değerler üzerinde yapılacak hesaplamalarda farklı değişkenler tanımlanmalıdır.

SaMP kaynak verisindeki fonksiyon, parametre ve değişken bildirimlerinin kapsamını belirlemek için düz (*flat*) blok yapısı kullanılır. Düz blok yapısında, program kodu birbiriyle ortak alanları bulunmayan bloklara ayrılarak, genel (*global*) ve yerel (*local*) olmak üzere iki bildirim düzeyi oluşturulur. Bunun sonucu olarak, bir fonksiyon tanımlamasına ait her bir denklem ile bir blok meydana getirilecektir. Blokların her biri parametre ve değişkenler için yapılan yerel düzeyli bildirimleri, tüm program kodu ise fonksiyonlar için yapılan genel düzeyli bildirimleri içerir.

II. SÖZDİZİM ANALİZİ

Bir programlama dilinin sözdizim (*syntax*) analizi bir biçimsel (*formal*) gramerin tasarlanmasını gerektirir. Bu bölümde, SaMP sözdizimine dayalı bir gramerin tasarımı yapılarak, JavaCC ile ayrıştırıcı üretiminde nasıl kullanılacağı gösterilmiştir.

A. GRAMER TASARIMI

Biçimsel dil gramerlerini tanımlamak amacıyla geliştirilen birkaç matematiksel gösterim biçimi vardır. BNF en fazla kullanılan gösterim biçimlerden biridir. EBNF (Extended Backus Naur Form) denilen diğer bir gösterim biçimi, BNF'nin meta karakterler (*, + , ?, | vb.) eklenerek genişletilmesiyle oluşturulmuştur. İki gösterim biçiminde de, gramerlerin belirtimi için, her biri üretim kuralı (*production rule*) olarak adlandırılan birbirine bağlanmış tanımlamalar dizisi kullanılır. Bir gramer kuralı "=", "::=" veya "->" gibi sembollerle ayrılan sol ve sağ yanlı tanımlamalarına sahiptir. Kuralın sol yanı bir sonlu olmayan sözcük (*non-terminal*), sağ yanı ise bir veya birkaç sonlu sözcük (*terminal*) ya da sonlu olmayan sözcük içerir. Aşağıdaki gramer bildirimlerinde sonlu sözcükler çift tırnak içerisinde, sonlu olmayan sözcükler ise "<" ile ">" simgeleri arasında yazılmıştır.

Sözdizim analizlerinde öncelikle kaynak veri soldan sağa doğru taranarak sonlu sözcük dizilerine parçalanır. Daha sonra bu dizideki sözcük sırası bir gramer yardımıyla incelenir. Sözcük incelemesi yukarıdan aşağıya (LL(k) ayrıştırması) ya da aşağıdan yukarıya (LR(k) ayrıştırması) doğru iki farklı yönde yapılabilmektedir; burada k, kural seçiminin en az kaç sözcük ile yapılabileceğini gösterir. JavaCC ile sadece LL(k) ayrıştırıcıları üretilebildiği için, SaMP dilini temsil edecek biçimsel gramerdeki kural kümesinin aşağıda verilen üç kriteri karşılaması gerekir.

- Bir kuralın bütün alternatifleri (sol tarafında aynı sonlu olmayan sözcük bulunan kurallar) tarafından üretilen ilk k sonlu sözcüğün dizilimleri farklı olmalıdır.
- Sonlu sözcük üretmeyebilen bir kural ile kendinden sonra çağrımı yapılan diğer bir kural tarafından üretilen ilk k sonlu sözcüğün dizilimleri farklı olmalıdır.
- Bir kuralın sağ tarafı soldan özyineli bir tanımlama içermemelidir.

Bu kriterlere uygun olarak k=1 için geliştirilen LL(1) grameri, EBNF gösterim biçimi kullanılarak Şekil 1'de verilmiştir. Gramerde <id>, <num>, <dnum> ve <str> sonlu olmayan sözcükleri için

gerekli kural tanımlamaları yapılmamıştır. Bu kurallar ile üretilecek verilerin biçimi, ayrıştırıcının birim sözcük (*token*) bildirim bloğu içerisinde tanımlanacaktır.

```
<program> ::= <func> (<program>)?
<func> ::= <header> (<block >)? <body>
<header> ::= <id> "(" (<parlist>)? ")"
<block> ::= "{" (<stmlist>)? "}"
<body> :.= ("=" <expr> | <eqlist>)
<parlist> ::= <param> ("," <parlist>)?
<param> ::= <id> ("+" <num>)? | <num>
<eqlist> ::= "|" <bexpr> "=" <expr> (<eqlist>)?
<bexpr> ::= <and> ("||" <and>)*
<and> ::= <not> ("&&" <not>)*
<not> ::= "!" "(" <bexpr> ")"
<not> ::= <belem> | "otherwise"
<belem> ::= <expr> <boper> <expr>
<boper> ::= "==" | "/=" | "<" | "<=" | ">=" | ">"
<stmlist> ::= <stm> ";" (<stmlist>)?
<stm> ::= <id> "=" <expr>
<stm> ::= "print" "(" <explist> ")"
<explist> ::= <expr> ("," <explist>)?
<expr> ::= (<sign>)? <term> (<sign> <term>)*
<sign> ::= "+" | "-"
<term> ::= <power> (("*" | "/" | "%") <power>)*
<power> ::= <elem> ( "^" <power> )?
<elem> ::= <id> ("(" (<explist>)? ")")?
<elem> ::= <num> | <dnum> | <str>
<elem> ::= "(" <expr> ")"
<elem> ::= "ln" "(" <expr> ")"
<elem> ::= "log" "(" <expr> ")"
<elem> ::= "exp" "(" <expr> ")"
<elem> ::= "sin" "(" <expr> ")"
<elem> ::= "cos" "(" <expr> ")"
<elem> ::= "tan" "(" <expr> ")"
<elem> ::= "abs" "(" <expr> ")"
<elem> ::= "sqrt" "(" <expr> ")"
<elem> ::= "round" "(" <expr> ")"
<elem> ::=
      "drv" "(" <fcall> "," (<num> ",")? <id> ")"
<fcall> ::= <id> "(" (<explist>)? ")"
```

Şekil 1. *SaMP dili biçimsel grameri*

Şekil 1'de gösterilen gramer, SaMP dilinin anlamsal yapısına ilişkin bazı tanımlamaları da içerir. Gramerin ilgili kuralları, anlamsal yapının iki önemli konusu olan işleç öncelikleri ile birleşme yönlerini kapsayacak biçimde tanımlanmıştır. Örneğin, <expr> kuralı, diğer aritmetik işleçleri içeren kurallara (<term> ve <power>) göre daha yukarı bir konumda bulunmaktadır. Bu konum, sözdizim analizi yapılırken, <expr> kuralının daha önce çağrılmasını ve ayrıştırma ağacının köküne yakın bir düğümüne yerleştirilmesini sağlar. Dolayısıyla, değerlendirme sürecinin ağacın yapraklarından kök düğümüne doğru ilerlediği göz önüne alındığında "+" ve "-" işleçleri daha düşük öncelik düzeyine sahip olacaktır. Diğer yandan, işleçlerin birleşme yönleri, gramer kuralları soldan ya da sağdan özyineli yapıda tanımlanarak temsil edilmiştir. Örneğin, <power> kuralı sağdan özyineli yapılarak, kuralın içerdiği "^" işlecine sağdan sola birleşme yönü kazandırılmıştır.

B. SÖZCÜK ÜRETECİ

Bir programlama dilinin sözcük yapısı genellikle düzenli ifadeler (*regular expression*s) yardımıyla tanımlanır. SaMP dilinin sözcük kümesini işlevsel ve biçimsel sözcükler olarak iki kısma ayırabiliriz. İşlevsel sözcükler, dilin kendisi tarafından belirtilen işleç, anahtar kelime ve diğer simgelerden oluşur. Biçimsel sözcükler ise Şekil 1'de gösterilen <id>, <num>, <dnum> ve <str> sonlu olmayan sözcük kuralları tarafından üretilir. Her bir işlevsel sözcüğün ayrı bir birim sözcük (*token*) sınıfı tanımlanırken, biçimsel sözcükler arasından aynı biçime sahip olan (yani, aynı kural tarafından üretilen) sözcükler için sadece bir birim sözcük sınıfına ihtiyaç vardır. Sözdizim analizinde, verinin değerinden ziyade türü önemli olduğundan, birim sözcük sınıflarının sözcüğün biçimine göre tanımlanması yeterlidir.

Şekil 2'de, JavaCC tanımlama formatına uygun olarak, SaMP dilindeki birim sözcük sınıflarına ait tanımlamalar TOKEN bloğu içerisinde verilmiştir. SKIP bloğu içinde yapılan tanımlamaların kapsadığı sözcükler için herhangi bir birim sözcük üretimi yapılmaz. LETTER ve DIGIT gibi # ile başlayan sınıf isimleri diğer tanımlamaların parçası olan alt tanımlamaları gösterir.

```
TOKEN: {
    <PLUS: "+"> | <MINUS: "-"> | <TIMES: "*"> | <DIVIDE: "/"> | <MOD: "%">
  | <POWER: "^"> | <AND: "&&"> | <OR: "||"> | <NOT: "!"> | <AEQ: "=">
  | <EQ: "=="> | <NE: "/="> | <LE: "<"> | <LT: "<="> | <GT: ">=">
  | <GE: ">"> | <COMMA: ","> | <SEMI: ";"> | <GUARD: "|"
  | <LCURLY: "{"> | <RCURLY: "}"> | <LPAREN: "("> | <RPAREN: ")">
  | <LN: "ln"> | <LOG: "log"> | <EXP: "exp"> | <SIN: "sin"> | <COS: "cos">
  | <TAN: "tan"> | <ABS: "abs"> | <SQRT: "sqrt"> | <ROUND: "round"> | <DRV: "drv">
  | <PRINT: "print"> | <OTHER: "otherwise">
  | <#LETTER: ["a"-"z","A"-"Z"]> | <#DIGIT: [0"-"9"]>
  | <ID: <LETTER>(<LETTER> | <DIGIT>)*>
  | <NUM: (<DIGIT>)+> | <DNUM: (<DIGIT>)+"."(<DIGIT>)+>
  | <STR: ("\"" (~["\""] | "\\" "\"")* "\"")>
}
SKIP: { " " | "\t" | "\r" | "\n" }
```

Şekil 2. *Birim sözcük tanımlamaları*

Sözcük üreteci, kaynak veriyi Şekil 2'de verilen birim sözcük sınıflarına göre parçalar ve bir birim sözcük dizisi oluşturur. Örneğin, print(x*2, y) ifadesi için üretilecek birim sözcük dizisi aşağıdaki gibi olacaktır.

PRINT LPAREN ID TIMES NUM COMMA ID RPAREN

Bu sözcük dizisi, Şekil 2'deki tanımlama sırasının önemli olduğunu gösterir. Sözcük üreteci, kaynak veride taradığı her bir sözcük için, yukarıdan aşağıya doğru bütün birim sözcük tanımlamalarını değerlendirir ve sözcüğü ilk kapsayan tanımlamayı seçerek ilgili birim sözcüğü üretir. Örneğin, "print" sözcüğünü kapsayan iki mümkün sözcük sınıfı vardır; PRINT ve ID. Ancak, SaMP dilinde "print" bir anahtar sözcük olduğu için, PRINT birim sözcüğünün üretimi yapılmalıdır. Bu nedenle, PRINT tanımlaması, ID'den önce gelen bir konuma yerleştirilerek doğru birim sözcüğün üretimi sağlanmıştır.

C. AYRIŞTIRICI

Ayrıştırıcılar, biçimsel bir dilde yazılan kaynak verinin ifade yapısının analizinde kullanılır. El yordamıyla veya JavaCC benzeri otomatik kod üreten bir araç yardımıyla geliştirilebilmeleri mümkündür. Kod üretim araçları, genellikle dilin gramer kurallarına dayandırılan metot tanımlamaları içerisinden ayrıştırıcılar için gerekli kaynak kodu üretir. Kodun dili, kullanılan araca göre değişiklik gösterir. Örneğin, JavaCC aracı ile ayrıştırıcı kodunun üretimi Java programlama dilinde gerçekleşir.

Dilin sözdizimini temsil eden gramere bağlı olarak farklı yönlerde gerçekleştirilen iki temel ayrıştırma çeşidi vardır; LL(k) ve LR(k). Yukarıdan aşağıya doğru sözdizim analizi yapan bir LL(k) ayrıştırıcısı, gramer kurallarının yapısına uygun şekilde yapılandırılabilir. Şekil 1'de verilen ve sadece bir birim sözcük (k=1) ile kural seçimi yapılabilen biçimsel gramer için bir LL(1) ayrıştırıcısı geliştirmek

mümkündür. Bu yol içerisinde, her bir gramer kuralını temsilen metot tanımlamaları ayrıştırıcıya eklenir ve kuralların birbirlerini çağırdığı sıraya göre çağrılmaları sağlanır. Şekil 3'te, JavaCC tanımlama formatı kullanılarak, bir LL(1) ayrıştırıcısına ait metot tanımlamalarından bazıları gösterilmiştir.

```
void start() : { } { program() <EOF> }
void program() : { } { function() ( program() )? }
void function() : { } { header() ( block() )? body() }
void header() : { } { <ID> <LPAREN> ( parlist )? <RPAREN> }
void block() : { } { <LCURLY> ( stmlist() )? <RCURLY> }
void body() : { } { (<AEQ> expr() | eqlist() ) }
void parlist() :{ } { param() (<COMMA> parlist() )? }
void param() : { } { <ID> (<PLUS> <NUM> )? | <NUM> }
void eqlist() : { } { <GUARD> bexpr() <AEQ> expr() ( eqlist() )? }
void bexpr() : {} { and() ( <OR> and() )* }
void and() : {} { not() ( <AND> not() )* }
void not() : { } { <NOT> <LPAREN> bexpr <RPAREN> } | <OTHER> | belem()
void belem() : { } { … }
void stmlist() : { } { stm()<COMMA> (stmlist() )? }
void stm() : { } { <ID> <AEQ> expr() | <PRINT> <LPAREN> explist() <RPAREN> }
void explist() : { } { expr() (<COMMA> explist() )? }
void expr() : { } { ( <PLUS> | <MINUS> )? term() ( ( <PLUS> | <MINUS> ) term() )* }
vod term() : { } { poser() ( (<TIMES> | <DIVIDE> | <MOD> ) power() ) * }
void power() : { } { elem() ( <POWER> power() )? }
elem() : { } { … }
```

Şekil 3. *Ayrıştırıcı metotları*

Şekil 3'de görüldüğü gibi, ayrıştırıcının başlangıç metodu start() olarak isimlendirilmiştir. Bir JavaCC birim sözcük bileşeni olan <EOF>, standart girişten girilebilen ya da bir dosyadan okunabilen kaynak verinin sonunu işaretlemek için kullanılmıştır. Ayrıştırma işlemi, kaynak veride bu birim sözcük ile karşılaşıldığında sonlanır.

D. SÖZDİZİM SINIFLARI

Kaynak veriyi nesneye dayalı programlama yapıları yardımıyla temsil etmek için tanımlanan sınıflar, sözdizim sınıfları olarak adlandırılır. Dilin gramer kuralları, tanımlanması gereken sözdizim sınıflarının belirlenmesinde önemli bir role sahiptir. Genel olarak, bir işleç ya da anahtar kelimeyi içeren her bir gramer kuralına karşılık bir sözdizim sınıfı tanımlanır. Alternatifi bulunan kuralları temsil eden sınıfların tanımlaması, aynı üst sınıftan miras alınarak yapılır ve kurallara benzer şekilde birbirlerine alternatif olabilmeleri sağlanır.

Bazı özel durumlar gramer kurallarının birleştirilerek tek bir sözdizim sınıfı ile temsil edilmesini gerektirir. Özellikle belirli bir kuralın tamamlayıcısı olan veya o kural tarafından çağrımı yapılan kuralların sadece bir sözdizim sınıfı ile tanımlanması yeterlidir. Örneğin, bir programlama dili ifadesinin birden fazla gramer kuralı ile tanımlanması durumunda, bu kuralların tamamını temsil edecek biçimde bir sözdizim sınıfı tanımlanır. Şekil 1'deki gramerde, <function> kuralının bileşenleri olan <block> ve <body> kuralları için sözdizim sınıfları tanımlanmamıştır.

Bir sözdizim sınıfına, temsil edilen gramer kuralının ismi veya bu kural tarafından üretilen ifade ile bağlantılı başka bir isim verilebilir. Sınıfın alan verileri, ilgili kuralın içerdiği sonlu ve sonlu olmayan sözcük bileşenlerine uygun veri türleri ile tanımlanır. Kural tanımlamaları yapılmayan< id>, <num>, <dnum> ve <str> sonlu olmayan sözcük bileşenleri için sırasıyla *Var*, *Num*, *DNum* ve *Str* sınıfları oluşturulmuştur. Şekil 4'te, sözdizim sınıflarından bazıları gösterilmiştir.

```
class Program {
  Function def; Program prog;
  public Program(Function x, Program y)
   { def = x;  prog = y; }
}
class Function {
  Header f;  Stm s;  QList eq;
  public Function(Header x, Stm y, QList z)
    { f = x;  s = y;  eq = z; }
}
class EList {
  Exp e;  EList eq;
  public EList(Exp x, EList y)
    { e = z;  eq = y; }
}
class QList {
  BExp b;  Exp e;  QList eq;
  public QList(BExp x, Exp y, QList z)
    { b = x;  e = y;  eq = z; }
}
```

```
class Stm { }
class LStm extends Stm {
 Stm a, b;
  public LStm(Stm x, Stm y) { a = x; b = y; }
}
class AStm extends Stm {
  String id;  Exp e;
  public AStm(String x, Exp y)
    { id = x;  e = y; }
}
class Exp { }
class Header extends Exp {
  String id; EList eq;
  public Header(String x, EList y)
    { id = x;  eq = y; }
}
class Plus extends Exp {
 Exp a, b;
 public Plus(Exp x, Exp y) { a = x; b = y; }
}
```

Şekil 4. *Sözdizim sınıfları*

E. SOYUT SÖZDİZİM AĞACI

Bir sözdizim ağacı, kaynak verinin ağaç veri yapısı biçiminde bir temsili olarak üretilir. Sözdizim sınıflarından türetilen nesneler birbirine bağlanarak oluşturulduğu için, hiyerarşik bir yapısı vardır. Kaynak kodun işlem bileşenleri (işleçler, anahtar kelimeler, vb.) ağacın ara düğümlerini, veri bileşenleri (sabitler, değişkenler, vb.) ise yapraklarını meydana getirir. Bu yol içerisinde, ağacın her bir düğümü ilgili kod bileşeninin türüne bağlı olarak farklı bir sözdizim sınıfına ait nesne içerebilir.

Sözdizim ağaçları, kaynak veri üzerinden yapılması zor olan tür denetimi ve kod yorumlama gibi işlemleri gerçekleştirmek için kullanılır. Kaynak verinin sözdizim analizinde olduğu gibi, sözdizim ağaçlarının üretiminde de JavaCC aracından yararlanılabilir. Bunun için, ayrıştırıcı metotlarının gövdelerinde özel kod blokları açılarak, sözdizim ağacının ilgili düğüm verisini üretecek Java dili ifadeleri bu bloklar içerisine eklenir. Sözdizim analizi ile eşzamanlı olarak icra edilen bu ifadeler üzerinden ağacın üretimi sağlanır. Şekil 5'te bazı ayrıştırıcı metotları, sözdizim ağacına yönelik eklenen kod blokları ile birlikte gösterilmiştir.

```
Program start() : { Program prog; }
  { prog=program() <EOF> { return prog; } }
Program program() :{ Function def; Program prog = null; }
  { def=function() (prog=program() )? { return new Program(def, prog); } }
Function function() : { Header fn; Stm s=null; QList eq; Exp e; }
  { fn=header() ( s=block() )? <AEQ> eq=body() { return new Function(fn, s, eq); } }
Header header() : { Token t; EList eq = null; }
  { t=<ID> <LPAREN> (eq=parlist())? <RPAREN> { return new Header(t.image, eq); } }
Stm block() : { Stm s=null; }
  { <LCURLY> ( s=stmlist() )? <RCURLY> { return s; } }
QList body() : { Exp e; QList eq; }
  { (<AEQ> e=expr(){ eq = new QList(new BNum(true), e, null); } | eq=eqlist() ) { return eq; } }
EList parlist() :{ EList eq = null; Exp e; }
  { e=param() (<COMMA> eq=parlist() )? { return new EList(e, null, eq); } }
Exp param() : { Token t, t2; Exp e; }
  { t=<ID> (<PLUS> t2=<NUM>
              { return new Plus(new Var(t.image), new Num(Integer.parseInt(t2.image))); } )?
    { return new Var(t.image); } | t=<NUM> { return new Num(Integer.parseInt(t.image)); } }
QList eqlist() : { Exp e; BExp b; QList eq = null; }
  { <LCURLY> b=bexpr() <AEQ> e=expr() <RCURLY> ( eq=eqlist() )?
    { return new QList(b, e, eq); } }
```

Şekil 5. *Sözdizim ağacını üreten ifadelerin eklenmesi*

Sözdizim analizi start() metodunun çağrımı ile başlayacağı için, sözdizim ağacının kök düğümünde daima Program türünden bir nesne bulunur. Analiz esnasında çağrımı yapılacak diğer metotlar farklı türden nesneler üretir. Örneğin, Şekil 4'teki sınıf tanımlamaları ile Şekil 5'deki metot tanımlamalarını kullanarak, print(x*2, y) ifadesi için ayrıştırıcının üreteceği sözdizim ağacı aşağıdaki gibi olacaktır.

```
PStm p = new PStm(new EList(new Times(new Var("x"),
            new Num(2)), null, new EList(new Var("y"), null, null)));
```

Bağlı liste olarak tanımlanan EList ve QList sınıfları, birkaç ayrıştırıcı metodu tarafından nesne üretiminde kullanılır. Fonksiyon formal parametrelerinin listesi, print fonksiyonunun ve kullanıcı tanımlı fonksiyonların argüman listesi EList sınıfının bir nesnesi olarak oluşturulur. QList sınıfının nesneleri ise koşul ifadeleri ile birlikte parçalı fonksiyonların listelerini tutar.

III. DEĞERLENDİRME

Anlamsal denetleyici, türev alıcı ve kod değerlendirici olmak üzere kaynak kodun yorumlanması aşamasında üç bileşen kullanılır. AST verisini farklı biçimlerde yorumlayan bu bileşenlerin kodlanmasında yazılım mühendisliği tasarım desenleri [18] arasında bulunan Ziyaretçi (Visitor) deseninden yararlanılmıştır.

A. ZİYARETÇİ TASARIM DESENİ

Ziyaretçi tasarım deseni (*pattern*), hiyerarşik yapıya sahip ağaç verilerinin değerlendirilmesinde kullanılır. Soyut sözdizim ağacını oluşturan sözdizim sınıflarının yerel verileri üzerinde tanımlanacak işlemlerin farklı bir sınıfa taşınabilmesini sağlar. Bunun için, Accept ve Visitor gibi iki arayüz bileşeninin gerçeklenmesine ihtiyaç vardır. Accept arayüzünü sözdizim sınıfları gerçeklerken, Visitor arayüzünü sözdizim ağacını değerlendirecek sınıf gerçekler.

Accept arayüzünün gerçeklenmesine yönelik her bir sözdizim sınıfına accept() isminde bir metot eklenir. Şekil 6'da Stm, AStm, Exp ve Header sınıflarının, accept() metotları eklenmiş yeni versiyonları gösterilmiştir.

```
abstract class Stm { public abstract void accept(Visitor v); }
class AStm extends Stm {
  String id;  Exp e;
  public AStm(String x, Exp y) { id = x;  e = y; }
  public void accept(Visitor v) { v.visit(this); }
}
abstract class Exp { public abstract Object accept(Visitor v); }
class Header extends Exp {
  String id; EList eq;
  public Header(String x, EList y) { id = x;  eq = y; }
  public Object accept(Visitor v) { return v.visit(this); }
}
```

Şekil 6. *Accept arayüzünün gerçeklemesi*

Visitor arayüzü, sözdizim sınıfları üzerinde tanımlanması gereken visit() ismindeki metotların başlık bilgilerini içerir. Şekil 7'de Visitor arayüzünü oluşturan metotların bazıları listelenmiştir.

```
public interface Visitor  {
  public void visit(Stm s);
  public void visit(LStm s);
  public void visit(AStm s);
  public void visit(PStm s);
  public Object visit(Exp e);
  public Object visit(Header e);
  public Object visit(Plus e);
  public Object visit(Minus e);
  public Object visit(Times e);
  public Object visit(Divide e);
  public Object visit(Mod e);
  public Object visit(Power e);
  public Object visit(Var e);
  public Object visit(Num e);
  public Object visit(DNum e);
  public Object visit(Str e);
  public Object visit(Ln e);
  public Object visit(Log e);
  public Object visit(Ep e);
  public Object visit(Sin e);
  public Object visit(Cos e);
  public Object visit(Tan e);
  public Object visit(Abs e);
  public Object visit(Sqrt e);
  public Object visit(Round e);
  public Object visit(DrvExp e);
  public boolean visit(BNum e);
  ……..
}
```

Şekil 7. *Visitor arayüzü*

Şekil 7'deki metot bildirimlerinin bazıları için geri döndürülen veri türü Object olarak belirtilmiştir. Bu çeşit metot bildirimleri, soyut sözdizim ağacı üzerinde yapılacak farklı değerlendirme işlemlerinde (tür denetimi, simgesel türev, vb.) aynı Visitor arayüzünün kullanılabilmesini sağlamak için gereklidir.

B. ANLAMSAL DENETLEYİCİ

Anlamsal denetleyici bileşeni ile yapılan tür denetim işlemlerinde, kaynak kod içindeki tanımlamalardan yararlanılarak oluşturulan bir sembol tablosu kullanılır. Bu tabloya, fonksiyonların gövdesinde tanımlanan parametreler ile yerel değişkenler birer sembol olarak eklenir. Her bir sembol ekleme işleminde isim kontrolü yapılır. Birden fazla parametre veya değişkene aynı ismin verilmiş olması durumunda hata gösterilir. Şekil 8'de sembol tablosu olarak kullanılan SymTable sınıfı gösterilmiştir.

```
class SymTable {
  int size; int index = -1;
  Hashtable[] table;
  public SymTable(int s) {
    size = s;
    table = new Hashtable[s];
  }
  public int beginScope() {
    if (++index >= table.length)
      return -1;
    table[index] = new Hashtable();
    return 0;
  }
  public void endScope() {
    -- index;
  }
  public void put(String id, Object obj) {
    if (obj == null)
      return ;
    table[index].put(id, obj);
  }
  public Object get(String id) {
    return table[index].get(id);
  }
}
```

Şekil 8. *SymTable sınıfı*

Bir fonksiyonun analizi yapılacağı zaman sembol tablosunda yeni bir sembol bloğu oluşturularak, fonksiyona ait bütün semboller ve türleri bu blok içerisinde saklanır. Sembol bloğu sadece ilgili fonksiyonun analizi esnasında erişime açıktır ve analiz sona erdiğinde serbest bırakılır.

Kaynak kodun anlamsal analizinde ayrıştırıcının oluşturduğu nesne ağacından yararlanılır. Bu ağacın bütün düğümleri ziyaret edilerek kaynak veride tanımlaması yapılan her bir isim, sembol tablosunda (isim,tür) çifti biçiminde saklanmıştır. Örneğin, *int* türünden bir x değişkeni için ("x", new Num(0)) çifti, String türünden bir y değişkeni için ("y", new Str("")) çifti sembol tablosuna eklenir. Kullanımı yapılan isimler için sembol tablosundan sorgulama yapılarak türleri okunur.

Sembol tablosu üzerinden yapılan tür denetimlerinde kontrol edilmesi gereken çeşitli durumlar vardır. Bu durumlar kaynak verinin bileşenleri ile ilişkilendirilerek aşağıda özetlenmiştir.

- *Türler:* İşleçler, doğru türden değişkenlere veya sabit verilere uygulanmalıdır. Argümanlar ile karşılığı olan parametrelerin türleri uyumlu olmalıdır. Parametre ve değişkenlere, türleriyle uyumlu değerler atanmalıdır.
- *Denklemler:* Bir fonksiyonun bütün denklemleri aynı sayıda parametre kullanmalı ve aynı türden veriyi geri döndürmelidir.

- *Bildirimler:* Kullanımı yapılan değişen ya da fonksiyonlar tanımlanmış olmalıdır. Değişkenlere ilk kullanımlarından önce değer atanmalıdır.

Anlamsal denetleyici, Visitor arayüzünün bir gerçeklemesi olan TypeVisitor sınıfı ile temsil edilmiştir. Şekil 9'da, TypeVisitor sınıfını oluşturan bazı visit() metotlarının tanımlamaları verilmiştir.

```
class TypeVisitor implements Visitor {
  Program p;
  SymTable t;
  public DeriveVisitor(Program pg, SymTable tb) {
    p = pg;
    t = tb;
  }
  public void visit(PStm s) {
    EList eq = s.eq;
    while (eq != null) {
      Object a = eq.e.accept(this);
      if (a == null) {
        System.out.println("Type error: " + new PrintVisitor().visit(eq.e));
        return;
      }
      eq = eq.eq;
    }
  }
  public Object visit(Minus e) {
    Object a = e.a.accept(this);
    Object b = e.b.accept(this);
    if (a == null || b == null)
      return null;
    if (a instanceof Str || b instanceof Str) {
      System.out.println("Type error: " + new PrintVisitor().visit(e));
      return null;
    }
    else if (a instanceof DNum || b instanceof DNum)
      return new DNum(0);
    else
      return new Num(0);
  }
  …….
}
```

Şekil 9. *TypeVisitor sınıfı*

Şekil 9'da Minus sınıfı için verilen visit() metodu, çıkarma işlecini içeren ifadelerin tür denetimini yapmaktadır. İfadenin solundaki ve sağındaki değişken ya da verilerin tür uyumluluğu kontrol edilmektedir. Verilerden birinin tür çıkarımın string olması durumunda hata gösterimi yapılır. Bir ifadenin tür çıkarımı, özellikle işleçlerin öncelik düzeylerine ve birleşme yönlerine göre yapılmaktadır. Örneğin, aşağıdaki deyimde x değişkeni için tür çıkarımı string olarak yapılacaktır.

x = 5 - 3.4 + "2"

Burada, toplama işlemi solda sağa doğru yapıldığından, ilk işlem olan 5 - 3.4 ile hesaplanan 1.6 verisinin türü *double*, sonraki işlem olan 1.6 + "2" ile hesaplanan "1.62" verisinin türü *string* olacaktır. Tür çıkarımı yapılamaması ya da tür tutarsızlığı ile karşılaşılması durumunda, kaynak koddaki ilgili ifadeyi göstermek için, yine Visitor arayüzünden gerçeklemesi yapılan PrintVisitor sınıfı kullanılır.

C. TÜREV ALICI

SaMP dilinde bir fonksiyonun tanımlaması, zorunlu olmayan bir deyimler bloğunu takiben bir ana gövde ifadesi veya bu ifadenin yerine yazılabilen bir parçalı ifadeler listesi içerir. Türev işlemi sadece ifadeler üzerinde yapılabildiğinden, bir fonksiyonun türevi için, ana gövde ifadesi veya seçimi yapılabilen ilk parçalı ifade kullanılmıştır. Fonksiyonun geri dönüş değeri, fonksiyona geçilen parametrelerle önce deyimler bloğu ve ardından türev uygulama sonucu elde edilen ifade değerlendirilerek hesaplanır.

Fonksiyonların simgesel türevleri sözdizim ağacı üzerinden gerçekleştirilir. Kaynak veride tanımlı bütün fonksiyonların gövde ifadeleri ağacın belirli bir bölgesinde bulunur. Visitor arayüzünün bir gerçeklemesi olan DeriveVisitor sınıfı, ifade bileşenlerini içeren bu ağaç bölgesi üzerinde işlemlerini gerçekleştirir. Bu nedenle, visit() metotlarının sadece Exp sınıfının mirasçıları üzerinde tanımlanması yeterlidir. Şekil 10'da DeriveVisitor sınıfının bazı örnek metotları gösterilmiştir.

```
public class DeriveVisitor implements Visitor {
  …….
  public Object visit(Power e) {
    Exp a = (Exp)(e.a.accept(this));
    Exp b = (Exp)(e.b.accept(this));
    return new Plus(new Times(new Times(e, b), new Ln(e.a)),
              new Times(new Times(e.b, a), new Power(e.a, new Minus(e.b, new Num(1)))));
  }
  public Object visit(Log e) {
    Exp a = (Exp)(e.a.accept(this));
    return new Divide(a, new Times(e.a, new Ln(new Num(10))));
  }
  public Object visit(Sin e) {
    Exp a = (Exp)(e.a.accept(this));
    return new Times(a, new Cos(e.a));
  }
  public Object visit(Var e) {
    return (e.id.equals(v) ? new Num(1) : new Num(0));
  }
  …….
}
```

Şekil 10. *DeriveVisitor sınıfı*

Şekil 10'da görüldüğü gibi, türev tanımlamaları ifadelerin genel durumlarına yönelik verilmiştir. Örneğin, Power sınıfıyla temsil edilen x^n (n tamsayı) ifadesi yerine u^v ifadesinin türevi aşağıdaki ifadeden uyarlanarak tanımlanmıştır.

$f'(x)=u^v \cdot v'.\ln(u)+v.u'.u^{v-1}$

Bunun sonucu olarak, $f(x)=x^n$ için simgesel türev aşağıdaki gibi hesaplanacaktır.

$f'(x)=x^n \cdot n'.\ln(x)+n.x'.x^{n-1}= n.x^{n-1}$

SaMP kaynak verisi içerisinden türevin bildirilebilmesi için drv() fonksiyonu tanımlanmıştır. Fonksiyonun argümanlarını, sırasıyla, çağrım ifadesi, türevin mertebesi ve türev alınan değişken oluşturur; örneğin, fn(x) fonksiyonunun x=3 noktasındaki ikinci mertebeden türevi drv(fn(3), 2, x) sözdizimi ile ifade edilir. Türevin uygulanması, sözdizim ağacındaki DrvExp düğümünün değerlendirilmesiyle gerçekleşir.

D. DEĞERLENDİRİCİ

Kaynak kodu değerlendirme sürecinde de, Şekil 8'de sembol tablosunu temsilen verilen SymTable sınıfı kullanılmıştır. Sözdizim ağacının düğümleri ziyaret edilerek yapılan değerlendirmede, parametrelere ya da yerel değişkenlere atanan değerler (isim,değer) çiftleri biçiminde sembol tablosuna yerleştirilir. Örneğin, x=5.0 ifadesi için ("x", new DNum(5.0)) çifti tabloya eklenir. Bu ifadenin y=x*2 gibi diğer bir ifade ile takip edilmesi durumunda, x değişkeninin değeri sembol tablosundan sorgulanır ve diğer bir değer çifti ("y", new DNum(10.0)) olarak tabloya eklenir.

Kod değerlendirici bileşeni, Visitor arayüzünden gerçeklenen EvalVisitor sınıfı ile temsil edilmiştir. Şekil 12'de, bazı visit() metotlarının tanımlamaları verilerek EvalVisitor sınıfı gösterilmiştir.

```
class EvalVisitor implements Visitor {
  …….
  public void visit(LStm s) {
    s.a.accept(this);
    s.b.accept(this);
  }
  public void visit(AStm s) {
    t.put(s.id, s.a.accept(this));
  }
  public Object visit(DrvExp e) {
    DeriveVisitor dvisitor = new DeriveVisitor(p, t, e.id);
    Exp exp = e.e;
    for (int i = 0; i < e.n; i++)
      exp = (Exp)(dvisitor.visit(exp));
    return exp.accept(this);
  }
  public Object visit(Num e) {
    return new Integer(e.n);
  }
```

```
  public Object visit(Var e) {
    return t.get(e.id);
  }
  …….
}
```

Şekil 12. *EvalVisitor sınıfı*

Visitor arayüzüne sahip bütün sınıflarda, nesne ağacının bütün düğümlerini değerlendirebilecek visit() metotları arasındaki veri geçişi Object nesnesi üzerinden gerçekleştirilmiştir. Bu nedenle, bir değerin hesaplamalarda kullanımını yapabilmek için nesne verisinden ilgili temel tür verisine dönüşümü yapılmalıdır. Örneğin, bir x tamsayısını temsil eden obj=new Integer(x) nesnesinden temel türe geri dönüşüm işlemi x=((Integer)obj).intValue() biçiminde uygulanır.

III. YORUMLAMA

Bu bölümde yorumlayıcı bileşenlerini entegrasyonu ile SaMP dilinde geliştirilen bazı kodlama örneklerine yer verilmiştir. Ayrıca, dilin performansı Java, C ve Haskell programlama dillerine ilaveten Matlab ve Mathematica araçları ile karşılaştırmalı bir analiz yapılarak değerlendirilmiştir.

A. BİLEŞENLERİN ENTEGRASYONU

SaMP dilinin yorumlayıcısı, önceki bölümlerde tanıtılan Ayrıştırıcı, Anlamsal Denetleyici ve Türev Alıcı ve Değerlendirici bileşenlerinin entegrasyonu ile oluşturulmuştur. Interpreter sınıfı ile temsil edilen bu yorumlayıcının ana metodu Şekil 13'de gösterildiği gibi kodlanmıştır.

```
public class Interpreter {
  public static void main(String[] args) {
    try {
      Program p = new Parser(System.in).Prog();
      SymTable t = new SymTable(10000);
      TypeVisitor type = new TypeVisitor(p, t);
      Object res = type.visit(new Fn("main", null));
      if (res != null) {
        EvalVisitor eval = new EvalVisitor(p, t);
        eval.visit(new Fn("main", null));
      }
    }
    catch(ParseException ex) {
      System.out.println(ex.getMessage());
    }
  }
}
```

Şekil 13. *Interpreter sınıfı*

Yorumlayıcı modülleri arasında bulunan sözdizim sınıfları *AST.java*, JavaCC formatında tanımlanan ayrıştırıcı *Parser.jj*, tür denetleyicisi *TypeVisitor.java*, türev alıcısı *DeriveVisitor.java* ve kod değerlendiricisi *EvalVisitor.java* dosyalarına kaydedilmiştir. Aynı dizine yerleştirilen bu modüller yardımıyla yorumlayıcının üretimi aşağıdaki gibi yapılır.

```
$>  javacc Parser.jj
$>  javac *.java
```

SaMP diliyle kodlanan prog.txt dosyasındaki kaynak verinin yorumlaması için ise

```
$>  java Interpreter < prog.txt
```

komutu çalıştırılmalıdır.

B. KÖK BULMA YÖNTEMLERİNİN PROGRAMLANMASI

SaMP sözdiziminin kullanımını göstermek amacıyla fonksiyon köklerini hesaplayan bazı sayısal yöntemlerin kodlaması ve yorumlama sonuçları aşağıda verilmiştir.

Basit İterasyon Yöntemi: Bu yöntem ile f(x)=0 eşitliği x=g(x) biçiminde düzenlenerek kök bulma problemi sabit nokta hesaplama problemine dönüştürülür. g(x) fonksiyonu için hesaplanabilen sabit noktalar f(x) fonksiyonunun kökleri olacaktır. Bir sabit nokta x=x0 başlangıç değerini ve bir mutlak hata sınırlamasını kullanarak aşağıdaki iterasyon ifadesi ile belirlenebilir.

$$x_{n+1} = g(x_n)$$

Yöntemin SaMP dilindeki kodu Şekil 14'te gösterilmiştir; g(x) fonksiyonu, f(x) = exp(x)-3^x+4 fonksiyonundan türetilmiştir.

Program kodu	**Çıktısı**

g(x) = ln(exp(x)+4) / ln(3)
FIX(x) {
  y=g(x);
  print();
  print(x,y);
} | abs(y-x) < 0.001 = y
  | otherwise = FIX(y)
main() { print("x ", "y"); } = FIX(1.0)
	x y
	1.000 1,734
	1,734 2,065
	2,065 2,253
	2,253 2,370
	…….
	2,607 2,608
	2,608 2,609

Şekil 14. *Basit iterasyon yönteminin programlanması*

Yarılama Yöntemi: f(x) fonksiyonu için f(a) ve f(b) değerleri zıt işaretli olacak şekilde bir [a, b] aralığı seçilir. Bu aralığın orta noktası c = (a+b)/2 hesaplanır ve f(c)'nin işaretine göre yeni aralık [a, c] ya da [c, b] olarak alınır. Yöntemin programlaması Şekil 15'te gösterildiği gibi yapılabilir.

Program kodu	Çıktısı

f(x) = exp(x)-3^x+4
BS(x,y) {
  z=(x+y)/2;
  print();
  print(round(x,3), round(z,3), round(y,3));
} | abs(x-z) < 0.001 = z
  | f(x) * f(z) < 0 = BS(x,z)
  | otherwise = BS(z,y)
main() { print("x", "z", "y"); }
  = BS(1.0,3.0)
	x z y
	1000 2.000 3.000
	2.000 2,500 3.000
	2,500 2,750 3.000
	2,500 2,625 2,750
	2,500 2,562 2,625
	2,562 2,594 2,625
	…….
	2,609 2,611 2,613
	2,611 2,612 2,613

Şekil 15. *Yarılama yönteminin programlanması*

Yer Değiştirme Yöntemi: f(x) fonksiyonu için f(a) ve f(b) değerleri zıt işaretli olacak şekilde bir [a, b] aralığı seçilir. Bu aralığın içerisinde bir c noktası, aşağıda verilen iterasyon ifadesi ile belirlenir ve f(c)'nin işaretine göre yeni aralık [a, c] ya da [c, b] olarak alınır.

$$c_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}$$

Yöntemin kodu Şekil 16'da gösterilmiştir.

Program kodu	Çıktısı

f(x) = exp(x)-3^x+4
RF(x,y) {
  z=(x*f(y)-y*f(x))/(f(y)-f(x));
  print();
  print(round(x,3), round(z,3), round(y,3));
} | abs(x-z) < 0.001 = z
  | f(x) * f(z) < 0 = RF(x,z)
  | otherwise = RF(z,y)
main() { print("x", "z", "y"); } = RF(1.0,3.0)
	x z y
	1.000 2,121 3.000
	2,121 2,485 3.000
	2,485 2,581 3.000
	2,581 2,604 3.000
	2,604 2,610 3.000
	2,610 2,611 3.000
	2,611 2,611 3.000

Şekil 16. *Yer değiştirme yönteminin programlanması*

Newton-Raphson Yöntemi: Bir f(x) fonksiyonunun kökü, x=x0 başlangıç değeri ile aşağıda verilen iterasyon formülü kullanılarak hesaplanır.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Yöntemin programlaması Şekil 16'da verilmiştir.

Program kodu	Çıktısı
f(x) = exp(x)-3^x+4 NR(x) {	x y 1.000 7,438

y=x-f(x)/drv(f(x),x);
print();
print(round(x,3), round(y,3));
} | abs(y-x) < 0.001 = y
| otherwise = NR(y)
main() : { print("x", "y"); } = NR(1.0)
	7,438 6,599
	6,599 5,775
	5,775 4,973
	…….
	2,616 2,611
	2,611 2,611

Şekil 16. *Newton-Raphson yönteminin programlanması*

Halley Yöntemi: Bir f(x) fonksiyonunun kökü, x=x0 başlangıç değeri ve aşağıda verilen iterasyon formülü ile hesaplama yapılır.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\left(1 - \frac{f(x_n)f''(x_n)}{2(f'(x_n))^2}\right)^{-1}$$

Yöntem Şekil 17'de gösterildiği gibi programlanabilir.

Program kodu	**Çıktısı**

f(x)=exp(x)-3^x+4
HL(x) {
y=x-f(x) / drv(f(x),x) /
(1-f(x)*drv(f(x),2,x)/(2*drv(f(x),x)^2));
print();
print(round(x,3), round(y,3));
} | abs(y-x) < 0.001 = y
| otherwise = HL(y)
main() { print("x ", "y"); } = HL(1.0)
	x y
	1.000 2,068
	2,068 2,585
	2,585 2,611
	2,611 2,611

Şekil 17. *Halley yönteminin programlanması*

C. KARŞILAŞTIRMALI ANALİZ

Bir dilin sözdizimi matematiksel hesaplamaların tanımlanması ve doğrulanması üzerinde önemli bir etkiye sahiptir. Matematiksel ifade yazım biçimlerine en fazla benzerlik gösteren sözdizimine sahip dil ailesi fonksiyonel dillerdir. Bu dillerde fonksiyonlar için yapılan tanımlamalarda matematiksel fonksiyon davranışlarının yönlendirici bir rolü vardır. Bu nedenle, bilgisayar donanım elemanları geliştiren Intel ve Ericsson gibi şirketler kod doğrulama süreçlerinde sırasıyla Forte [19] ve Erlang [20] fonksiyonel dillerini kullanırlar. Sözdizimin matematiksel notasyonlara yakın olmasına bağlı olarak, kod tanımlama ve doğrulama süreçlerinin yönetimi daha kolay ve hızlı yapılabilmektedir.
SaMP dilinin sözdizim yapısı Haskell dilinden uyarlanarak kodlama sürecinin verimliliği arttırılmıştır.

SaMP diline yönelik yapılacak karşılaştırmalı analiz için C, Java, Haskell, Matlab ve Mathmatica dilleri seçilmiştir. Özellikle sayısal hesaplamalar için tasarlanan Matlab aracının lineer cebir kütüphaneleri C ve türevi olan diller ile Java diline dayandırılmıştır. Mathematica ise simgesel hesaplamalar için tasarlanmış olup fonksiyon kütüphanelerinin geliştirilmesinde Wolfram [21] diline ilaveten C, C++ ve Java dilleri kullanılmıştır. Bu iki aracın performansı, kaynak verinin içereceği hesaplamaların türü ve işlem yüküne bağlı olarak değişir. Programlama dilleri farklı veri türleri

üzerinde farklı performans gösterirler [22, 23]. Bu nedenle, yöntemlerin seçilen dillerdeki değerlendirmesi sadece *double* veri türü üzerinden yapılmıştır.

Programlama dillerinin değerlendirilmesine öncülük eden ve bazı kaynaklarda tasarım prensipleri olarak isimlendirilen çeşitli metrikler vardır [24, 25]. Bu metriklere ilaveten başka dil özelliklerine dayalı değerlendirmeler de yapılabilmektedir [26-29]. Aşağıda bazı dil metrikleri kısaca açıklanarak, seçilen dillerin karşılaştırılmasında kullanılmıştır ve sonuçlar Tablo 2'de sunulmuştur. Metrikler üzerinde önemli ayırt edici etkileri bulunan dil aileleri de karşılaştırmaya dahil edilmiştir.

- Sürdürülebilirlik *(Maintainability):* Programlara yeni özellikler ekleyebilme ve hataların belirlenip düzeltilebilmesi.
- Yazılabilirlik *(Writability):* Dil sözdiziminin kolayca kavranarak programların hızlı bir şekilde geliştirilebilmesi.
- Okunabilirlik *(Readability):* Bir program tarafından icra edilen hesaplamaların kolayca kavranabilmesi.
- Ortogonallik *(Orthogonality):* Dil yapılarının bütün sözdizim bileşenleri içerisinden aynı davranışı gösterebilmesi.
- Tekdüzelik *(Uniformity):* Benzer manalı dil yapılarının benzer davranış gösterebilmesi.
- Genişleyebilirlik *(Extensibility):* Programcılar tarafından dile yeni özellikler eklenebilmesi.
- Etkinlik *(Efficiency):* Programların icra sürelerinin kısalığı ve kullanılan bellek miktarının azlığı.
- Güvenilirlik *(Reliability):* Programların kaynak veri ile belirtilen hesaplamaları yerine getirebilmesi.
- Taşınabilirlik *(Portability):* Programların farklı makineler üzerinde değişiklik yapılmadan derlenebilmesi veya yorumlanabilmesi.

Tablo 2. *Seçilen dillerin değerlendirme metrikleri yardımıyla karşılaştırılması*

	SaMP	**C**	**Java**	**Haskell**	**Matlab**	**Mathem.**
Emirsel	X	X	X		X	X
Fonksiyonel	X			X		X
Nesneye dayalı			X		X	X
Sürdürülebilirlik	X	X	X	X	X	X
Yazılabilirlik	X	X	X	X	X	X
Okunabilirlik	X	X	X	X	X	X
Ortogonallik	X			X		
Tekdüzelik	X		X	X		
Genişleyebilirlik		X	X	X	X	X
Etkinlik	X	X	X	X	X	X
Güvenilirlik	X	X	X	X	X	X
Taşınabilirlik	X		X			

Diğer bir karşılaştırma, seçilen dillerdeki programların icra süreleri ile ilgilidir. Bu amaç için gerçekleştirilecek kod performans ölçümünde önceki bölümde programlaması yapılan kök bulma yöntemleri kullanılmıştır. Seçilen her bir dilde, bu yöntemler için bir fonksiyon tanımlaması yapılmıştır. Daha güvenilir bir performans değerlendirmesi için bu fonksiyonların 10 kez çağrımı yapılarak icra sürelerinin ortalamaları hesaplanmıştır. Simgesel türev almayı gerektiren iki yöntemin

değerlendirmesinde, kökü hesaplanan fonksiyonun türevinin kaynak veriye eklenmesi ya da bu türev işleminin dilin kendisine yaptırılması durumlarına göre iki zaman değeri ölçülmüştür. Tablo 3'te ölçüm sonuçları gösterilmiştir.

Tablo 3. *Kök bulma yöntemlerinin bazı dil çevreleri için icra süreleri*

Yöntem	**SaMP**	**C**	**Java**	**Haskell**	**Matlab**	**Mathematica**
Basit iterasyon	14.2ms	1.8ms	3.3ms	18.2ms	3.8ms	8.8ms
Yarılama	12.0ms	1.4ms	2.7ms	15.3ms	3.2ms	8.2ms
Yer Değiştirme	9.5ms	1.3ms	2.2ms	12.6ms	2.7ms	7.6ms
Newton-Raphson	8.3ms	1.1ms	1.5ms	12.2ms	1.8ms	6.1ms
	10.6ms*	--	--	--	2.4ms*	6.8ms*
Halley	8.2ms	0.8ms	1.2ms	11.4ms	1.5ms	5.8ms
	10.4ms*	--	--	--	1.8ms*	6.2ms*

*Yöntemin içerdiği simgesel türev işleminin yapılması durumunda ölçülen zamanı gösterir.

IV. SONUÇ

Bu çalışmada sayısal hesaplama yöntemlerinin programlanması ve yorumlanmasına yönelik bir programlama dilinin (SaMP) geliştirme süreci anlatılmıştır. Süreç, bir biçimsel gramerin tasarlanması, sözdizim analizi, soyut sözdizim ağacının üretimi, anlamsal analiz ve kaynak verinin yorumlanması gibi birkaç aşamadan oluşur. Dilin sözdizimi, matematiksel ifade yazım biçimlerine dayandırılmıştır. Biçimsel gramer kuralları, LL(1) tabanlı ayrıştırıcıların gereksinimlerine göre düzenlenmiştir. Kaynak verinin sözcük ve ifade yapılarının analizi ile sözdizim ağacının üretimi için gerekli bütün tanımlamalar JavaCC aracı kullanılarak gerçekleştirilmiştir. Anlamsal analiz ve yorumlama aşamalarında, sözdizim ağacı üzerinden değerlendirme yapan Ziyaretçi tasarım deseninin üç farklı gerçeklemesi yapılmıştır. Kaynak kodu verilen birkaç SaMP programı üzerinde, dil yorumlayıcısı tarafından yapılan değerlendirmelerin sonuçları gösterilmiştir. Ayrıca, dil metriklerini kullanarak C ve Java gibi programlama dilleri ile karşılaştırmalı analiz yapılmış ve bazı sayısal yöntemlerin kodu üzerinden koşum zamanları ölçülmüştür.

SaMP dilinin sözdizimi ile genel amaçlı programlamada ihtiyaç duyulan bütün temel dil yapıları kapsanmıştır. Kaynak verinin düzenlenmesinde değişkenler, ifade dizileri, seçim (veya koşul) ifadeleri, döngüler ve fonksiyonlar gibi en temel programlama yapılarının kullanımı mümkündür. Değişkenlere dayalı fonksiyon tanımlamalarında çeşitli matematiksel fonksiyonları (üstel, logaritmik, trigonometrik gibi) içeren ifadelere yer verilebilir. Koşul ifadeleri ile iki ya da daha fazla ifade arasında yapılan seçime bağlı olarak farklı icra yolları oluşturulabilir. Döngü ifadeleri sadece özyineli fonksiyon tanımlamaları içerisinden temsil edilebilir.

Kodlama sürecinde sayısal hesaplamaların gerektirdiği çok farklı matematiksel işlemlerin programlanabilmesini sağlamak için dil sözdizimi genişletilmelidir. Standart girişten veri okuma ve farklı türlere dönüşüm işlemlerinin desteklenmesi önemlidir. Özellikle, dinamik olarak bildirimi yapılan fonksiyonların, sabit nokta iterasyonlarında olduğu gibi değişik biçimlerde düzenlenerek değerlendirilebilmesini sağlayan dil fonksiyonlarına ihtiyaç vardır. Diğer bir ihtiyaç denklem sistemlerinin çözümlerinde kesirli sayı aritmetiğinin kullanılabilmesidir. Bu çeşit gereksinimler, dil

sözdizimine yeni veri türleri ve anahtar kelimeler eklenerek veya bir dil kütüphanesi geliştirilerek karşılanabilir.

V. KAYNAKLAR

[1] J.V.Z. Gathen and J. Gerhard, "Modern Computer Algebra", 3rd ed., Cambridge University Pres, Cambridge, 2013.

[2] MathWorks, (24 Ocak 2019). [Online]. Erişim: http://www.mathworks.com.

[3] Maplesoft, (24 Ocak 2019). [Online]. Erişim: http://www.maplesoft.com.

[4] Wolfram, (24 Ocak 2019). [Online]. Erişim: http://www.wolfram.com.

[5] W. A. Martin and R. J. Fateman, "The MACSYMA system", Proceedings of the 2nd ACM symposium on Symbolic and Algebraic Manipulation. ACM, pp. 59-75, 1971.

[6] S. H. Sadat-Mohtasham and A. A. Ghorbani, "A language for high-level description of adaptive web systems", Journal of Systems and Software, vol. 81, no. 7, pp. 1196-1217, 2008

[7] S. Louise, "An OpenMP backend for the ΣC streaming language", Procedia Computer Science, 108, pp. 1073-1082, 2017.

[8] N. Kokash, S. L. Moodie, M. K. Smith and N. Holford, "Implementing a domain-specific language for model-based drug development", Procedia Computer Science, 63, pp. 308-316, 2015.

[9] E. G. Pinho and F. H. de Carvalho Junior, "An object-oriented parallel programming language for distributed-memory parallel computing platforms", Sci. Comput. Program., 80 pp. 65-90, 2014.

[10] J. R. Levine, J. R. Levine, T. Mason and D. Brown, "Lex & Yacc", O'Reilly Media, Inc., Sebastopol, CA, USA, 1992.

[11] E. M. Gagnon and L. J. Hendren, "SableCC, an object-oriented compiler framework", In TOOLS USA 98 (Technology of Object-Oriented Languages and Systems), IEEE, 1998.

[12] V. Kodaganallur, "Incorporating language processing into Java applications: A JavaCC tutorial", IEEE Software, vol. 21, no. 4, pp. 70–77, 2004.

[13] A. J. Dos Reis, "Compiler Construction Using Java, JavaCC, and Yacc", IEEE Computer Society, Inc., 2012.

[14] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 7th ed., McGraw Hill, 2015.

[15] B. W. Kernighan and D. M. Ritchie, "The C Programming Language", Prentice Hall Professional Technical Reference, 1988.

[16] K. Arnold, J. Gosling, and D. Holmes, "The Java programming language", Addison Wesley Professional, 2005.

[17] S. P. Jones. "Haskell 98 language and libraries: the revised report", Cambridge University Press, 2003.

[18] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Reading, MA, 1995.

[19] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 9, pp. 1381-1405, 2005.

[20] J. Armstrong, "Erlang", Communications of the ACM, vol. 53, no. 9, pp. 68–75, 2010.

[21] Wolfram Language & System Documentation Center (24 Ocak 2019). [Online]. Erişim: reference.wolfram.com.

[22] H. Chen, "Comparative Study of C, C++, C# and Java Programming Languages", M.S. thesis, Info. Tech., VAMK Univ., Vaasa, Finland, 2010.

[23] P. Gooyal, "A comparative study of C, Java, C# and Jython", M.S. thesis, School of Comp., UNF, Florida, USA, 2014.

[24] L. K. C. Louden, Programming Languages: Principles and Practices, *Cengage Learning*, 2011

[25] R. Harper, Practical Principles for Programming Languages, 2nd ed., *Cambridge University Press*, 2016.

[26] S. Nanz, and C. A. Furia, "A Comparative Study of Programming Languages in Rosetta Code", 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 1, pp. 778-788, 2015.

[27] B. O. Oguntunde, "Comparative analysis of some programming languages", Transnational Journal of Science and Technology, vol. 2, no. 5, pp. 107-118, 2012.

[28] K. Biswa, B. Jamatia, D. Choudhury and P. Borah, "Comparative analysis of C, Fortran, C# and Java Programming Languages", International Journal of Computer Science and Information Technologies (IJCSIT), vol. 7, no. 2, pp. 1004-1007, 2016.

[29] R. Naim, M. F. Nizam, S. Hanamasagar, J. Noureddine, M. Miladinova, "Comparative studies of 10 programming languages within 10 diverse criteria – a Team 10 COMP6411-S10", Rep. arXiv:1008.3561, 2010.