İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

GENETİK PROGRAMLAMA İÇİN ALANA ÖZGÜ PROGRAMLAMA DİLİ

YÜKSEK LİSANS TEZİ

Cem Başar ÇAYIROĞLU

Anabilim Dalı : İleri Teknolojiler

Programı : Bilgisayar Bilimleri

Tez Danışmanı: Yrd. Doç. Dr. A. Şima UYAR

EYLÜL 2010

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

GENETİK PROGRAMLAMA İÇİN ALANA ÖZGÜ PROGRAMLAMA DİLİ

YÜKSEK LİSANS TEZİ

Cem Başar ÇAYIROĞLU
(704071005)

Tezin Enstitüye Verildiği Tarih : 15 Eylül 2010
Tezin Savunulduğu Tarih : 20 Eylül 2010

Tez Danışmanı : Yrd. Doç. Dr. A. Şima UYAR (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. Nadia ERDOĞAN (YTU)
Dr. Ender ÖZCAN (NOTT)

EYLÜL 2010

ÖNSÖZ

Çalışmamda danışmanım Şima Uyar ve Turgut UYAR'a teşekkürü bir borç bilirim.

Eylül 2010

Cem Başar ÇAYIROĞLU
(Bilgisayar Mühendisi)

İÇİNDEKİLER

Sayfa

KISALTMALAR

GA **:** GA
GP **:** Genetik Programlama

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

GENETİK PROGRAMLAMA İÇİN ALANA ÖZGÜ PROGRAMLAMA DİLİ

ÖZET

Genetik Programlama[1] (GP) evrimsel algoritmaların bir formu olup, bireylerin temsil ediliş şekli ağaçtır. Genetik programlama ile uğraşan araştırmacılar çalışmalarının tasarım, uygulamaya geçiş ve de test aşamalarında birçok yardımcı araç kullanırlar. Bu araçlar içinde C++ ve Java tabanlı çatılar son yıllarda en çok kullanılan çatılardır. Bu çatılar GP projelerinde uygulamaya geçiş süresini kısaltmakta fakat genelde taban dilleri hakkında derin bilgiye sahip olmayı gerektirmektedir. Dahası, taban programlama dillerinin limitleri yüzünden, araştırmacılar GP alanında projelerinde iyi bir soyutlama yapamamaktadır.

Alana özgü diller [2] (DSL) programlama dilleri olup o alana özel olarak tasarlanmıştır. Örnek olarak, SQL bir alana özgü dil olup veritabanlarına özel olarak geliştirilmiştir. Genel amaçlı programlama dilleri ile kıyaslamak gerekirse alana özel diller o alan için daha çok anlatımcıdır çünkü o alanda olan bir algoritmayı anlatabilecek daha doğal bir yol sağlarlar. Bu anlatımcılığı arttıran özeliği sayesinde araştırmacıların üretkenliği artar ve tecrübesiz programcıların bu alanda daha kolay uygulama geliştirmesine yardımcı olur.

Bu çalışmada, genetik programlama için alana özgü bir programlama dili geliştirdik. Alana özgü dil geliştirmek için Xtext[3] adlı Eclipse platformu üzerinde çalışan programlama dili geliştirme aracı kullandık. Xtext gramer denetimi ve de kod üretimi özellikleri vardır. Kod üretimi özelliği sayesinde genel amaçlı programlama dilleri, Xtext tarafından geliştirilmekte olan alana özgü dilden üretilebilir. Alana özgü dil geliştirmek için, ilk olarak genetik programlama alanı analiz edildi. Daha sonrasında geliştirilecek olan dilin grameri tasarlandı. Daha sonraki aşamada ise alana özgü dili Java programlama diline çevirecek kod üretici modül geliştirildi. Geliştirilen alana özgü dil GP 'de bulunan üst seviye operasyonları desteklemektedir. Bunlar arasında; seçme, çaprazlama, mutasyon vardır. Bunun yanında daha alt seviye ağaç işlemlerini de desteklemektedir. Geliştirilen alana özgü dili kullanarak genetik programlama araştırmacıları karışık ağaç işlemlerini kolayca yapabilirler. Geliştirilen primitif komutlar ile araştırmacılar yeni GP operasyonları tanımlayabilirler.

A DOMAIN SPECIFIC LANGUAGE FOR GENETIC PROGRAMMING

SUMMARY

Genetic programming is a specific topic of genetic algorithm paradigm. In genetic programming domain, individuals are programming codes. The distinctive feature of genetic programming is individual types and output of the algorithm. Individuals are code parts with varying size and shape and output is a ready-to-use computer programs. The main goal is generating computer programs (usually based on high level language) automatically.

Domain-specific languages are languages dedicated to a domain. Domain specific languages commands are really close to the domain related terms. When you think about SQL programming language its key words all related to database terms. By using only SQL you can not write a regular imperative computer programs. However, it is hard to deal with database without SQL. DSL is becoming more popular due to rise of domain base design. They are more expressive comparing to a general-purpose language in their domain. Therefore, using DSL increases productivity and lowers maintenance cost. By reducing required programming experience, more people can understand program code written in DSL compared to general-purpose language. Using existing general-purpose language is better if the domain specific language does not have enough advantages. General-purpose language is robust and popular. In addition, developing a DSL is hard because it requires domain knowledge and DLS development experience.

Fitness calculation, crossover and mutation are done on these code parts. In order to apply genetic algorithm operations to code parts, representation of an individual should be tree. Genetic programming researchers use several programming tools in order to design, implement and verify their study. They use tools such as java or C language programming frameworks that become very popular in last years. Frameworks decrease development time of genetic algorithm implementation but usually required a good knowledge about their base programming language. Although frameworks are well designed, due to base programming language, researchers may not make a good abstraction of a genetic programming problem. In this paper, a new domain specific language (DSL) will be introduced in order to increase productivity of researchers, to use high level abstraction of genetic programming domain and to increase maintainability of their projects. To develop a new DSL, decision, analysis, design, implementation, and deployment steps should be executed. In our study, these steps were followed and will be explained in detail.

DLS should fit these genetic programming requirements: individual representation, terminal and operator representation, individual evaluation and evaluator, generate random individual and population, mutation, cross-over, tree operations such as prune, random grow, population representation, probability support, loops, random selection and strategies, offspring support, built in individual and evaluators such as symbolic regression etc.

1 GİRİŞ

Genetik programlama GA ’nın bir alt dalıdır. Genetik programlamayı diğer GA 'lardan ayıran en büyük özellik bireylerin türü ve algoritmanın çıktısıdır. Genetik programlamada bireyler çalıştırılabilir program kodlarıdır. Bireyler değişebilir boyutta ve şekilde program kodlarından oluşmakta olup çıktısı ise kullanılabilir bilgisayar programlarıdır. En temel amaç, bilgisayar programlarını (genelde üst seviye dillerle yazılmış) sorunu adresleyecek şekilde otomatik olarak üretmektir.

Başarım hesaplama, çaprazlama ve mutasyon işlemleri bu kod parçalarına uygulanmaktadır. Genetik programlamada bireyler ağaç yapısı ile tutulur. Genetik programla ile ilgilenen araştırmacılar projelerini tasarlamak, gerçeklemek ve test etmek için birçok genetik programlama aracı kullanmaktadır. Java ve C programlama dili tabanlı anaçatılar son yıllardaki en popüler geliştirme araçlarıdır. Bu anaçatılar geliştirme zamanını düşürmektedir, ancak genelde kullanılabilmeleri için yazıldığı dil hakkında iyi bir tecrübe gerektirmektedir. Ek olarak anaçatılar iyi tasarlanmış olmasına rağmen soyutlama yetenekleri yeterli olmayabilir. Bu çalışmada, araştırmacıların üretkenliğini arttırmak, daha üst düzey soyutlama yapabilmelerini sağlamak ve projelerinin sürdürülebilirliğini arttırmak için genetik programlama için kullanılacak yeni bir alana özgü dil sunulmuştur. Alana özgü dil geliştirmek için takip edilen adımlar: karar, analiz, tasarım, gerçekleme ve yükleme aşamalarıdır. Çalışmamızda bu adımlar takip edilmiştir ve detaylı olarak açıklanacaktır. Tezin amacı genetik programla için kullanılacak olan alana özgü bir dil geliştirmektir. Geliştirilecek alana özgü dil genetik programlamanın şu gereksinimlerini karşılamalıdır: birey, terminal ve operatör tanımlama, birey değerlendirme (başarım hesaplama) ve değerlendirici tanımlama, rasgele birey ve toplum üretme, mutasyon ve çaprazlama, ağaç işlemleri (budama, rasgele büyüme) yapabilme, toplum tanımlama, olasılıklı işlem yapabilme, döngü, rasgele seçim, önden tanımlı değerlendiriciler (sembolik regresyon).

Tezin 2. bölümünde GA 'lar ve programlama hakkında bir literatür araştırması yapılmıştır. Ardından 3. bölümde ise alana özgü dil nedir ve geliştirme adımları

detaylı olarak incelenecektir. 4. bölümde ise Genetik programlama için geliştirdiğimiz dil bu adımlar takip edilerek anlatılacaktır. Bu adımlar; karar, analiz, tasarım ve gerçeklemedir. 5. bölümde sonuç ve yorumlara yer verilecektir.

2 GENETİK PROGRAMLAMA

2.1 Giriş

Bir toplumda ortama en çok uyum sağlayan bireyler diğer bireylere göre hayatta kalma ve çoğalma şansıları daha fazla olan bireylerdir. Bu uyum miktarı başarım hesabı ile bulunur. Bütün biyolojik yapıların bir başarım hesaplama fonksiyonu bulunur ve bu onun diğer bireylere göre ne kadar güçlü olduğunu gösterir. Bu başarım değerinin doğası gereği; seçilme, çaprazlama ve mutasyon otomatik olarak gerçekleşir [6].

GA genetik evrim süreçlerini temel alır. Bu süreçler: seçilme, çaprazlama ve mutasyondur. GA karmaşık optimizasyon problemlerini çözmek için kullanılır. GA kullanabilmek için, çözüm kümesi olmalıdır ve her çözüm için bir kromozom tanımlanması gerekmektedir. Kromozom olarak tanımlanmasının sebebi mutasyon ve çaprazlama işlemlerini yapabilmek içindir.

GA 'nın ana akışı aşağıdaki gibidir:

Adım 0

Rasgele bir toplum üret ve değerlendir.

Adım 1

Toplumdan ebeveyn bireyler seç (başarım değerleri temel alınarak seçim yapılabilir).

Adım 2

Çaprazlama ile yeni bireyler yarat.

Adım 3

Belli bir olasılıkla yeni oluşan bireyde mutasyon işlemi yap.

Adım 4

Yeni oluşan bireyi topluma ekle, toplumu değerlendir ve küçült, eğer belli bir koşul sağlanmadıysa (istenen başarım düzeyi) adım 1'e geri dön.

GP evrimsel bir algoritmadır ve genetik algoritmaların bireyleri bilgisayar programları olan özelleşmiş şeklidir. Temelleri 1985 yılında Nichael L. Cramer tarafından "ağaç tabanlı genetik programlama" geliştirilerek atılmıştır. John R. Koza tarafından geliştirilerek optimizasyon ve arama problemlerinde kullanılmıştır.

Amaç evrim süreçlerini kullanarak belli bir amaca hizmet edecek bilgisayar programları üretmektir. GP kromozomların temsil edilişi ağaç şeklinde olur. Ağaç rekürsiviteden dolayı kolay şekilde değerlendirilebilir.

Ağacın düğümlerinde operatörler, yapraklarında terminaller bulunur. Programlama dili için operatörler komut, terminaller ise parametre veya değişken olabilir.

GP 'da çaprazlama iki ağacın birer düğümünü seçip, alt ağacı ile yer değiştirmekle olur. Mutasyon ise rasgele seçilen bir düğümün alt ağacını kesip, rasgele o düğümden büyüterek olur.

2.2 GP Adımları

GP ile program kodu üretmek için uygulanması gereken adımlar aşağıdaki gibidir:

Adım 0

Rasgele toplum üretilir. Toplumdaki bireyler ağaç yapısında olmalı. Ağaç düğümleri fonksiyonlardan, yapraklar ise terminallerden oluşmalıdır.

Adım 1

Toplumdaki bütün programları koştur ve başarım değerlerini hesapla.

Adım 2

Çaprazlama ve mutasyon işlemleri kullanarak yeni bir toplum oluştur.

Adım 3

Bitiş koşulu sağlamadıysa Adım 1'e geri dön.

2.3 GP 'dan Örnekler

GP kullanılan bazı alanlar şunlardır:

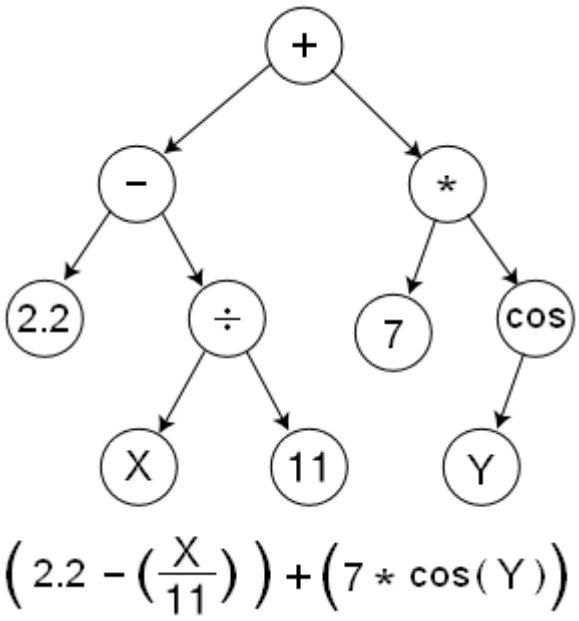
Optimal Kontrol: GP kullanılarak bir kontrol stratejisi geliştirilmiştir. Araç GP 'nın ürettiği program ile en kısa yoldan hedefine ulaşabilir.

Robotik Planlama: Engebeli ve büyük boşluklu bir arazide robot programı ile üretilmiştir.

Sembolik Regresyon: Örnek veriyi elde etmek için bir matematiksel ifade üretilebilir

2.4 Sembolik Regresyon

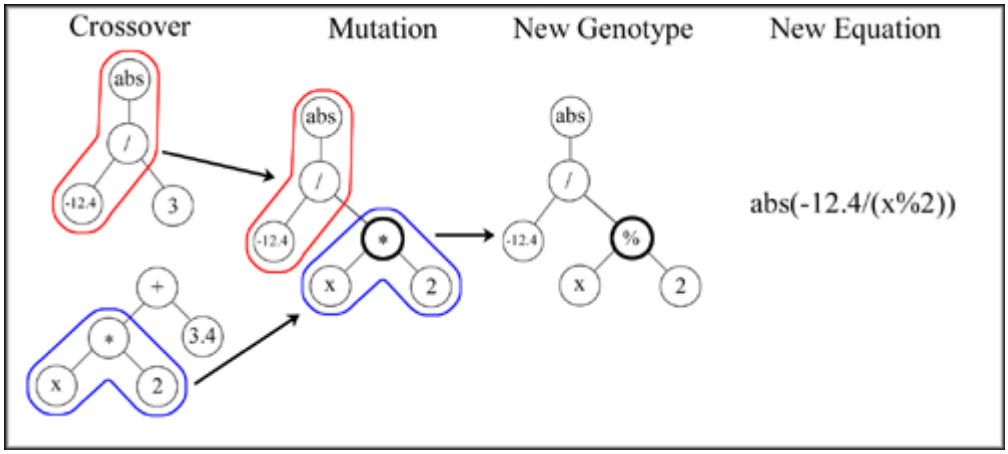
GP adımları açıklamak için sembolik regresyon problemi ele alınacaktır. Sembolik regresyonda amaç belli bir veriden yola çıkarak matematiksel bir denklem ifade etmektir. Veri, belli girdilere karşılık belli bir çıktıları içerir. Bulunan matematiksel ifade bu çıktıları aynı girdilerle minimum hata ile bulmalıdır. Bu denklem artı, eksi, bölme, çarpma gibi operatörlerden oluşur. GP 'da bireyler ağaçtır. Şekil 2.1 bir denklemin ağaç şeklinde ifadesini göstermektedir. Artı, eksi, bölüm operatörleri iki değer, sinüs operatörü ise bir değer almaktadır. Bütün operatörler ağacın iç kısmında yer almaktadır. Değişken ve değerler ise yapraklarda yer almaktadır.



Şekil 2.1 : Sembolik regresyon ağacı [4]

Nesil metodu klasik genetik algoritmada kullanılan metotla aynıdır. Çaprazlama ve mutasyon yeni birey yaratmak için kullanılmaktadır.

Çaprazlama, ağaçlardan rasgele iki düğüm seçilerek yapılır. Bu iki noktadan alınan alt ağaçlar yer değiştirilir. Mutasyon ise belli bir olasılıkla gerçekleştirilir. Mutasyonda rasgele bir düğüm seçilir ve de koşullara bakılarak bir rasgele düğümle yer değiştirilir. Şekil 2.2 de çaprazlama ve de mutasyon işlemlerinin nasıl yapıldığını görebilirsiniz.



Şekil 2.2 : Çaprazlama ve mutasyon

3 ALANA ÖZGÜ DİLLER

3.1 Giriş

Alana özgü dilleri tanımlamak için net bir ifade yoktur [14]. Alana özgü dil çok genel bir ifade olup, bağımsız bir programlama dili olacağı gibi genel amaçlı dillerle yazılan bir alt dil de olabilir [8,9,10] . Alana özgü diller belli bir alana göre özelleşmiştir ve alana özelleşmiş artan yazılım ihtiyacını adresler [12]. Bu diller hizmet ettiği alana ait unsurlara çok yakındır ve problem ile programın arasındaki semantik mesafeyi azaltır [13]. Örnek olarak SQL programlama dilini düşündüğümüzde dilin yapısı doğrudan veritabanını ilgilendiren işlemleri kapsar. SQL kullanarak bir imparativ program geliştiremezsiniz. Fakat veritabanı ile çalışmak SQL olmadan çok zordur. Alana özgü diller alan tabanlı tasarım yaygınlaştıkça popüler hale gelmeye başlamıştır. Alana özgü dillerin kendi alanlarında diğer genel amaçlı dillere kıyasla ifade yetenekleri daha güçlüdür. Bundan dolayı belli bir alanda o alana özgü bir dil kullanmak üretkenliği artırır ve maliyetleri düşürür. Aynı zamanda genel programlamaya kıyasla daha az tecrübe gerektirdiğinden, insanlar alana özgü dilleri daha kolay şekilde anlarlar. Genel amaçlı dillerin kullanımına, alana özgü dilin özelliklerinin yetmediği durumlarda başvurulmalıdır çünkü genel amaçlı diller daha popüler ve sağlamdır. Alana özgü bir dil geliştirmek ise çok iyi programlama yetisi ve alan bilgisi gerektirdiğinden dolayı çok zordur.

Alana özgü dil kullanmanın dezavantajları [7]:

Dil tasarlamanın ve gerçeklemenin karmaşık bir iş oluşu: Bu kuşkusuz ki alana özgü bir dil geliştirmenin en önemli dezavantajıdır. Bir derleyici veya önişlemci geliştirmek veya var olanlardan kullanmak çok zordur. Ayrıca derleyici veya önişlemci dil ile ilgili problemlerle de ilgilenmek zorunda kalınmaktadır. Bu ek olarak gelen geliştirme ihtiyacı durumu iyice karmaşıklaştırır.

Bakım zorluğu: Geliştirilecek olan derleyici veya önişlemci karmaşık olacağından bakımı da zor olacaktır.

Dili öğretme maliyeti: Dilin kullanıcıları tecrübeli olmadığından dili öğretme maliyeti olacaktır.

Yavaş uygulamalar: Diğer genel amaçlı diller görmediğimiz optimizasyonları yaparak programın daha hızlı çalışmasını yapabilir. Geliştirilecek olan dilin bu yetenekte olması zordur.

Alana özgü dil kullanmanın avantajları:

Üretkenlik artışı: Alana özgü diller programcının daha hızlı geliştirme yapmasını sağlar. Bu dil geliştirmeye iten en büyük nedendir.

Bakım kolaylığı: Alana özgü dillerde yazılmış uygulamalar altyapının karmaşıklığından arındırılmıştır. Programcı programlama kodu okurken anala özgü olduğundan yapılan işlemi daha rahat anlar.

Potansiyel geliştiricilere ulaşmak: Alanla uğraşan fakat programlama bilgisi yeterli olmayan kullanıcılar geliştirilen dili kullanabilir.

Alan bilgisi gelişimi: Geliştirmeyi yapan şirket veya kuruluşta alan hakkında bilgi düzeyi artar.

3.2 Alana Özgü Dil Geliştirmek

Alana özgü dil geliştirmek için uygulanan adımlar: karar, analiz, tasarım, gerçekleme ve yüklemedir. [2]. Alana özgü diller soyut [2] ve anlatımcıdır [15]. Analiz ve tasarım sırasında geliştiriciler alan hakkında çok iyi bilgi sahibi olmalıdır. Alanın ilgili kavramları dilde doğrudan bulunmalıdır [16].

3.2.1 Karar

Alana özgü dil geliştirmek için verilen yatırım kararı çok zor bir iştir. Yatırımın geri dönüşü olması gerekir. Dahası, alana özgü dil geliştirip kullanmaya başladıktan sonra eksiklikleri görüp geri dönmek çok daha zordur. [2]

3.2.2 Analiz

Analiz fazında alan uzmanları alan analiz metotlarını kullanarak alan hakkında bilgi toplarlar [17,18,19]. Analiz aşamasında ilgili alan çok iyi bir şekilde tanımlanmalı ve alan bilgisi toplanmalıdır. Teknik dokümanlar, alanla uğraşan uzmanların görüşleri, var olan alana hizmet eden bilgisayar programları ve kaynak kodları, müşterilerin tavsiyeleri analiz aşamasında veri olarak kullanılır. Bütün adımlar içerisinde analiz aşaması en önemli adımdır. Bu aşamada yapılan bir hata diğer adımlarda düzeltildiğinde çok daha maliyetli olacaktır. [2]

Analiz aşamasın çıktısı diğer adımda kullanılacak olan dokümanlardır. Tasarım aşamasında kullanılacak olan dokümanlar:

- Alan tanımı yapan kapsam dokümanı
- Alana özel kavramları tanımlayan doküman
- Alana özel kavramların birbirleri ile olan ilişkilerini tanımlayan doküman

3.2.3 Tasarım

Alana özgü dil, çok büyük bir alan bilgisinin program derleyicisine ve çalışma zamanı sistemine aktarılmasıdır. Bundan dolayı alana özgü dil tasarlamak daha çok alanı anlamaktır [11]. Alana özgü bir dil tasarlamak için var olan başka bir dilin yapısından esinlenilebilir. Bu tekniği kullanmanın avantajı var olan programlama dilline yakınlıktan dolayı alana özgü dilin kullanıcıları da dile kolay uyum sağlayacaktır. Ancak bu durum, var olan programlama dili ile pek tecrübesi olmayan bir kişi için dezavantaja dönüşecektir.

Alana özgü dil geliştirmenin bir yolu ise var olan bir dili genişletmektir. Genişletilen dil geliştirme yapıldıktan sonra alana daha yakın olacaktır. Genelde genişletilen dil eski özellikleri korumaktadır. Burada yapılan iş alanla ilgili özelliklerin var olan bir dile entegre edilmesidir.

Alana özgü geliştirmede kullanılan son teknik ise başka var olan dillerden hiçbirisiyle bağlantısı olmayan bir dil geliştirmektir. Genelde yöntem olarak en zor olanı budur. Fakat dilin tasarımı iyi bir şekilde yapılırsa alana en uygun dil bu yöntemle elde edilir.

3.2.4 Gerçekleme

Gerçekleme aşamasında tasarladığımız dilin çalışacağı ortamı geliştiririz. Bir alana özgü dili yorumcu kullanarak çalıştırabiliriz. Dilimiz için bir yorumcu yazarsak dil çok daha esnek ve kontrollü olabilir. Genel amaçlı dile çeviren bir araç geliştirmek de başka bir alternatiftir olarak düşünülebilir.

Başka dile çevirici geliştirmenin ve kullanmanın avantajları [2]:

- Sentaks alana yakın olur
- Hata ayıklama ve adım adım çalıştırma kolay olur
- Analiz, doğrulama ve optimizasyon kolay olur

Başka dile çevirici geliştirmenin ve kullanmanın dezavantajları:

- Derleyici veya çevirici geliştirmek karışık ve de uzun süren bir işlemdir.
- Geliştirilen dil genel amaçlı dillere uzak olduğundan daha çok risk taşır.
- Eğer dil tasarlanırken genel amaçlı bir dile çok benzetildiyse, dil kullanıcıları bu ara katmanı kullanmak yerine genel amaçlı dili kullanacaktır.

Var olan başka bir dile gömülü tekniği de dil geliştirmek için kullanılabilir. Burada bir dilin fonksiyonları yerine getirmek için genel amaçlı dillerden yararlanılır. Kütüphane fonksiyonları geliştirmeye benzer şekilde geliştirme yapılır. Dil başka bir programlama dilinin veri tiplerini kullanır.

Gömülü yöntemin avantajları:

- Geliştirme esnasında var olan dil işleyiciler kullanıldığından geliştirme zamanı kısadır.
- Geliştirilen dil kullanılan genel amaçlı dilin de özelliklerini taşır.
- Genel amaçlı dilin alt yapısı kullanılabilir.
- Genel amaç dile yakınlıktan dolayı daha kolay öğrenilir.

Gömülü yöntemin dezavantajları:

- Sentaks alana uzaktır
- Var olan dil genişletilirken eski özellikleri ezilebilir.

- Çalışma esnasında oluşan hatalar var olan programlama dili türünden oluşacaktır, alanla ilişkilendirmek zordur.

3.3 Alana Özgü Dil Geliştirme Araçları

Alana özgü dil geliştirmek hem alan hem de programlama bilgisi gerektirdiğinden çok zordur. Dil geliştiriciler dil geliştirme araçlarından faydalanabilir. Bu araçlar ile dil için sentaks kontrol eden bir editör, kod çevirici veya yorumlayıcıyı otomatik olarak yaratabilir.

Xtext alana özgü dil geliştirmek için kullanılan bir araçtır. Xtext kullanarak geliştirici kolayca dil tasarımı yapıp bir editör ve kod çevirici geliştirebilir. Xtext bir Eclipse uygulama geliştirme platformunda çalışan eklentidir.

Xtext ile üretilebilen editörün bazı özellikleri aşağıdadır [3]:

Kod renklendirme

Xtext kodun okunurluğunu arttırmak için koda renklendirme yapabilen bir editör üretmenizi sağlayabilir. Geliştirici isterse belli anahtar sözcüklerin rengini ve de fontunu değiştirebilir. Hatalı kodu kırmızı ile renklendirerek geliştiriciyi hataya doğru yönlendirebilir.

Kod içinde yönlendirme

Xtext ile üretilen editörde ctrl+sağ tık ile kod içinde dolaşmam mümkündür.

Kod tamamlama

Xtext ile üretilen editörde ctrl+boşluk ile dilin sentaksına uygun şekilde kod tamamlatmak mümkündür.

4 GENETİK PROGRAMLAMA İÇİN ALANA ÖZGÜ DİL GELİŞTİRMEK

Alana özgü dil geliştirmek için daha önceden bahsedildiği gibi sırası ile karar, analiz, tasarım, gerçekleme ve de yükleme aşamalarından geçmek gerekir. Çalışmamızda bu aşamalar sırası ile uygulanmıştır. Bu bölümde aşamalar detaylı olarak anlatılacaktır.

4.1 Karar

GP ile uğraşan araştırmacılar genel amaçlı birçok anaçatı veya programlama aracı kullanmaktadır. Araç ve anaçatılar genel amaçlı tasarlandıklarından çalışmaları gerçekleme zor bir olmaktadır. Araç ve anaçatıları kullanmak için çalışma ile ilgisiz emek harcanır. Bu emeği en küçükleme için alan özgü dil geliştirme kararı alınmış ve bu kapsamda çalışma yapılmıştır.

4.2 Analiz

Analiz safhasından araştırmacıların kullandıkları araç ve programlama dilleri incelenmiştir. Araştırmacıların ihtiyaçlarını da ne ölçüde karşıladığına bakılmış ve GP için bir dilde hangi özellikler olması gerektiği bulunmuştur.

Bu adımda GP ile ilgili temel kavramlar üzerinde durulmuştur. Bunlar arasında, birey tanımlama, terminaller, operatörler, nesil üretme, ağaç işlemleri, mutasyon, çaprazlama vardır. Geliştirilecek olan dil bu kavramlara doğrudan destek vermelidir.

Dilin destek vermesi gerek kavramlara aşağıda listelenmiştir:

- Birey soyutlama
- Toplum soyutlama
- Çocuk soyutlama
- Terminal ve operatör tanımlama
- Birey değerlendirme
- Rasgele toplum ve birey üretme

- Mutasyon gerçekleme
- Çaprazlama yapabilme
- Temel ağaç işlemleri yapabilme
- Olasılıklı işlem yapabilme
- Döngü kurabilme
- Rasgele seçim ve seçim stratejisi belirleme
- Popüler problemlere hazır değerlendiriciler içerme
- Değerlendirici ara yüzü içerme ve gelişime açık olma

4.2.1 Birey soyutlama

GP 'da birey ağaç şeklinde temsil edilir. Bu özellikten dolayı araştırmacılar genelde ağaç operasyonları ile uğraşır. Dil bu noktada birey için bir soyutlama yapması gerekir. Çünkü araştırmacı için birey bir çözümdür bir ağaç olmamalıdır. Ağaç işlemleri araştırmacıdan gizlenmelidir. Araştırmacılar dil ile geliştirme yaparken bireyle ilgili bir operasyon yaptığını hissetmelidir.

4.2.2 Toplum soyutlama

GP 'da toplum temel kavramlardan bir tanesidir. Toplum soyutlanmalıdır. Toplumu ilgilendiren birey ekleme gibi işlemler en üst düzeyde araştırmacıdan yalıtılmalıdır. Araştırmacı liste veya ağaç işlemleri ile uğraşmamalıdır.

4.2.3 Çocuk soyutlama

Algoritmada iki tane ebeveyn çözümden mutasyon ve çaprazlama ile çocuklar oluşturulur. Bu çocuklar topluma katılmadan önce geçici bir yerde tutulmalı ve de topluma belli bir stratejiye göre katılmalıdır.

4.2.4 Terminal ve operatör tanımlama

GP ‘da bireyi oluşturan ağaç terminal ve operatörlerden oluşur. Terminaller ağacın yaprakları, operatörler de ağacın içi düğümleridir. Araştırmacılar terminal ve operatörlerin yapısını tanımlamak durumdadır. Terminal ve operatörleri isimlendirmeli ve de kısıtlarını belirtmelidir. Örneğin, terminallerin hiç çocuğu olmazken operatörlerin belli sayıda çocuğu olabilir. Araştırmacı geliştirilecek dil ile terminal ve operatörleri çocuk sayıları ile belirleyebilmelidir. Dil daha sonrasında tüm ağaç işlemlerinde araştırmacıdan kontrolleri yalıtmalıdır.

4.2.5 Birey değerlendirme

Genetik algoritmalarda ve dolayısı ile GP ‘da birey değerlendirme en temel kavramlardan biridir. Dil belli bir anahtar sözcükle bu işlemi daha önceden seçilmiş bir değerlendirici kullanarak yapabilir. Değerlendiricilerden daha sonraki bölümlerde daha detaylı bahsedilecektir.

4.2.6 Rasgele toplum ve birey üretme

Rasgele birey ve toplum üretmek GA ’yı uygularken ilk aşamada yapılan işlemdir. Algoritmaya bağlı olarak daha sonradan da araştırmacı rasgele birey üretmek istemeyebilir. Üretilen rasgele birey kısıtlara uygun olmalıdır. Geliştirilecek olan dil belli anahtar sözcüklerle bu işlemlere destek vermelidir.

4.2.7 Mutasyon

Mutasyon bireye uygulanan bir işlemdir. Mutasyonun sonucunda oluşan bireyin kısıtlara uygun olduğu garanti edilmelidir. Birden çok mutasyon uygulama tekniği bulunmaktadır. Dolayısı ile geliştirilecek dil belli temel ağaç işlemlerine de destek vermelidir.

4.2.8 Çaprazlama

Çaprazlama mutasyon gibi bireye uygulanan ve de birçok tekniği olan bir işlemdir. Dil temel ağaç işlemlerine destek vererek araştırmacıya çaprazlama yapma olanağı sunmalıdır.

4.2.9 Temel ağaç işlemleri

Araştırmacı ağaç olan bireyde temel belli işlemleri yapmak isteyebilir. Bular ağaçtan bir alt ağaç alma, alt ağacı belli bir noktadan birleştirme, budama ve rasgele büyütme işlemleridir. Dil anahtar sözcüklerle bu işlemlere destek vermelidir.

4.2.10 Olasılıklı işlemler

GA 'larda bazı işlemler belli olasılıklara göre yapılmak istenebilir. Genel amaçlı dillerde bu rasgele sayı üretip bunun üzerinden aritmetik işlem yapılarak bu işlem gerçeklenir. Geliştirilecek dil bu işleme doğrudan destek vermelidir.

4.2.11 Döngüler

GA, döngüler içinde gerçeklenir. Örnek olarak, algoritma belli bir koşul sağlamadıkça yeni nesiller yaratıp toplumu değiştirir. Bu işlem döngü içinde gerçeklenir. Bunun dışında kullanıcı döngüleri toplum içinde bütün bireyleri taramak ve bireylere bir işlem uygulamak isteyebilir. Bu sebeplerden dolayı geliştirilecek dil döngülere destek vermelidir.

4.2.12 Rasgele seçim ve seçim stratejisi

GA 'da toplumdan veya bireylerin oluşturduğu bir gruptan rasgele seçim yapılabilir. Bu seçim işlemi direk eşit olasılıklarla veya başka bir yöntem kullanılarak yapılmak istenebilir. Dil bu operasyonlara destek vermelidir.

4.2.13 Popüler problemlere hazır değerlendiriciler içerme

Bir bireyin başarım değerini hesaplamak tamamen probleme özgü bir işlemdir. Araştırmacı kendi değerlendiricisini belli bir ara yüz kullanarak yazmalıdır. Geliştirilecek olan dilde dili öğrenmek ve tanıtmak adına sembolik regresyon gibi popüler bir problemin değerlendiricisini hazır olarak bulundurmalıdır. Değerlendirici seçimini ise kodlama esnasından dilin anahtar sözcükleri ile yapabilir.

4.2.14 Değerlendirici ara yüzü içerme ve gelişime açık olma

Kullanıcıya hazır değerlendiriciler yeterli olmaz ise kendi değerlendiricisini belli bir ara yüz kullanarak yazabilmelidir.

4.3 Tasarım

Tasarım aşamasında analizde elde edilen çıktı kullanılır. Geliştirilecek olan dil analizde belirlenmiş ihtiyaçları karşılamalıdır. Tasarım aşamasında iki tane temel iş yapılması gerekir. Bunlardan biri dilin grameri ve çalışma anındaki davranışının tasarlanmasıdır. Dilin gramer yapısı doğrudan analizde ortaya çıkan ihtiyaçlara göre olmalıdır. Gramer dilin kullanılabilirliğini de doğrudan etkiler. Alana özgü bir dil olduğundan kullanıcıların dile kolayca adapte olmak isterler. Eğer anlaşılması ve kullanılması zor ise kullanıcılar genel amaçlı dillere kayabilir. Bir diğer iş ise dilin çalışma anındaki davranışının tasarlanmasıdır.

Tasarladığımız dil 2 ana bölümden oluşur. Bunlar konfigürasyon ve çalışma segmentleridir. Konfigürasyonda ağaç yapısı, seçme ve çaprazlama tekniği gibi kavramlar tanımlanırken. Çalışma segmentinde ise algoritmanın akışını sağlayan komutlar bulunur.

Bu bölümde tasarlanan bellek yönetimi ve komutlar anlatılacaktır.

4.3.1 Bellek yönetimi

Bellek yönetimi bir uygulama bağlamı üzerinden yapılması tasarlanmıştır. Uygulama bağlamında bütün değişken ve konfigürasyon ile ilgili değerler bulunmaktadır. Şekil 4.1 de uygulama bağlamının yapısı gösterilmiştir.

Uygulama bağlamı aşağıdaki yapılardan oluşur:

- Global Değişken
- Toplum
- Çocuk Listesi
- İsimli Değişkenler
- Seçim Stratejisi
- Değerlendirici
- Ağaç Üretme Stratejisi

4.3.1.1 Global değişken

Global değişken yürütülen en son komutun etkilediği değişkeni tutar. Genel amaçlı çoğu dil bütün komutlarda eğer bir değer saklanmak isteniyorsa kullanıcıyı isimli bir değişken kullanmaya zorlar. Ama genelde GA ve programlamada bütün blok işlemler tek bir değişken üzerinden yapılır. Global değişken kullanarak kullanıcılar hiçbir değişken tanımı yapmadan istediği komutları çalıştırabilir. Tasarlanan komutların çoğu hiçbir isimli değişken kullanılmadan çağırılabilir. Değişken kullanılmadan çağırıldığında komut direk global değişkeni parametre olarak kabul eder.

4.3.1.2 Toplum

Toplum GA 'larda çok temel bir kavramdır. Bellekte tüm algoritma boyunca tek bir toplum tutulması düşünülmüştür. Yürütülen toplum ile ilgili komutlar uygulama bağlamında bulunan toplum değişkenini güncelleyecektir. Bu komutlar topluma birey ekleme, toplumu değerlendirme ve en iyi bireyi bulma gibi komutlardır.

4.3.1.3 Çocuk listesi

Çocuk listesi yapısal olarak topluma çok bezerdir. Araştırmacı yeni çocuk bireyleri belli bir süre bir listede tutmak isteyebilir. Çocuk listesi daha sonradan topluma katılabilir. GA 'larda genelde bu işlem yeni nesil elde edildikten sonra olur.

4.3.1.4 İsimli değişkenler

Tasarlanan dilde tip tanımı yapmadan isimli değişkenlere de destek verilmesine karar verildi. Kullanıcı isterse global değişkeni kullanmayıp isimlendirdiği bir değişkeni de komutlara parametre olarak verebilir. İsimli değişkenler araştırmacının mutasyon, çaprazlama gibi işlemleri gerçekleştirirken kullanılabilir. İsimli değişkenler bir eşlem de tutulur. Bu eşlem uygulama bağlamında saklanır.

4.3.1.5 Seçim stratejisi

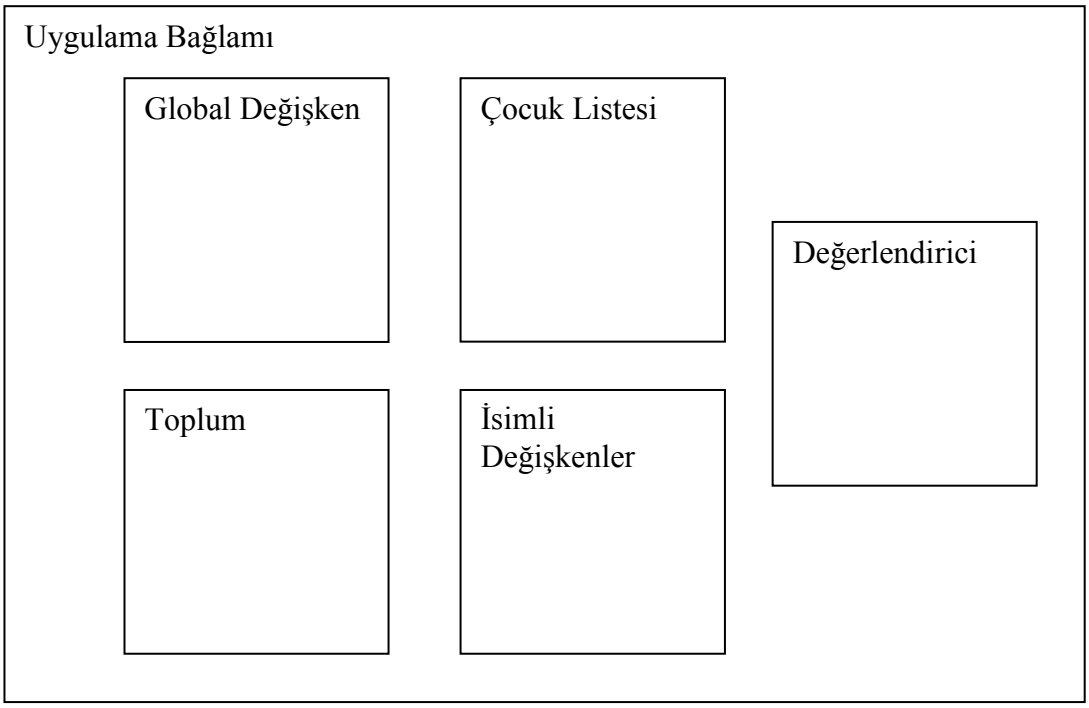
Rasgele birey ve birey içinde olan bir düğümü seçmek için çeşitli stratejiler kullanılabilir. Kullanıcı konfigürasyon segmetinde seçim stratejisini tanımlar. Seçim stratejisi uygulama bağlamında saklanır.

4.3.1.6 Değerlendirici

Değerlendirici probleme özgü olarak değişir. Kullanıcı var olan değerlendiricilerden seçimi konfigürasyon segmentinde yapar. Bu değer uygulama bağlamında saklanır.

4.3.1.7 Ağaç üretme stratejisi

Rasgele birey üretmek daha önceden de bahsedildiği gibi dilin bir özelliği olmalıdır. Rasgele birey üretirken üç adet yönteme destek verilmesi düşünülmüştür. Kullanıcı bu yöntemi konfigürasyon segmentinde belirleyebilir. Bunlardan bir tanesi "eşit derinlik". Bu yöntemde üretilen ağacın bütün yaprakları eşit derinlikte olacaktır. Bir diğer yöntem ise "maksimum derinlik" 'tir. Bu yöntemde yapraklar en fazla verilen bir değer derinliğinde olacaktır. Son yöntem ise ağacın diğer iki yöntemi yarı yarıya uygulamaktır. Ağacın yarısı maksimum derinliğe göre yarısı ise eşit derinliğe uygun olarak üretilecektir.



Şekil 4.1 : Uygulama Bağlamı

4.3.2 Konfigürasyon segmenti

Konfigürasyon segmenti kodun en tepesinde bulunur. İlk olarak kullanıcı konfigürasyon segmentini yazar. Konfigürasyon segmentinde kullanıcı bireylerin yapısını belirtmek üzere terminal ve operatör tanımı yapar. Kullanıcı terminal ve operatörleri isimlendir. Operatörlerin belli sayıda çocuğu olacağından çocuk sayısını da belirtir.

Konfigürasyon segmentini daha iyi anlatmak için bir sembolik regresyon problemi üzerinden gideceğiz.

Örneğimizde iki tane değişkenimiz x ve y, dört tane operatörümüz olsun çarpma, bölme, toplama ve çıkarma. Bu örnekte bütün operatörlerin iki adet çocuğu olabilir.

Örnek kod aşağıdadır. Terminaller "TERMINALS" anahtar sözcüğü ile tanımlanır. Operatörler ise "OPERATORS" anahtar sözcüğünün ardından tanımlanır. Operatörün ismin sonuna parantezler içinde çocuk sayısı yazılır.

```
TERMINALS = x y

OPERATORS = add(2) subtract(2) multiply(2) divide(2)
```

Operatör ve terminalleri tanımladıktan sonra değerlendirici seçme işlemi gelir. Değerlendirici probleme özgüdür. Bu örneğimizde hazır olan bir değerlendirici kullanılacaktır.

Örnek kod aşağıdadır:

```
EVALUATOR "SYMBOLIC"
```

Kullanıcı değerlendiriciyi seçtikten sonra ağaç üretme stratejisini seçer. Daha önceden de bahsedildiği gibi üç adet ağaç üretme yöntemi vardır. Bu örneğimizde bütün ağaçların aynı derinlikte olmasını istiyoruz.

Örnek kod:

```
TREEGENERATIONSTRATEGY EqualDepth(5)
```

Kullanıcı isterse çaprazlama yöntemini de direk seçebilir. Bu durumda "crossover" komutu kullanıldığında bu tekniğe göre çaprazlama yapılacaktır.

Örnek kod:

```
CROSSOVER ONE-POINT
```

Kullanıcı toplumdan birey seçim yöntemini de belirlemelidir. Bunlar "UNIFORM", "TOURNAMENT", "ROULETTE_WHEEL", "TRUNCATION" olabilir.

Örnek kod p = %30 olarak tournament selection yapmaktadır.

```
SELECTION TOURNAMENT(30)
```

4.3.3 Çalışma segmenti

Çalışma segmentinde kullanılabilecek komutlar aşağıda listelenmiştir.

- Sabitler
- Döngüler
- Olasılıklı blok çalıştırma
- Rasgele birey üretme
- Rasgele birey seçme
- Topluma ve çocuk listesine birey ekleme
- Birey Değerlendirme
- Birey kopyalama
- Toplumdan en iyi bireyi seçme
- Ağaç işlemleri

4.3.3.1 Sabitler

Sabitler tüm algoritma boyunca değişmeyecek değerleri tutarlar. Yürütme segmentinin en tepesinde bulunurlar. Döngü sayısını ve olasılıkları tanımlamak için kullanılabilir.

Örnek kod aşağıdadır:

```
a = 6

n = 2

k = 5
```

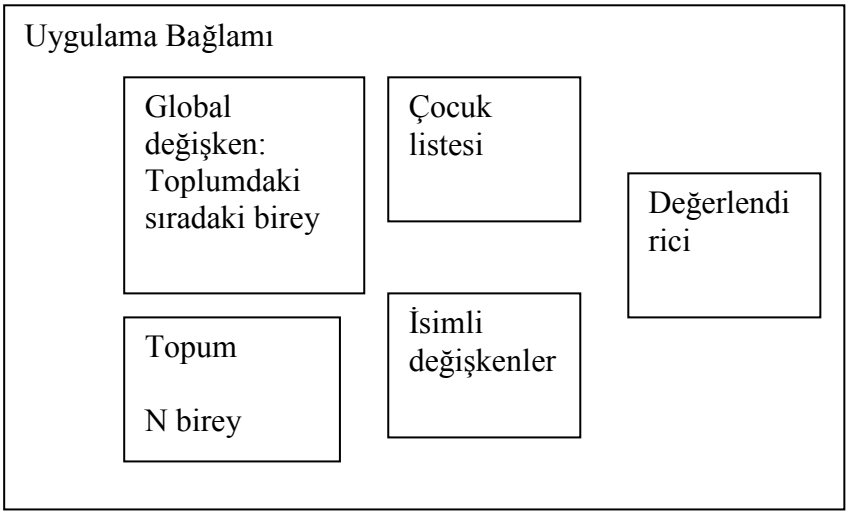
4.3.3.2 Döngüler

Programlama dilleri genelde döngülere destek verir. Geliştirmekte olduğumuz dil ile de döngü kurmak mümkündür. Dilde iki tip döngü vardır. Bunlar "foreach" ve "repeat" 'dir.

"foreach" döngüsü ile kullanıcı toplum içinde dolaşabilir. Döngüye girmeden önce toplumdan sırası ile bir birey alıp global değişkene konur. Şekil 4.2 de döngü bloğuna girmeden uygulama bağlamın yapısını önce görebilirsiniz.

Toplumda dolaşan döngünün örnek kodu aşağıdadır:

```
foreach individual in population {
        ...
        ...
}
```



Şekil 4.2 : foreach döngüsünün bloğuna girmeden önce uygulama bağlamı durumu
Uygulama Bağlamı

"repeat" bloğu ise basit bir şekilde kod bloğunu verilen parametre olarak verilen sayı kadar işletir.

Örnek kod aşağıdadır:

```
repeat a times {
     ...
```

```
    ...
}
```

4.3.3.3 Olasılıklı blok çalıştırma

Araştırmacılar belli blok işlemleri belli olasılıklara göre çalıştırmak isteyebilirler. Kullanıcı "with" anahtar sözcüğünü kullanarak daha önceden belirlediği bir olasılık oranı ile belli bir blok komutu çalıştırabilir.

Bununla ilgili örnek kod aşağıdadır:

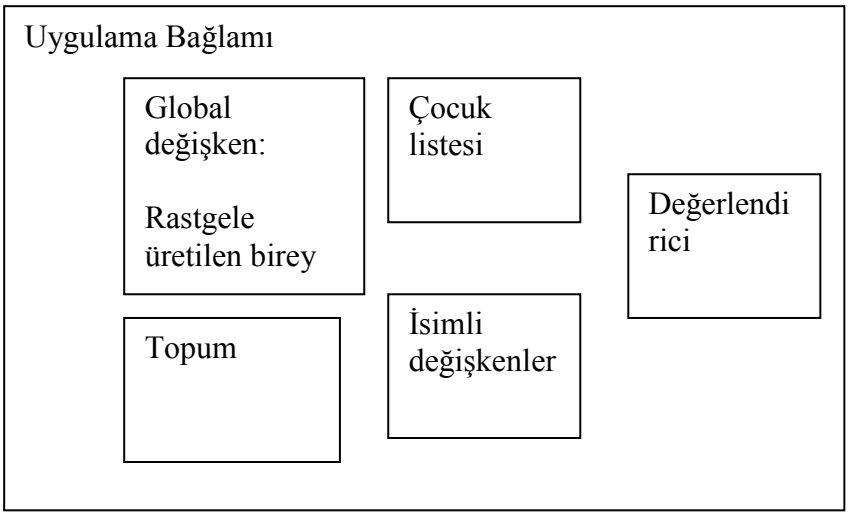
```
with pm {
    ...
    ...
}
```

4.3.3.4 Rasgele birey üretme

Kullanıcılar "random_individual" anahtar sözcüğünü kullanarak belirlediği kurallara göre rasgele birey üretebilirler. Bu komut parametresiz olarak işletildiğinde rasgele birey global değişkende bulunur. Kullanıcı isterse bunu ayrıca başka tanımladığı isimli değişkene de atayabilir. Şekil 4.3 de uygulama bağlamının komut çalıştırıldıktan sonraki durumunu görebilirsiniz.

Parametresiz kullanım:

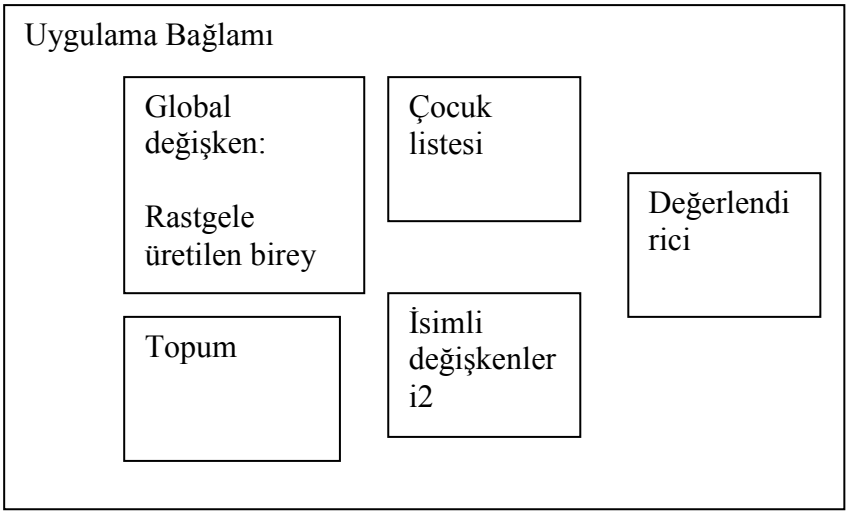
```
random_individual()
```



Şekil 4.3 : Uygulama Bağlamı

Kullanıcı daha önceden de bahsedildiği gibi isterse ürettiği bireyi başka isimli bir değişkene de atayabilir. Bu isimli değişkenlerde uygulama bağlamında saklanır. Ayrıca bu komut parametreli çalıştırılsa bile üretilen değişken birey global değişkene de atanacaktır. Şekil 4.4 de uygulama bağlamının komutun parametreli çalıştırıldıktan sonraki durumunu görebilirsiniz.

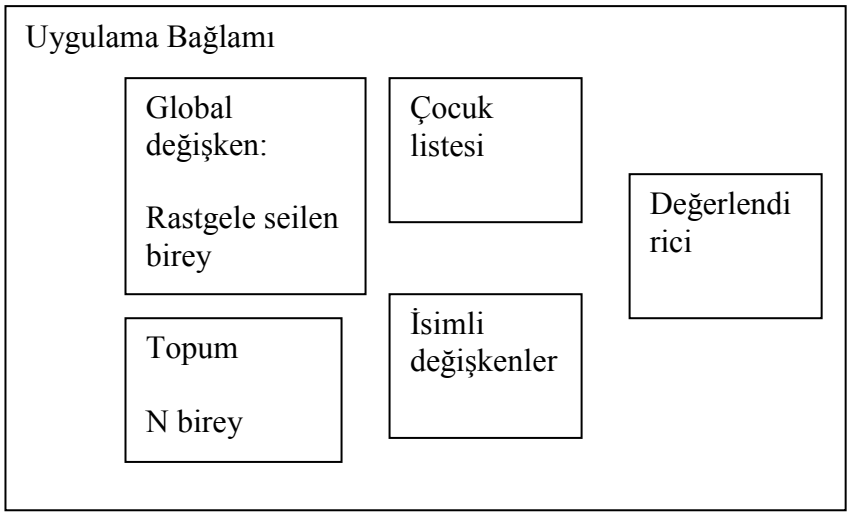
```
i2 = random_individual()
```



Şekil 4.4 : Uygulama bağlamı

4.3.3.5 Rasgele birey seçme

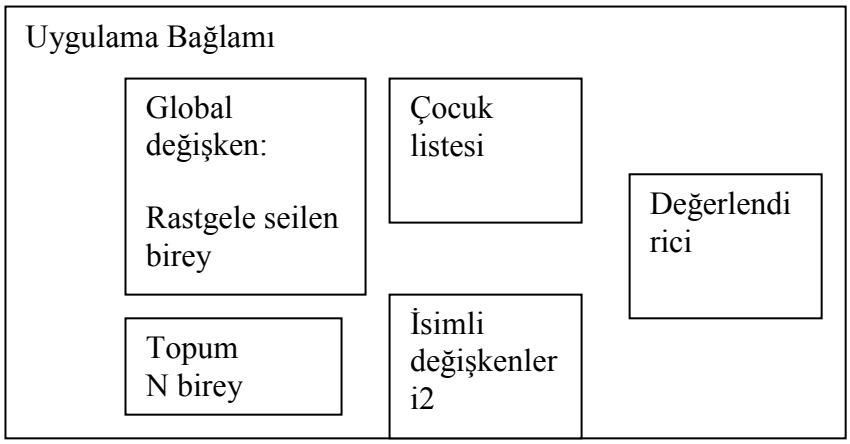
Kullanıcı “random_select” komutu ile toplumdan rasgele bir birey seçebilir. Komut çalıştırıldıktan sonra uygulama bağlamında saklanan toplumdan bir bireyi rasgele seçilip global değişkende saklanır. Şekil 4.5 de uygulama bağlamının komutun çalıştırıldıktan sonraki durumunu görebilirsiniz.



Şekil 4.5 : Uygulama bağlamı

Seçilen birey isimlendirilmiş bire değişkende de saklanabilir. Şekil 4.6 de uygulama bağlamının komutun parametreli çalıştırıldıktan sonraki durumunu görebilirsiniz.

```
i2 = random_select()
```



Şekil 4.6 : Uygulama bağlamı

4.3.3.6 Topluma birey ekleme

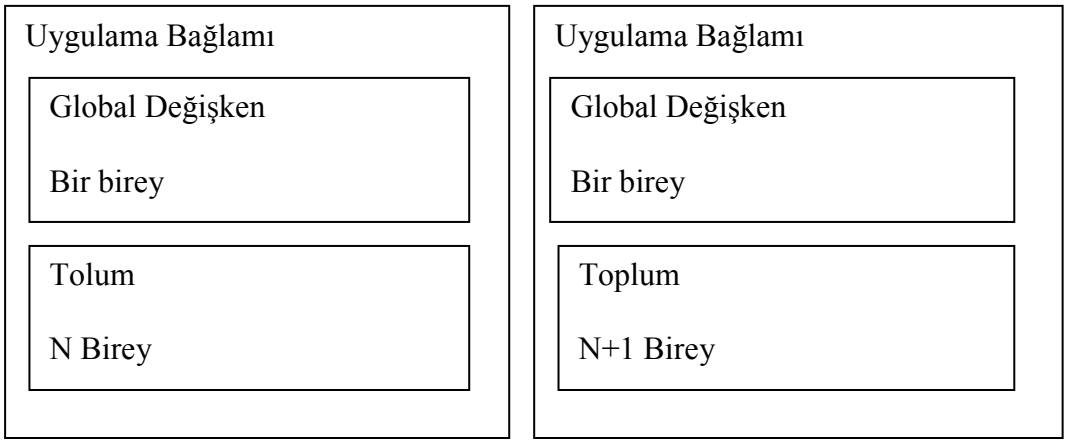
Kullanıcı "add_to_population" komutunu kullanarak topluma birey ekleyebilir. Parametresiz çalıştırıldığında global değişkendeki birey veya bireyleri topluma ekler. Global değişkende bu komut çalıştırmadan önce birey veya bireyler bulunması beklenir. Aksi durumda hata alınacaktır. Şekil 4.7 de uygulama bağlamının komutun çalıştırıldıktan önceki ve sonraki durumunu görebilirsiniz.

Aşağıdaki komut n bireyli rasgele bir toplum üretmektedir:

```
repeat n times {
     random_individual()
     add_to_population()
}
```

Komut istenirse parametre alabilir. Parametre olarak birey veya birey listesi beklenir:

```
add_to_population(i2)
```



Şekil 4.7 : Uygulama bağlamı (önce-sonra)

4.3.3.7 Çocuk listesine birey ekleme

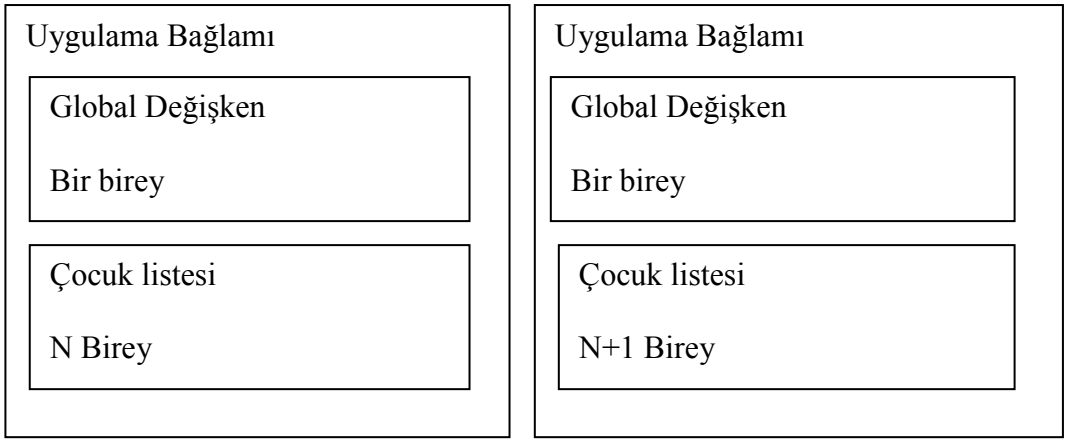
Kullanıcı "add_to_offspring" komutunu kullanarak çocuk listesine birey ekleyebilir. Çocuk listesi kullanıcıya yeni nesil yaratması için düşünülmüştür. Komut parametresiz çalıştırıldığında global değişkendeki bireyi çocuk listesine eklemektedir. Genelde çaprazlama ve mutasyonla işleminden üretilen çocuklar öncelikle bu listeye eklenir. Şekil 4.8 de uygulama bağlamının komutun çalıştırıldıktan önceki ve sonraki durumunu görebilirsiniz.

Aşağıdaki komut rasgele n adet birey üretip çocuk listesine eklemektedir.

```
repeat n times {
     random_select()
     add_to_offspring()
}
```

Komut parametre olarak birey veya birey listesi alabilir:

```
add_to_offspring(i2)
```



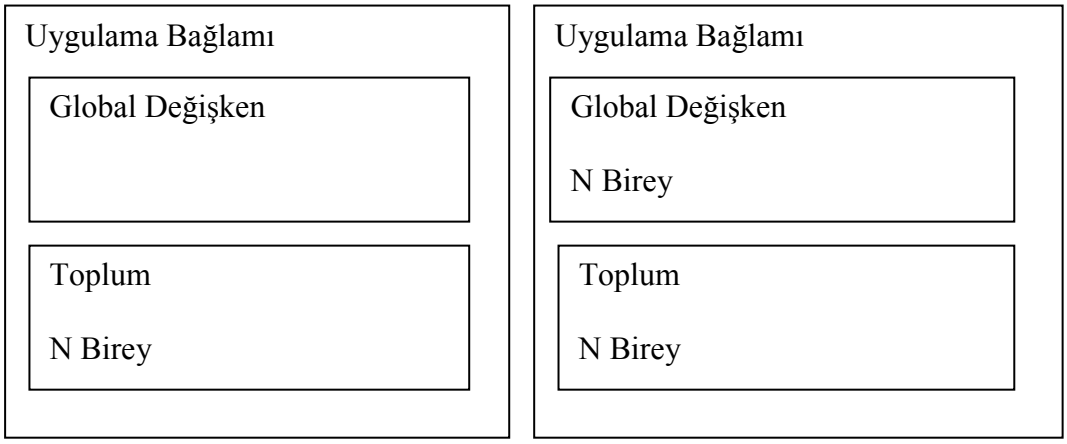
Şekil 4.8 : Uygulama bağlamı (önce-sonra)

4.3.3.8 Toplumu kullanma

Kullanıcı "use population" komutlu ile uygulama bağlamındaki toplumu global değişkene atayabilir. Bunu diğer komutları toplum üzerinde kullanmak için yapabilir. Şekil 4.9 de uygulama bağlamının komutun çalıştırıldıktan önceki ve sonraki durumunu görebilirsiniz.

Örnek kod aşağıdadır:

```
use_population()
```



Şekil 4.9 : Uygulama bağlamı (önce-sonra)

4.3.3.9 Toplum küçültme

Kullanıcı bireylerin başarım değerlerine göre toplumun sayısını azaltmak isteyebilir. Bu durumda "reduce_population" komutunu kullanarak bu işemi gerçekleştirebilir. Aşağıdaki komut yürütüldükten sonra toplumda en iyi n birey kalmış olacaktır:

```
reduce_population(n)
```

4.3.3.10 Birey değerlendirme

Kullanıcı "evaluate" komutunu bireyi değerlendirmek için kullanabilir. Parametresiz çalıştırması global değişkenin değerlendirilmesini sağlayacaktır. Ancak burada global değişkenin birey tipinden olması garanti edilmelidir. Aksi durumda hata alınacaktır.

Aşağıdaki komut toplumdaki bütün bireyleri değerlendirir:

```
foreach individual in population {
      evaluate()
}
```

Komut parametre olarak isimlendirilmiş değişken de alabilir. Komut işletildikten sonra değişken değerlendirilecek ve de aynı zaman da global değişkene de atanacaktır.

Örnek kod aşağıdadır:

```
evaluate(i1)
```

4.3.3.11 Birey kopyalama

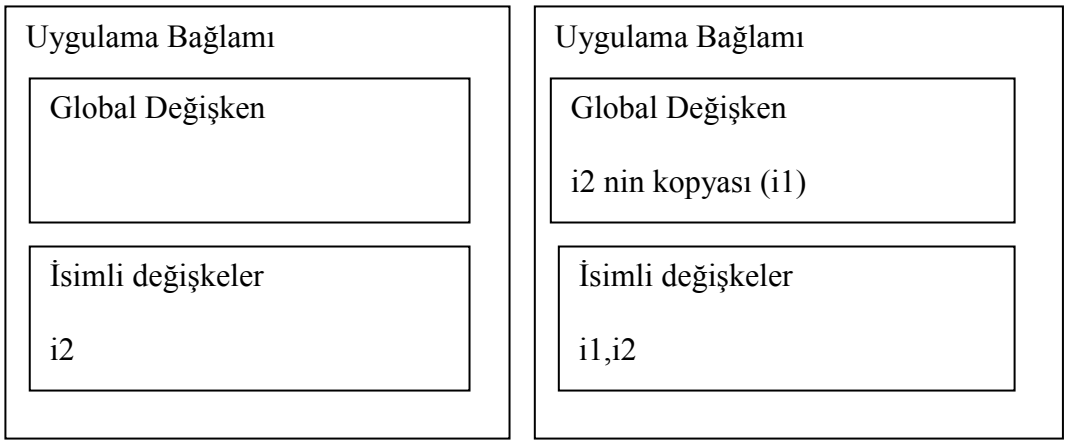
Dildeki bireylere ulaşım referanslar üzerinden gerçekleştiğinden kullanıcılar bireylerin kopyalılarını elde zorunda kalabilir. Örneğin çaprazlama yapmadan önce orijinal birey elde tutulmak isteniyorsa çaprazlama yapılacak bireylerin birer kopyası alınmalıdır. Bu işlemi "clone" komutu ile gerçekleştirebilirler.

Aşağıdaki kod toplumdan birey seçip global değişkene atar ardından bir kopyasını alıp tekrar global değişkene atar. Böylelikle yapılan değişikliklerden toplumdaki birey etkilenmemiş olur:

```
random_select()

clone()
```

Komut parametre olarak isimli değişken de alabilir. Örnek komut i2 değişkenin kopyasını i1 değişkenine atar. Şekil 4.10 de uygulama bağlamının komutun çalıştırıldıktan önceki ve sonraki durumunu görebilirsiniz.

```
i1 = clone(i2)
```



Şekil 4.10 : Uygulama bağlamı (önce-sonra)

4.3.3.12 Toplumdan en iyi bireyi seçme

Kullanıcılar "find_best" komutu ile toplumdaki başarım değeri hesaplanmış en iyi bireyi seçebilirler. Komut çalıştırıldıktan sonra en iyi birey global değişkene atanacaktır.

Örnek kod aşağıdadır:

find_best()

4.3.3.13 Ağaç işlemleri

Bireylerin temsilin ağaç şeklinde olduğundan kullanıcılar temel ağaç operasyonlarını da rahatlıkla gerçekleyebilmelidir.

4.3.3.13.1 Ağaçtan rasgele düğüm seçme

Kullanıcı mutasyon ve çaprazlama işlemlerini gerçekleyebilmek için ağaçtan rasgele bir düğüm seçmelidir. "random_position" komutunu kullanarak ağaçtan bir düğüm seçip onun global değişkene veya isimli bir değişkene atılmasını sağlayabilir.

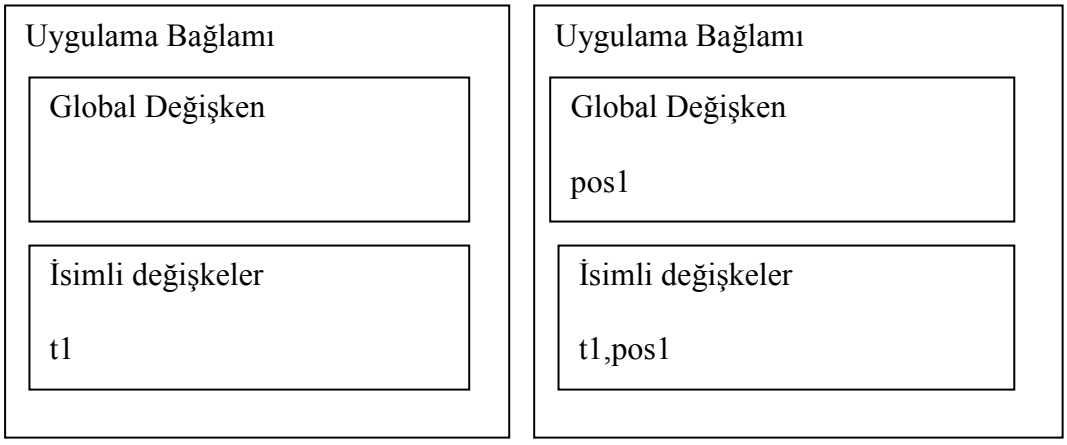
Aşağıdaki kod global değişkendeki ağaçtan bir düğüm seçip global değişkene atamaktadır:

```
random_position()
```

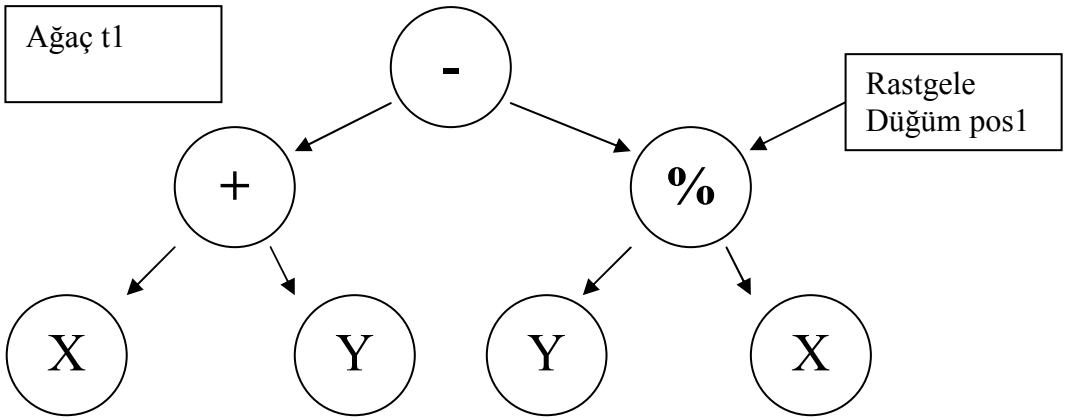
Kullanıcı isterse isimli değişken olarak tutulan bir ağaçtan isimli bir değişkene de rasgele bir düğüm atayabilir.

```
pos1 = random_position(t1)
```

Şekil 4.11 ve şekil 4.12 de uygulama bağlamının ve ağacın komutun çalıştırıldıktan önceki ve sonraki durumunu görebilirsiniz.



Şekil 4.11 : Uygulama bağlamı (önce-sonra)



Şekil 4.12 : Örnek Ağaç

4.3.3.13.2 Ağaç budama

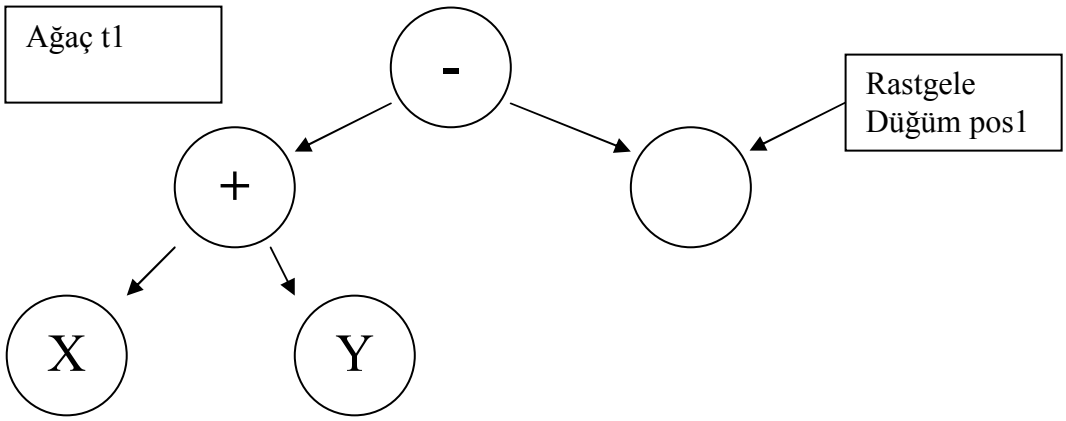
"prune" komutu ile kullanıcı bir ağacı belli bir düğümden budayabilir. Komut parametresiz çalıştırıldığında global değişkende düğümden itibaren budama işlemi yapılır.

Örnek kod aşağıdadır:

```
prune()
```

Kullanıcı isimlendirilmiş değişkende tutulan belli bir düğümden de budama işlemi yapabilir. Şekil 4.13 de budanmış ağacı görebilirsiniz.

```
prune(pos1)
```



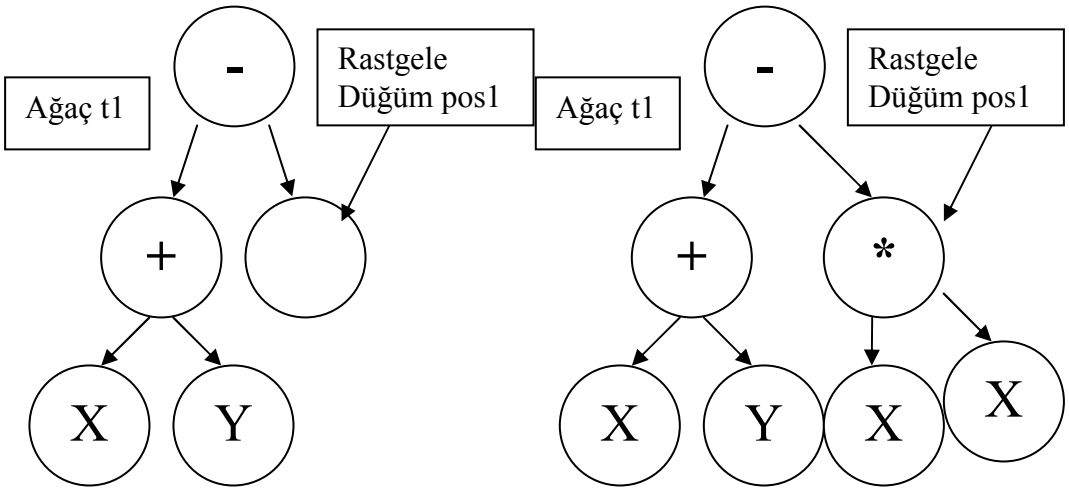
Şekil 4.13 : Budanmış Ağaç

4.3.3.13.3 Rasgele ağaç büyütmek

"grow" komutu ile kullanıcı bir ağacı belli bir düğümden rasgele kurallara göre büyütebilir. Komut parametresiz çalıştırıldığında global değişkende düğümden itibaren büyütme işlemi yapılır. Bu işlem mutasyon amaçlı kullanılabilir. Şekil 4.14 de büyütme işlemi gösterilmiştir.

Örnek kod:

```
grow(pos1)
```



Şekil 4.14 : Büyütme işlemi

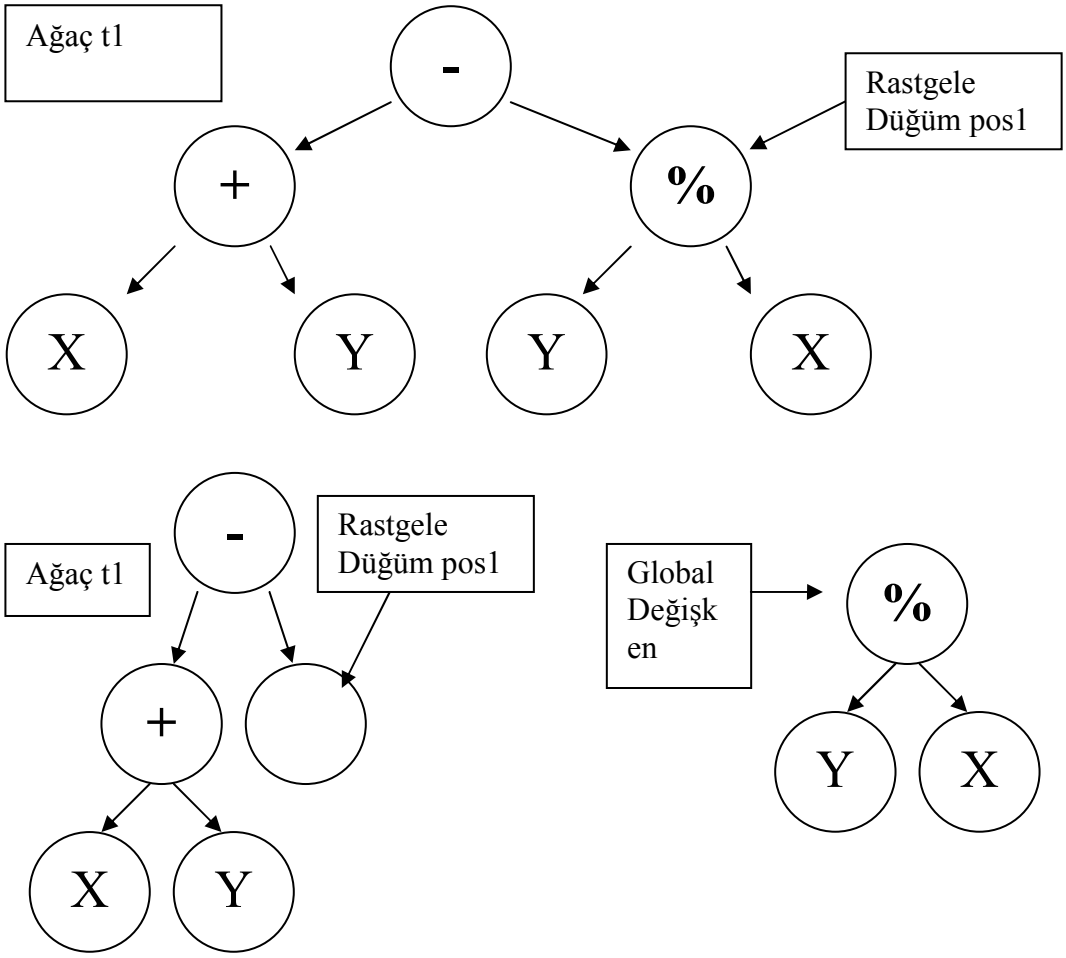
Örnek olarak aşağıdaki kod toplumdan bir birey seçip, kopyasını alıp, bir düğümden itibaren budayıp, tekrar büyütmektedir. Bu işlem tipik bir mutasyon işlemidir.

```
random_select()

clone()

random_position()

prune()

grow()
```

4.3.3.13.4 Ağaçtan alt ağaç koparma

"detach" komutu ile kullanıcı bir ağacı belli bir düğümden bir alt ağaç koparabilir. Komut parametresiz çalıştırıldığında global değişkende düğümden itibaren koparma işlemi yapılır. Koparılan ağaç tekrar global değişkene atanır. Bu işlem çaprazlama amaçlı kullanılabilir. Şekil 4.15 de koparma işlemi gösterilmiştir. Örnek kod:

```
detach(pos1)
```



Şekil 4.15 : Koparma İşlemi

Kopartılan ağaç istenirse bir isimli değişkene de atanabilir:

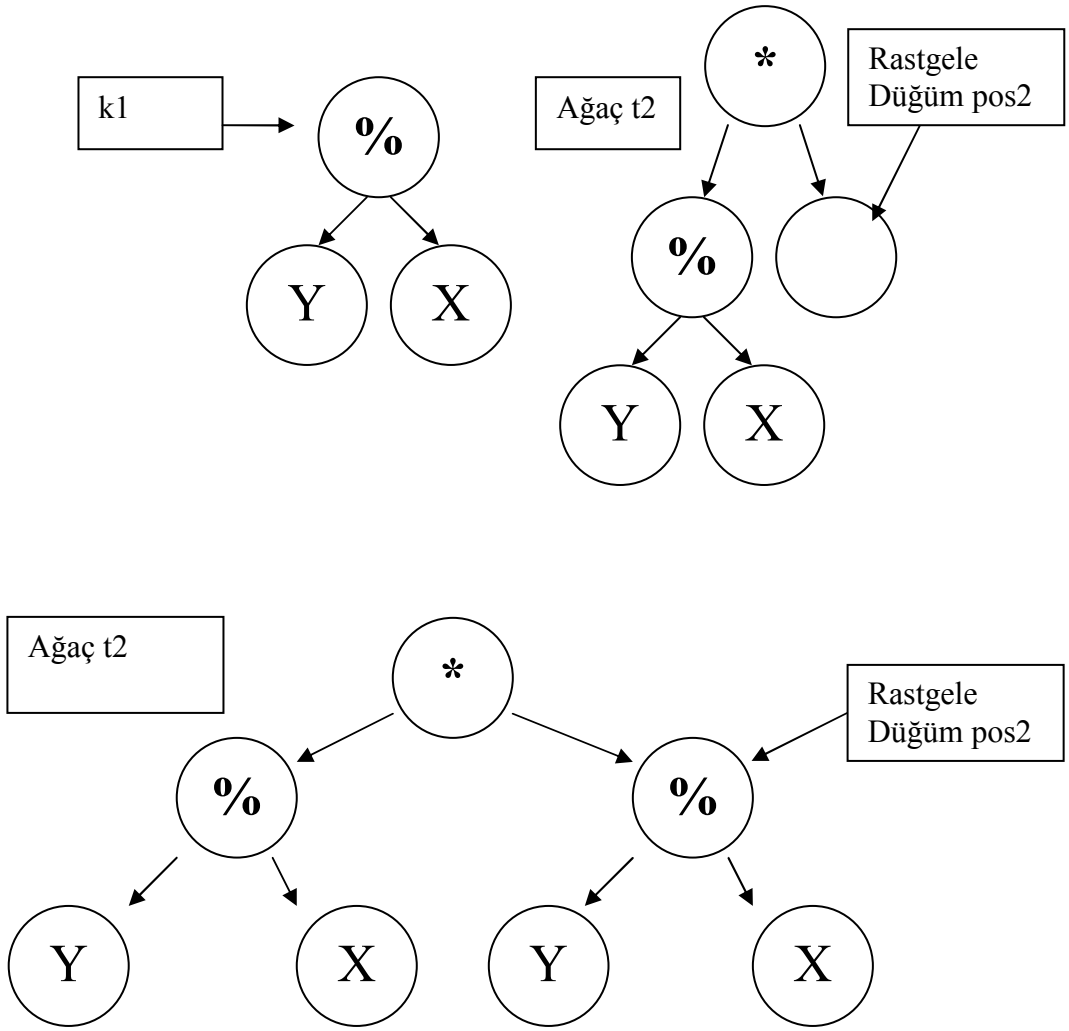
```
k1 = detach(pos1)
```

4.3.3.13.5 Ağaca alt ağaç ekleme

"attach" komutu ile kullanıcı bir ağacı belli bir düğümden bir alt ağacı bir ağaca ekleyebilir. Komut iki adet parametre alır. Bunlardan bir tanesi ekleme yapılacak düğüm, diğeri ise alt ağaçtır. Şekil 4.16 de k1 alt ağacının pos2 düğümünden eklenmesi gösterilmiştir.

Daha iyi anlaşılması açısından örnek vermek gerekirse aşağıdaki kod başız tavuk çaprazlaması yapmaktadır.

```
with pc {
  random_select()
  i1 = clone()
  i2 = random_individual()
  pos1 = random_position(i1)
  pos2 = random_position(i2)
  k1 = detach(pos1)
  k2 = detach(pos2)
  attach(pos1 k2)
  attach(pos2 k1)
  evaluate(i1)
  evaluate(i2)
  add_to_offspring(i1)
  add_to_offspring(i2)
}
```



Şekil 4.16 : Ekleme İşlemi

4.3.3.14 Örnek sembolik regresyon problemi

Dili daha anlatmak için örnek bir sembolik regresyon problemini tasarlanan dil ile gerçeklenecektir.

İlk olarak, problem için terminal ve operatör tanımlama işlemi yapılır. Bu işlem problem özgü bir işlemdir. Problemimiz de örnek olarak dört adet operatör olsun. Bunlar bölme, çarpma, toplama ve çıkarmadır. Bu operatörlerin hepsinin iki adet giriş değişkeni vardır.

Tanım aşağıdaki gibi yapılabilir:

```
OPERATORS = plus(2) minus(2) devide(2) multiply(2)
```

Operatör tanımı yapıldıktan sonra terminal tanımı yapılır. Problemimizde iki adet değişken olsun. Bunlar x ve y'dir. Tanım aşağıdaki gibi yapılabilir:

```
TERMINALS = x y
```

Operatör ve terminal tanımlamasından sonra değerlendiriciyi tanımlayabiliriz. Bu problemde sembolik regresyon için hazır yazılmış olan bir değerlendirici kullanılacaktır. Sembolik regresyonda bilindiği üzere belirli girdilere belirli çıktılar üreten bir denklem bulunmaya çalışılır. Değerlendirici için kullanıcı bir dizi girdi ve bunlara karşılık düşüne çıktılar vermek durumundadır.

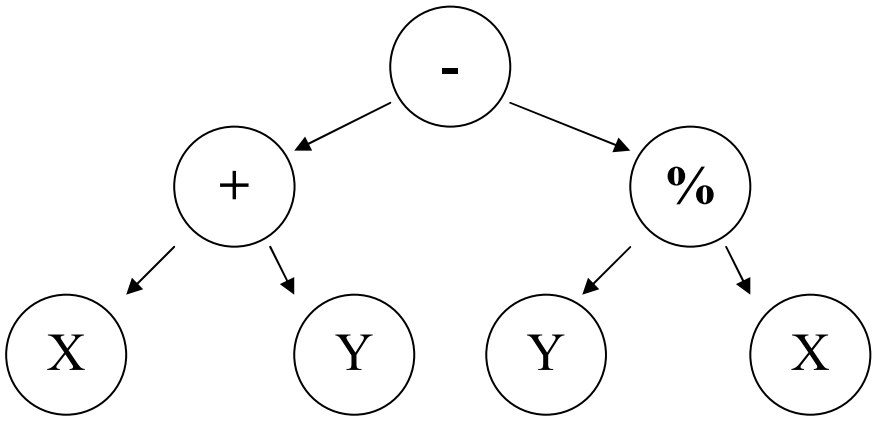
Değerlendirici aşağıdaki gibi tanımlanır:

```
EVALUATOR "SYMBOLIC"
```

Değerlendirici tanımlanmasından sonra ağaç üretme stratejisi belirlenir. Daha önceden de bahsedildiği gibi üç çeşit ağaç üretme stratejisine destek verilmektedir. Bunlar eşit, en fazla ve yarı yarıyadır. Biz bu problemde eşit stratejisini seçelim ve derinlik üç olsun.

```
TREEGENERATIONSTRATEGY EqualDepth(3)
```

Şekil 4.17 de örnek olarak bu tanımlamalara uygun olan bir örnek ağaç görebilirsiniz.



Şekil 4.17 : Örnek Ağaç

Son işlemle konfigürasyon segmentinin gerçeklenmesini tamamlamış olduk. Çalışma segmentine geçebiliriz.

Algoritma gereği İlk rasgele bir toplum üretmemiz gerekmektedir. Bunun için "repeat" döngüsü, "random_individual" ve "add_to_population" komutları kullanılacaktır. Toplum 100 adet bireyden oluşsun. 100 değerini sabit olarak tanımlayalım. Ardından bir "repeat" döngüsü içinde bireyleri "random_individual" ile üretip, "add_to_population" komutu ile topluma ekleyelim:

```
n = 100
repeat n times {
    random_individual()
    add_to_population()
}
```

Toplumu ürettikten sonra toplum bireylerini değerlendirelim. Yukarıdaki döngüde topluma bireyleri eklemeden öncede bu işlem yapılabilirdir. Fakat okunabilirliği arttırmak için burada "ForEach" döngüsü kullanılacaktır. Toplumdaki bütün bireyler değerlendirilir:

```
foreach individual in population {
     evaluate()
}
```

Algoritma gereği belli bir sayıda çocuk üretip, topluma eklenir. Daha sonra toplum nüfusu belli bir değere azaltılır. Bu işlem için nesil döngüsü kurulur. Bunu için "repeat" komutu kullanılacaktır. 100 kere bu işlemin yapılmasını isteyelim:

```
k = 100
repeat k times {
    ...
    ...
}
```

Bu ana döngüde çocuk üretip daha sonra topluma katılacağından bunun da bir iç döngü kullanılacaktır. İç döngü bitiminde çocuklar topluma katılacak ardından da toplum nüfusu azaltılacaktır. Toplum nüfusunu azaltmak için "reduce_population" komutu kullanılacaktır. Bu komut en iyi n bireyi seçip bunları toplumda tutarken diğerlerini toplumdan çıkarmaktadır. Kod aşağıdaki gibidir:

```
m = 100
repeat k times {
    repeat m times {
    ...
    ...
          }
          offspring_to_population()
          reduce_population(n)
}
```

İç döngüde çocuk üretmek için gerekli komutlar bulunmalıdır. Çocuklar çaprazlama ve mutasyon işlemleri ile üretilecektir. Örneğimizde iki tip çaprazlama yöntemi gerçeklenecektir. Bunlardan bir tanesi standart iki noktadan çaprazlama olup diğeri ise kesik başlı tavuk çaprazlamasıdır.

Standart iki noktadan çaprazlama için iki adet birey toplumdan rasgele seçilir. Daha sonradan bireylerden rasgele iki adet düğüm seçilir. Seçilen noktalardan alt ağaçları ile birlikte yer değiştirme işlemi yapılır. İlk olarak "random_select" komutu ile toplumdan bir birey seçelim ve global değişkene atanmasını sağlayalım.

```
random_select()
```

Nesneler referans gibi davrandığından çaprazlama işleminden önce bir kopyasını başka bir değişkene alalım.

```
i1 = clone()
```

Toplumdan aynı şekilde bir birey daha seçip bunu da belli bir değişkene atayalım.

```
random_select()
i2 = clone()
```

Çaprazlama için bireyden rasgele bir düğüm seçelim. Bunun için "random_position" komutu kullanılacaktır:

Rasgele düğüm pos1 değişkenine atanacaktır:

```
pos1 = random_position(i1)
```

Diğer ağaçtan alınan düğüm ise pos2 değişkenine atanacaktır:

```
pos2 = random_position(i2)
```

Seçtiğimiz düğümlerden değiştirme işlemi yapmak için alt ağaçları “detach” komutu ile koparıp k1 ve k2 değişkenlerine atayalım:

```
k1 = detach(pos1)

k2 = detach(pos2)
```

Çaprazlama yapmak için alt ağaçların yerlerini değiştirelim. Bunun için alt ağacı belli bir düğüm yerinden ağaca bağlayan ”attach” komutu kullanılacaktır. Burada k1 alt ağacını pos2’den, k2 alt ağacını ise pos1’den bağlamalıyız.

```
attach(pos1 k2)

attach(pos2 k1)
```

Daha sonradan yeni üretilen bireyleri değerlendirip çocuk listesine atalım:

```
evaluate(i1)

evaluate(i2)

add_to_offspring(i1)

add_to_offspring(i2)
```

Algoritmada bu çaprazlama yöntemini belli bir olasılıkla gerçekleştirmek isteniyorsa ”with” komutunu kullanarak bu işlemleri olasılıklı gerçekleyebiliriz. 50 % olasılıkla standart çaprazlama yapmak isteyelim:

```
psc = 50

with psc {

     ...

     ...

}
```

Standart çaprazlama kodunu bitirmiş olduk:

```
psc = 50

with pm{

     random_select()
```

```
    i1 = clone()
    random_select()
    i2 = clone()
    pos1 = random_position(i1)
    pos2 = random_position(i2)
    k1 = detach(pos1)
    k2 = detach(pos2)
    attach(pos1 k2)
    attach(pos2 k1
    evaluate(i1)
    evaluate(i2)
    add_to_offspring(i1)
    add_to_offspring(i2)
    }
```

Kesik başlı tavuk çaprazlamasında ilk birey toplumdan seçilir. İkinci birey ise rasgele üretilir. İki koddaki temel fark bu seçim işlemidir:

```
random_select()
i1 = clone()
i2 = random_individual()
```

Örneğimizde bu yöntem %10 ihtimalle uygulansın:

```
ph = 10
with ph {
    random_select()
    i1 = clone()
    i2 = random_individual()
    pos1 = random_position(i1)
    pos2 = random_position(i2)
```

```
        k1 = detach(pos1)
        k2 = detach(pos2)
        attach(pos1 k2)
        attach(pos2 k1
        evaluate(i1)
        evaluate(i2)
        add_to_offspring(i1)
        add_to_offspring(i2)
}
```

Dilimizle iki tip çaprazlama yöntemini gerçeklemiş olduk. Şimdi ise mutasyon işlemini gerçekleyebiliriz. Bizim öğreğimizde toplumdan bir birey seçilip rasgele olarak mutasyon işlemi gerçeklenecek ve ardından çocuk listesine katılacaktır.

Örneğimizde mutasyon işlem bloğu çaprazlama bloklarından daha sonra gelmektedir.

İlk olarak toplumdan bir birey seçelim:

```
random_select()
```

Rasgele birey global değişkendedir. Bireyin kendisi değiştirmemek için bir kopyasını alalım:

```
clone()
```

Artık rasgele seçtiğimiz bireyin kopyası global değişkende bulunmakta. Global değişkenden rasgele bir düğüm seçelim:

```
random_position()
```

Rasgele düğüm global değişkende bulunuyor. Bu düğümden itibaren ağacı budayalım:

```
prune()
```

Ağaç budandı ve boş olan düğüm şuanda global değişkende. Bu noktadan itibaren ağacı rasgele büyütürsek mutasyon işlemini yapmış olacağız:

```
grow()
```

Son olarak yeni bireyi değerlendirip çocuk listesine koyalım:

```
evaluate()
add_to_offspring()
```

Mutasyon işleminin %10 olasılıkla olmasını istiyorsak:

```
pm = 10
with pm {
     random_select()
     clone()
     random_position()
     prune()
     grow()
     evaluate()
     add_to_offspring()
}
```

Çaprazlama ve mutasyon işlemlerini gerçekledik. GP ile sembolik regresyonu tamamlamış olduk:

```
OPERATORS = plus(2) minus(2) devide(2) multiply(2)
TERMINALS = x y
EVALUATOR "SYMBOLIC"
TREEGENERATIONSTRATEGY EqualDepth(3)
n = 100
m = 100
k = 100
psc = 50
ph = 10
pm = 10
repeat n times {
    random_individual()
```

```
    add_to_population()
}
repeat k times {
    repeat m times {
       with psc {
          random_select()
          i1 = clone()
          random_select()
          i2 = clone()
          pos1 = random_position(i1)
          pos2 = random_position(i2)
          k1 = detach(pos1)
          k2 = detach(pos2)
          attach(pos1 k2)
          attach(pos2 k1
          evaluate(i1)
          evaluate(i2)
          add_to_offspring(i1)
          add_to_offspring(i2)
}
     with ph {
         random_select()
         i1 = clone()
         i2 = random_individual()
         pos1 = random_position(i1)
         pos2 = random_position(i2)
         k1 = detach(pos1)
```

```
        k2 = detach(pos2)
        attach(pos1 k2)
        attach(pos2 k1
        evaluate(i1)
        evaluate(i2)
        add_to_offspring(i1)
        add_to_offspring(i2)
    }
    with pm {
        random_select()
        clone()
        random_position()
        prune()
        grow()
        evaluate()
        add_to_offspring()
    }
}
    offspring_to_population()
    reduce_population(n)
}
```

4.4 Gerçekleme

Gerçekleme safhasında, Xtext geliştirme aracı olarak seçildi ve kullanıldı. Xtext gramer dili ile geliştirmek istenen dilin grameri tanımlanabilir.

4.4.1 Xtext gramer dili

Xtext gramer dilini açıklamak için basit bir örnek olarak verilecektir. Grameri belirten dosya Xtext projelerinde ".xtext" uzantılıdır [5].

Aşağıdaki örneği incelediğimizde:

```
Model:
  (types+=Type)*;              (1)
Type:
  DataType | Entity;           (2)
DataType:
  "datatype" name=ID;          (3)
Entity:
  "entity" name=ID "{"
    (features+=Feature)*       (4)
  "}";
Feature:
  type=[Type|ID] name=ID;      (5)
```

(1) Model nesnesi bir veya birden fazla Type nesnesinden oluşmaktadır.

(2) Type nesnesi DataType veya Entity olabilir.

(3) DataType "datatype" anahtar sözcüğü ile başlar ve bir isimle devam eder

(4) Entity nesnesi "entity" anahtar sözcüğü ile başlar ve isimle devam eder. Ardından süslü parantez ile bir bloktan oluşur. Blok Features nesnelerinden oluşur.

(5) Feature bir Type ve isimden oluşur.

Aşağıdaki kod bu gramere uygun bir örnektir:

```
datatype String
entity Person {
    String name
    String lastName
```

Address home

Address business

}

entity Address {

String street

String zip

String city

}

4.4.2 Xtext gramer dili ile tasarlanan dilin tanımlanması

Geliştirdiğimiz dil iki ana bölümden oluşur. Bunlar konfigürasyon ve yürütme segmentleridir. Kullanıcı kodlamaya konfigürasyon segmentinden başlamalıdır. Xtext gramer dili ile ana yapısı aşağıdaki gibi tanımlanmıştır:

```
Model:
(evaluator=Evaluator)?                                  (1)
(terminalDecleration=TerminalDecleration)?              (2)
(operatorDecleration=OperatorDecleration)?              (3)
(treeGenerationStrategy=TreeGenerationStrategy)?        (4)
(constants+=Constant)*                                  (5)
(statements+=Statement)+;                               (6)
```

Model nesnesi aşağıdaki nesnelerden oluşur:

(1) Değerlendirici tanımlama: Değerlendirici "EVALUATOR" anahtar sözcüğü ve bir katardan oluşur.

```
Evaluator:
'EVALUATOR' evaluator=STRING;
```

(2) Terminal tanımlama: 1..n Terminal içerir. "TERMINALS" anahtar sözcüğü ile başlayıp eşittir ile terminal isimlendirme işlemi yapılır.

```
TerminalDecleration:
```

```
'TERMINALS' '=' (terminals+=Terminal)*;  (2.1)
```

(2.1) Terminal Tanımlama: Terminallerin bir adet ismi olur.

```
Terminal:
name=ID;
```

(3) Operatör Tanımlama: 1..n Operatörden oluşur

```
OperatorDecleration:

'OPERATORS' '=' (operators+=Operator)*; (3.1)
```

(3.1) Operatör Tanımlama: Operatörün bir ismi ve parantez içinde çocuk sayısı olur.

```
Operator:
name=ID '(' childCount=INT ')';
```

(4) Ağaç üretme stratejisi: “TREEGENERATIONSTRATEGY” anahtar sözcüğü ile başlar ve “EqualDepth”, “AtMost”, “FiftyFifty” anahtar sözcüklerinden birini alır.

```
TreeGenerationStrategy:
'TREEGENERATIONSTRATEGY' type=('EqualDepth' | 'AtMost' |
'FiftyFifty') '(' childCount=INT ')';
```

(5) 0..n sabitler: Sabit adı ve bir tamsayı değeri alır.

```
Constant:
name=ID '=' value=INT ;
```

(6) 1..n yürütme komutları: Bir veya birden çok yürütme komutu içerir.

```
Statement:
   With | FindBest | Evaluate | Clone |
   Repeat | ForEach | RandomSelect | RandomIndividual |
   AddToPopulation | UsePopulation | AddToOffspring |
   Detach | Attach | RandomPosition | Prune | Grow |
OffspringToPopulation | ReducePopulation;
```

(6.1) With: “with” anahtar sözcüğü ile başlar bir sabitle devam eder ardından süslü parantezler içinde bir veya birden çok komut içerir.

```
With:
     command='with' n=[Constant] '{'
     (statements+=Statement)*
     '}';
```

(6.2) FindBest: “find_best” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
FindBest:
     (var=ID  '=')? command='find_best''('(arg+=ID)* ')
```

(6.3) Evaluate: “evaluate” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
Evaluate:
     (var=ID  '=')? command='evaluate''(' (arg+=ID)* ')';
```

(6.4) Clone: “clone” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
Clone:
     (var=ID  '=')? command='clone''(' (arg+=ID)* ')';
```

(6.5) Repeat: “repeat” anahtar sözcüğü ile başlar. Ardından bir sabit ve “times” anahtar sözcüğü ile bir bloktan oluşur. Blok bir veya birden fazla yürütme komutundan oluşur.

```
 Repeat:

   command='repeat' n=[Constant] ' times {'
      (statements+=Statement)*
   '}';
```

(6.6) ForEach: “foreach individual in population” anahtar sözcükleri ile başlar. Komutları içeren bir blokla devam eder.

```
ForEach:
     command='foreach' 'individual in population {'
     (statements+=Statement)*
     '}';
```

(6.7) RandomIndividual: “random_individual” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
RandomIndividual:
     (var=I '=')?
     command='random_individual''('(arg+=ID)* ')';
```

(6.8) AddToPopulation: “add_to_population” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir. AddToPopulation:

(var=ID '=')? command='add_to_population"(' (arg+=ID)* ')';

(6.9) UsePopulation: “use_population” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
UsePopulation:
     (var=ID          '=')?     command=’use_population''('
     (arg+=ID)* ')';
```

(6.10) AddToOffspring: “add_to_offspring” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
AddToOffspring:
     (var=ID         '=')?    command=’add_to_offspring''('
     (arg+=ID)* ')';
```

(6.11) Detach: “detach” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
Detach:
     (var=ID  '=')? command=’detach''(' (arg+=ID)* ')';
```

(6.12) Attach: “attach” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
Attach:
     (var=ID  '=')? command=’attach''(' (arg+=ID)* ')';
```

(6.13) Prune: “prune” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
Prune:
     (var=ID  '=')? command=’prune''(' (arg+=ID)* ')';
```

(6.14) Grow: “grow” anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
Grow:
```

```
    (var=ID  '=')? command='grow''(' (arg+=ID)* ')';
```

(6.15) OffspringToPopulation: "offspring_to_population" anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
OffspringToPopulation:
    (var=ID '=')? command=' offspring_to_population''('
    (arg+=ID)* ')';
```

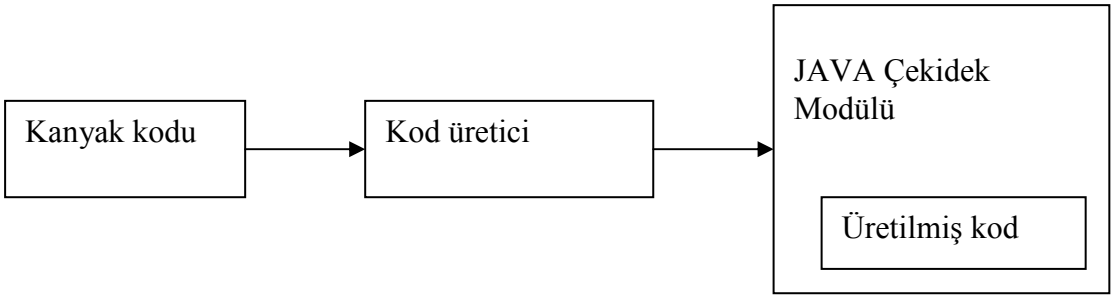
(6.16) ReducePopulation: "reduce_population" anahtar sözcüğü ile başlar. Parantez içinde 0..n tane argüman kabul eder. Sol tarafında bir değişken alabilir.

```
ReducePopulation:
    (var=ID     '=')?     command='reduce_population''('
    (arg+=ID)* ')';
```

4.4.3 Xtext ile kod üretme

Geliştirdiğimiz dilden direk bir çalıştırılabilir dosya üretmek yerine Java kodu üretmeyi tercih ettik. Bunun için Xtext kod üretme aracını kullandık.

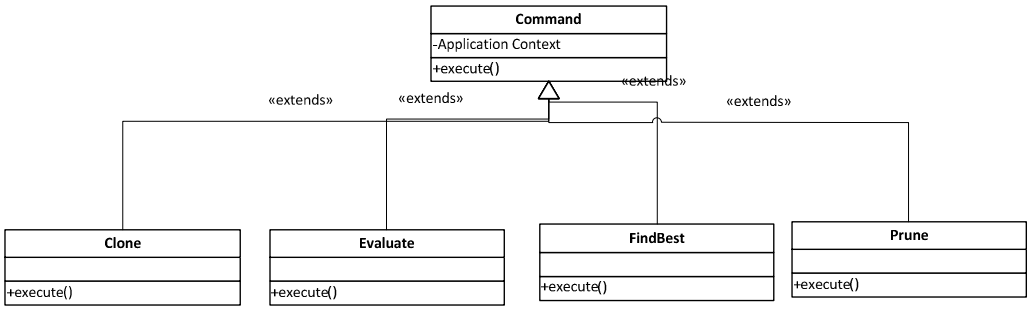
Bütün Java kaynak kodlarını tekrardan üretmek gereksiz olduğundan bir Java çekirdek kodu üretmeye karar verdik. Java çekirdek kodunda bellek yönetimi, komut işleme ve konfigürasyon yönetimi yapılmaktadır. Şekil 4.18 de kod üretme sürecini göstermektedir.



Şekil 4.18 : Kod üretme süreci

Java çekirdek modülü uygulama bağlamı, komutların gerçeklenmesi, model soyutlamalarını ve hazır değerlendiricileri içerir.

Çekirdekte komutu soyutlayan bir soyut sınıf bulunmaktadır. Dilde tanımlı olan bütün komutlar bu soyut sınıfı gerçekleyen gerçek sınıflar olarak yazılmıştır. Soyut komut sınıfı tekil olan uygulama bağlamı ve soyut bir "execute" metodu içerir. Komut yürütücüsü tarafından bütün komutların "execute" metodu çağırılır. Komutlar uygulama bağlamını bu metot ile değiştirir. Kod üretici terminal ve operatör sınıflarını ve bütün başlangıç işlemlerinin gerçeklendiği kodu üretir. Şekil 4.19 de komut sınıfın UML diyagramı gösterilmektedir.



Şekil 4.19 : Komut sınıfın UML diyagramı

Komut yürütücü komutları yürütür. Bütün komutlar komut kayıt defterine kaydedilir. Bu kayıt Java kodu aşağıda verilmiştir:

```
private static Map<String, ? extends BaseCommand>
commandRegistry = new HashMap<String, BaseCommand>() {
{
     put("random_select", new RandomSelect());
     put("random_individual",new RandomIndividual());
     put("add_to_population", new AddToPopulation());
     put("find_best", new FindBest());
     put("evaluate", new Evaluate());
     put("clone", new Clone());
     put("use_population", new UsePopulation());
     put("add_to_offspring", new AddToOffspring());
     put("detach", new Detach());
```

```
        put("attach", new Attach());
        put("random_position", new RandomPosition());
        put("prune", new Prune());
        put("grow", new Grow());
        put("offspring_to_population",new
OffspringToPopulation());
 }};
```

Komut yürütücü sınıfı tek bir "execute" metoduna sahiptir. Metot komut ismini katar şeklinde, argümanları ve bir değişkeni parametre olarak kabul eder.

Örnek olarak:

`k1 = detach(pos1)` komutu aşağıdaki gibi üretilir:

```
CommandExecuter.execute("detach","[pos1]","k1");
```

Komut yürütücü ilgili komutu kayıt defterinden bulup ilgili komutun nesnesinin "execute" metodunu çağırır.

5 KULLANIM VE ÖRNEKLER

5.1 Bir noktadan çaprazlama işlemi

Kullanıcı istediği takdirde “crossover” komutu kullandığında iki bireyi çaprazlar ve oluşun bireyleri çocuklar listesine ekleyebilir. Buna örnek olarak aşağıdaki kod toplumdan rasgele iki birey seçip tek noktadan çaprazlayıp oluşan bireyleri çocuklar listesine eklemektedir.

```
CROSSOVER ONE-POINT
SELECTION TOURNAMENT(30)
with ph {
          random_select()
          i1 = clone()
          random_select()
          i2 = clone()
          crossover(i1 i2)
}
```

5.2 Karışık çaprazlama işlemi

Kullanıcı isterse çaprazlama işlemlerini karışık olarak da kullanmak isteyebilir. Aşağıdaki örnekte iki noktadan çaprazlama ve kullanıcının ağaç komutları ile yazdığı başsız tavuk çaprazlaması gösterilmiştir:

```
CROSSOVER ONE-POINT
with po {
         random_select()
         i1 = clone()
         random_select()
```

```
        i2 = clone()
        crossover(i1 i2)
}
with ph {
      random_select()
      i1 = clone()
      i2 = random_individual()
      pos1 = random_position(i1)
      pos2 = random_position(i2)
      k1 = detach(pos1)
      k2 = detach(pos2)
      attach(pos1 k2)
      attach(pos2 k1)
      evaluate(i1)
      evaluate(i2)
      add_to_offspring(i1)
      add_to_offspring(i2)
   }
```

5.3 Bir Değerlendirici Yazmak

Daha önceden de bahsedildiği gibi değerlendiriciler probleme özgüdür. Geliştirilen yapıda kullanıcıya kendi değerlendiricisini yazma olanağı sunulmuştur. Değerlendirici yazarken genel amaçlı bir dil olan Java kullanılacaktır.

Değerlendiricimiz “Evaluator” ara yüzünü gerçeklemek durumundadır.

Ara yüz “evaluate” metodundan oluşur ve de parametre olarak ağaç kabul eder. Kullanıcı ağacı değerlendirdikten sonra ağacın başarım değerini güncellemelidir. Ve parametre olarak döndürmelidir.

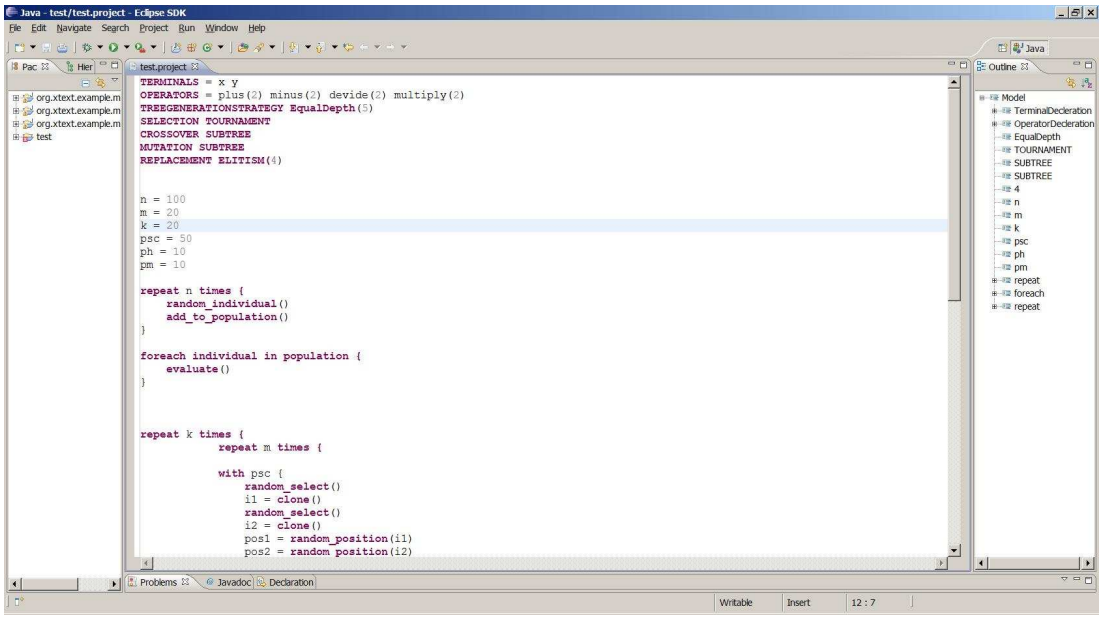
```
public interface Evaluator {
    public GPTree evaluate(GPTree t);
}
```

Örnek bir değerlendirici yazmak isteyelim. Değerlendiricimiz gelen bir ağacın başarım değerini rasgele olarak hesaplasın ve de geri döndürsün.

```
public class MyEvaluator implements Evaluator {
    public GPTree evaluate(GPTree t){
      t.setFitness( Math.random());
      return t;
   }
}
```

5.4 Dil ile Örnek Bir Uygulama Geliştirmek

Xtext daha önceden bahsedildiği gibi dil ile geliştirme yapmamıza yardımcı olacak editörü de bir Eclipse Application olarak üretir. Şekil 5.1 de editörü görebilirsiniz.



Şekil 5.1 : Editör

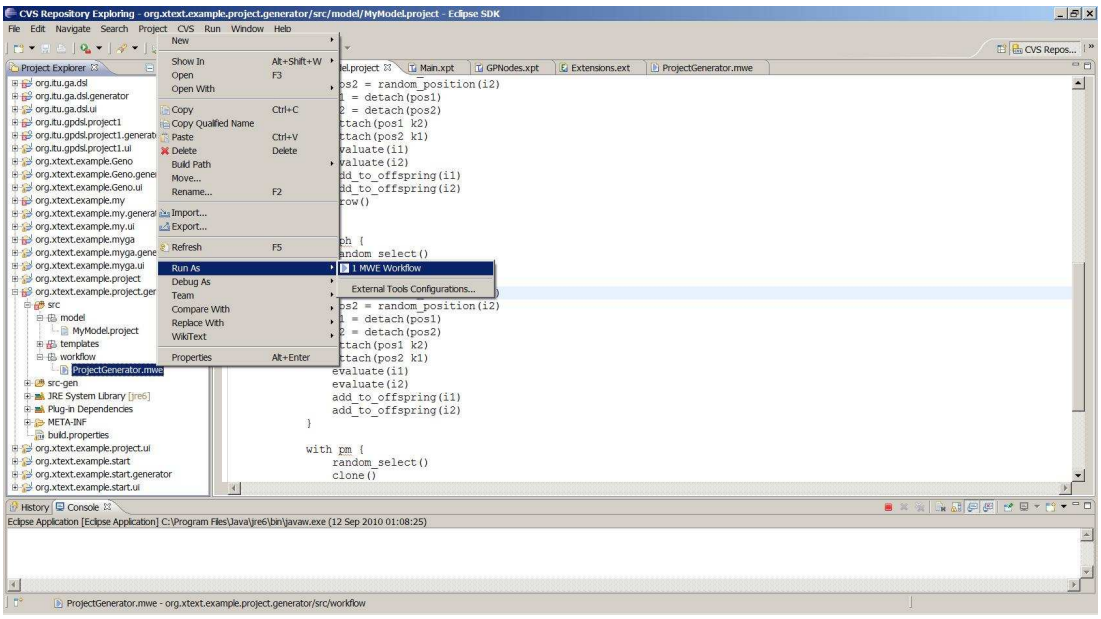
Daha önceden de bahsedildiği gibi değerlendiriciler probleme özgüdür. Geliştirilen yapıda kumlanıcıya kendi değerlendiricisini yazma olanağı sunulmuştur. Değerlendirici yazarken genel amaçlı bir dil olan Java kullanılacaktır.

Örnek olarak çalıştırılacak programlama kodu, sembolik regresyon problemini çözmektedir. Editör ile geliştirilen kod aşağıdadır:

```
TERMINALS = x y

OPERATORS = plus(2) minus(2) devide(2) multiply(2)

TREEGENERATIONSTRATEGY EqualDepth(5)

SELECTION TOURNAMENT

CROSSOVER SUBTREE

MUTATION SUBTREE

REPLACEMENT ELITISM(4)

n = 100

m = 20

k = 20

psc = 50

ph = 10

pm = 10

repeat n times {

    random_individual()

    add_to_population()

}

foreach individual in population {

evaluate()

}

repeat k times {

    repeat m times {
```

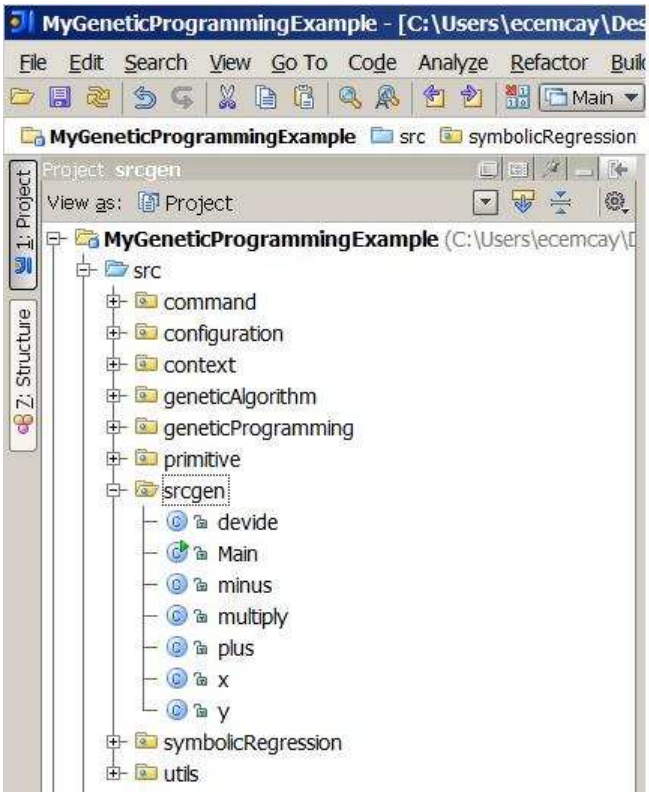
```
with ph {
random_select()
i1 = clone()
i2 = random_individual()
pos1 = random_position(i1)
pos2 = random_position(i2)
k1 = detach(pos1)
k2 = detach(pos2)
attach(pos1 k2)
attach(pos2 k1)
evaluate(i1)
evaluate(i2)
add_to_offspring(i1)
add_to_offspring(i2)
}
with pm {
   random_select()
   clone()
    random_position()
    prune()
    grow()
   evaluate()
   add_to_offspring()
}}
offspring_to_population()
reduce_population(n)
}
```

Kod yazımı bittikten sonra kullanıcı alana özgü dili Java programlama dilline çeviren aracı çalıştırmalıdır. Şekil 5.2 de örnek çalıştırma gösterilmiştir.



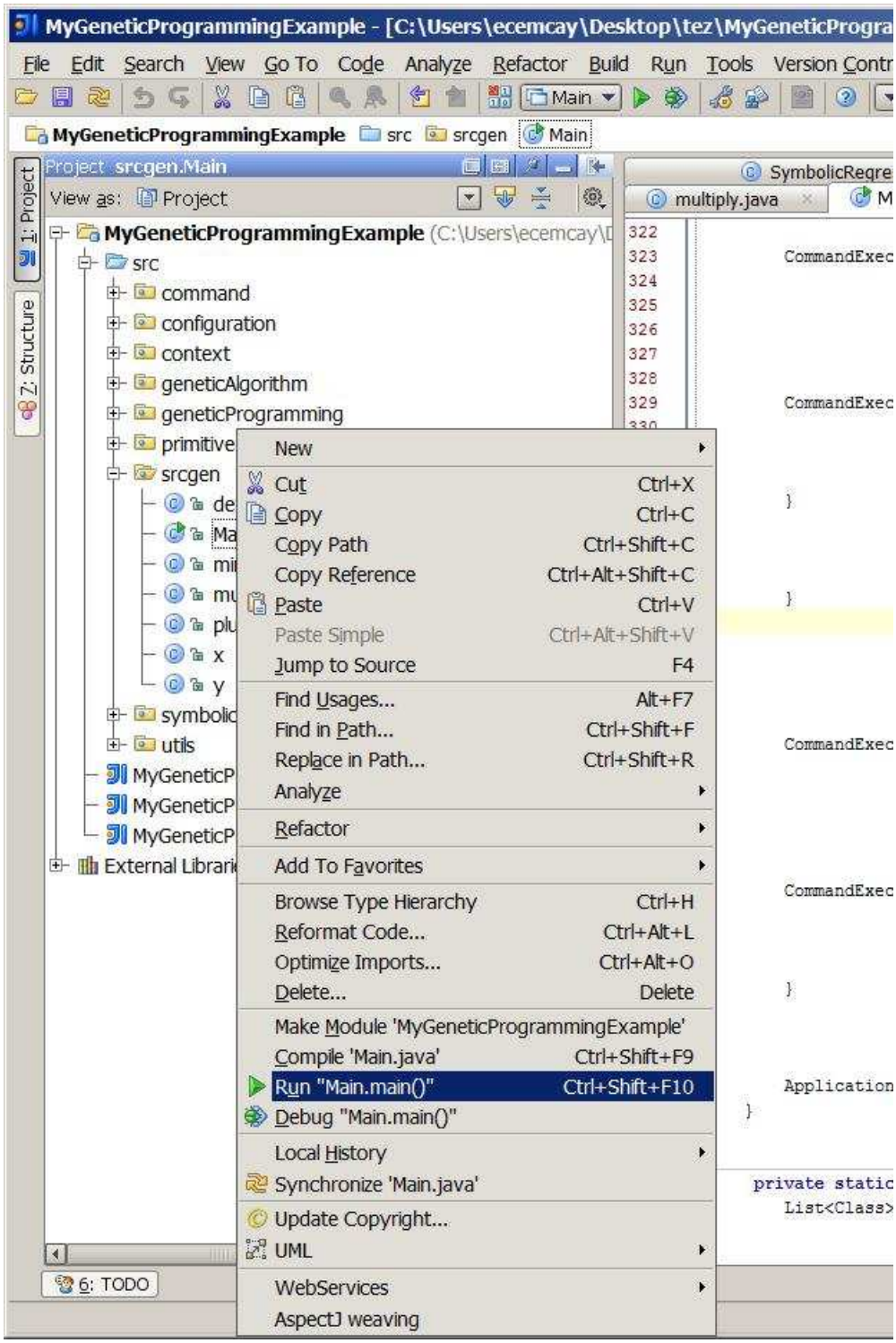
Şekil 5.2 : Kod çevirici

Kod çevirme işlemi Java ile yazılmış diğer bir modülün içine gerekli kodları yaratmıştır. Şekil 5.3 de yaratılan kodlar gösterilmiştir.



Şekil 5.3 : Yaratılan sınıflar

Son olarak kullanıcı programı çalıştırmalıdır. Çalıştırma işlemi yaratılan sınıflardan olan Main sınıfını çalıştırmak ile olur. Şekil 5.4 de çalıştırma işlemi gösterilmiştir.



Şekil 5.4 : Çalıştırma

Çeşitli veri kümelerinde algoritma denemiştir. Aşağıda bazı deneme ve sonuçları bulabilirsiniz.

Örnek veri:

X=3, Y= 1, Sonuç = 10

X=5, Y= 3, Sonuç = 7

X=7, Y= 4, Sonuç = 8

X=8, Y= 1, Sonuç = 11

X=4, Y= 9, Sonuç = 4

X=1, Y= 4, Sonuç = 3

Algoritma çıktısı:

minus(multiply(plus(plus(multiply(y)(y))(devide(y)(x)))(devide(plus(x)(y))(plus(y)(x))))(minus(minus(plus(x)(x))(devide(x)(y)))(minus(minus(x)(x))(devide(x)(y)))))(plus(devide(plus(multiply(x)(x))(devide(y)(x)))(multiply(devide(y)(x))(minus(y)(x))))(plus(multiply(multiply(x)(minus(plus(plus(y)(y))(multiply(y)(y)))(minus(plus(x)(x))(plus(y)(y)))))(minus(x)(x)))(minus(devide(x)(x))(plus(multiply(devide(minus(x)(x))(devide(x)(plus(y)(x))))(plus(multiply(x)(plus(x)(x)))(multiply(y)(x))))(minus(minus(multiply(y)(y))(plus(x)(x)))(devide(plus(y)(y))(y)))))))

Örnek veri:

X=4, Y= 1, Sonuç = 3

X=2, Y= 3, Sonuç = 7

X=1, Y= 5, Sonuç = 2

X=3, Y= 1, Sonuç = 3

X=9, Y= 2, Sonuç = 9

X=11, Y= 4, Sonuç = 3

Algoritma çıktısı:

plus(devide(devide(multiply(plus(y)(plus(y)(x)))(devide(y)(y)))(devide(devide(x)(y))(multiply(x)(x))))(multiply(multiply(plus(y)(y))(plus(x)(x)))(minus(multiply(y)(y))(plus(y)(x)))))(multiply(devide(multiply(multiply(x)(y))(devide(y)(y)))(multiply(plus(x)(devide(devide(minus(multiply(y)(y))(devide(x)(devide(devide(minus(multiply(y)(y))(devide(x)(x)))(multiply(plus(y)(devide(y)(y)))(devide(y)(x))))(devide(x)(y)))))(multiply(plus(y)(devide(y)(x)))(devide(y)(x))))(devide(x)(y)))(y)))(plus(multiply(devide(y)(x))(multiply(x)(y)))(devide(minus(y)(y))(minus(y)(x)))))

6 SONUÇ VE ÖNERİLER

Çalışmada GP ve alana özgü diller kapsamlı olarak araştırılmıştır. Bir alana özgü dil geliştirirken alan hakkında çok detaylı bilgiye sahip olunmalıdır. Alan detaylı olarak incelenirken aynı zamanda dil geliştirme araçları da araştırılmıştır. Alandan dile geçiş çok zorlu bir süreçtir. Alan hakkında bilgi kazanmak ve de ihtiyaçları daha iyi anlamak için genel amaçlı programlama dilleri ile belli GP uygulamaları yazılmıştır. Daha sonradan alandan dile aktarımda burada edinilen tecrübe kullanılmıştır.

Dil bu hali ile kullanılabilir durumdadır fakat gelişime açık çok yönü vardır. Bu yönler kullanıcılardan alınan geri beslemelerle geliştirilebilir.

Karşılaşılan en önemli sorun değerlendiricidir. Değerlendiriciler probleme özel olduklarından genel amaçlı diller ile yazılabilir durumdadır. Değerlendirici için bir ara yüz sağlanmakta fakat bu kullanıcıyı genel amaçlı dilden kurtaramamaktadır. Dile değerlendirici yazma desteği sağlanmalıdır.

Karşılaşılan diğer bir sorun ise geliştirme yapıldıktan sonra kullanıcı dostu olmayacak şekilde programı çalıştırmak için yapılan işlemlerdir. Bu durum düzgün bir paketleme ve komut dosyası ile otomatik hale getirilebilir.

Geliştirilen dilden genel amaçlı bir dil yaratıldığından adım adım çalıştırma mümkün olmamaktadır. Bu da geliştiriciler için çok zor bir durumdur. Adım adım çalıştırma özelliği kod çevirme işlemi ortadan kaldırıldığı takdirde mümkün olabilir. Bu da çok büyük bir geliştirme işidir.

Proje istenilen düzeye gelmiştir. Bir dil tasarlanıp geliştirilmiş ve de çalıştırılabilir programlar yazılmıştır. Sonuçlar ise projenin düzgün çalıştığını göstermektedir.

KAYNAKLAR

[1] **Poli R., Langdon W. B., McPhee N. F.,** 2008: A Field Guide to Genetic Programming, Lulu Enterprises UK

[2] **Mernik, M., Heering, J., Sloane, A.M.,** 2005: When and How to Develop Domain-Specific Languages, ACM Computing Surveys, 37:4, pp. 316-344.

[3] <*http://www.eclipse.org/Xtext*>, alındığı tarih 29.03.2010.

[4] <http://www.alesdar.org/oldSite/IS/chap6-2.html>, alındığı tarih 29.03.2010.

[5]<http://www.openarchitectureware.org/pub/documentation/4.3/html/contents/xtext_tutorial.html>, alındığı tarih 29.03.2010.

[6] **John R. Koza.,** 1992: Genetic Programming, pp. 1, 77, 123

[7] **Niels H. Christenses**., Domain-specific languages in software development and the relation to partial evaluation, pp. 17, 18, 19, 2003

[8] **J. Bentley and J. Bentley.,** 1999: Programming Pearls. Addison-Wesley Professional.

[9] **A. van Deursen, P. Klint, and J. Visser.** 2000: Domain-specific languages: an annotated bibliography.

[10] **P. Hudak.,** 1996: Building domain-specific embedded languages. ACM Comput. Surv., page 196, 1996.

[11] **Matthias Anlauff., Asuman Sünbül.,** 2000: Domain Specific Languages in Software Architecture.

[12] **Michael Jackson.**,1999: Specializing in software engineering. IEEE Software,119-121.

[13] **J. Bell, F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D. P. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou.** 1994: Software design for reliability and reuse: a proof-of-concept demonstration. 396-404.

[14] **Watt, D.A.,** 1990: Programming language concepts and paradigms. Prentice-Hall, Inc., UpperSaddle River, NJ.

[15] **Hudak, P.,** 1996: Building domain-specific embedded languages. ACM Computing Surveys 196–202.

[16] **Hudak, P.,** 1998: Modular domain specific languages and tools.

[17] **Kang, K.C. Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S**. 1990: Feature-oriented domain analysis feasibility study.

[18] **Simos, M.A**. 1995: Organization domain modeling (ODM) 196–205.

[19] **Frakes,W.,** 1998: Diaz, R.P., Fox, C. Domain analysis and reuse environment. 125–141.

ÖZGEÇMİŞ

Ad Soyad: Cem Basar CAYIROGLU

Doğum Yeri ve Tarihi: Kirsehir 20/08/1983

Adres: Jan Van Beersstraat 42 2018 Antwerp Belcika

Lisans Üniversite: Istanbul Teknik Universitesi

Yayın Listesi:

Cem Basar Cayiroglu, H. Turgut Uyar, A. Sima Uyar. 2010: A Domain Specific Language for Genetic Programming.