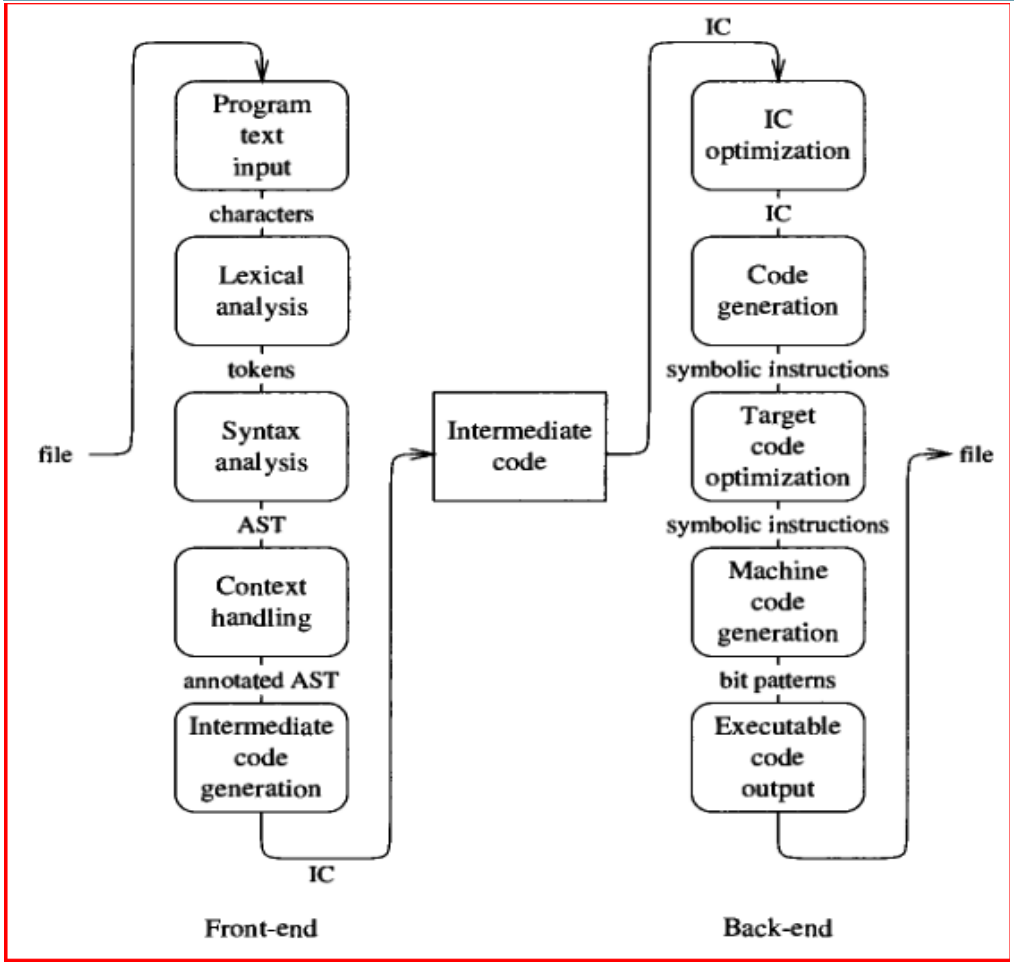
DERLEYİCİ TASARIMI

Bu Haftaki Konu Başlıkları

- Bir derleyicinin Yapısı
  - Program Metnini Okuma
  - Sözcüksel Analiz (Lexical Analysis)
    - Kurallı İfadeler (Regular Expression)
    - Öncelik Kuralları (Precedence Rules)
    - Kısa Gösterimler (Shorthands)
    - Bazı Kurallı İfade Türetimleri
    - Kurallı İfadelerde İstisna Karakterler
    - Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

Bir Derleyicinin Yapısı

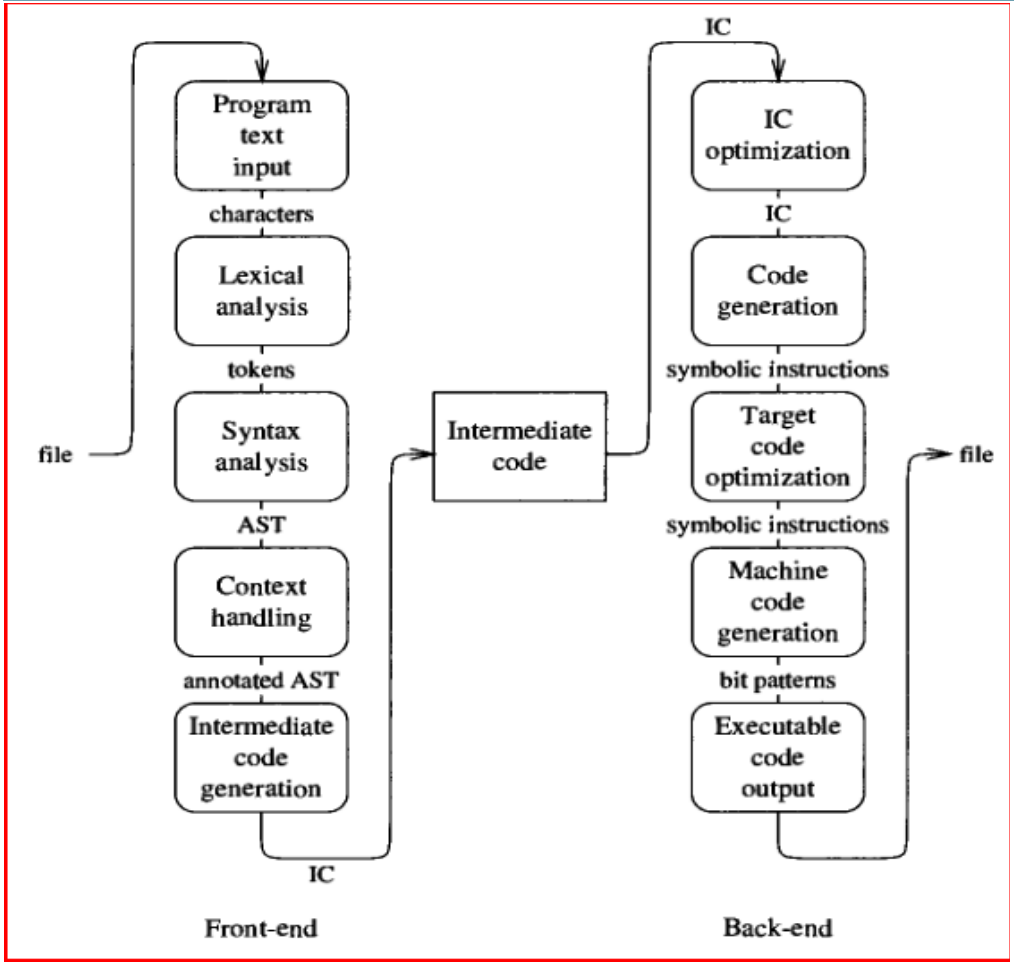


Program metni giriş modülü: Program metnini bulur, okur ve onu bir karakter akışına çevirir.

Sözcüksel analiz modülü: giriş akışındaki tokenleri ayırır ve bu tokenlerin sınıflarına ya da sunumlarına karar verir.

Söz dizimi analizi: token akışını bir AST (**A**bstract **S**yntax **T**ree – Soyut Sözdizimi Ağacı)'ye dönüştürür. Bazı sözdizimi analizörleri iki modülden oluşabilir. İlk modül token akışını okur ve onayladığı her bir sözdizimi yapısı için ikinci modülden bir foksiyon çağırır. İkinci modüldeki fonksiyonlar daha sonra AST'nin düğümlerini oluşturur ve onları bağlar.

Bir Derleyicinin Yapısı



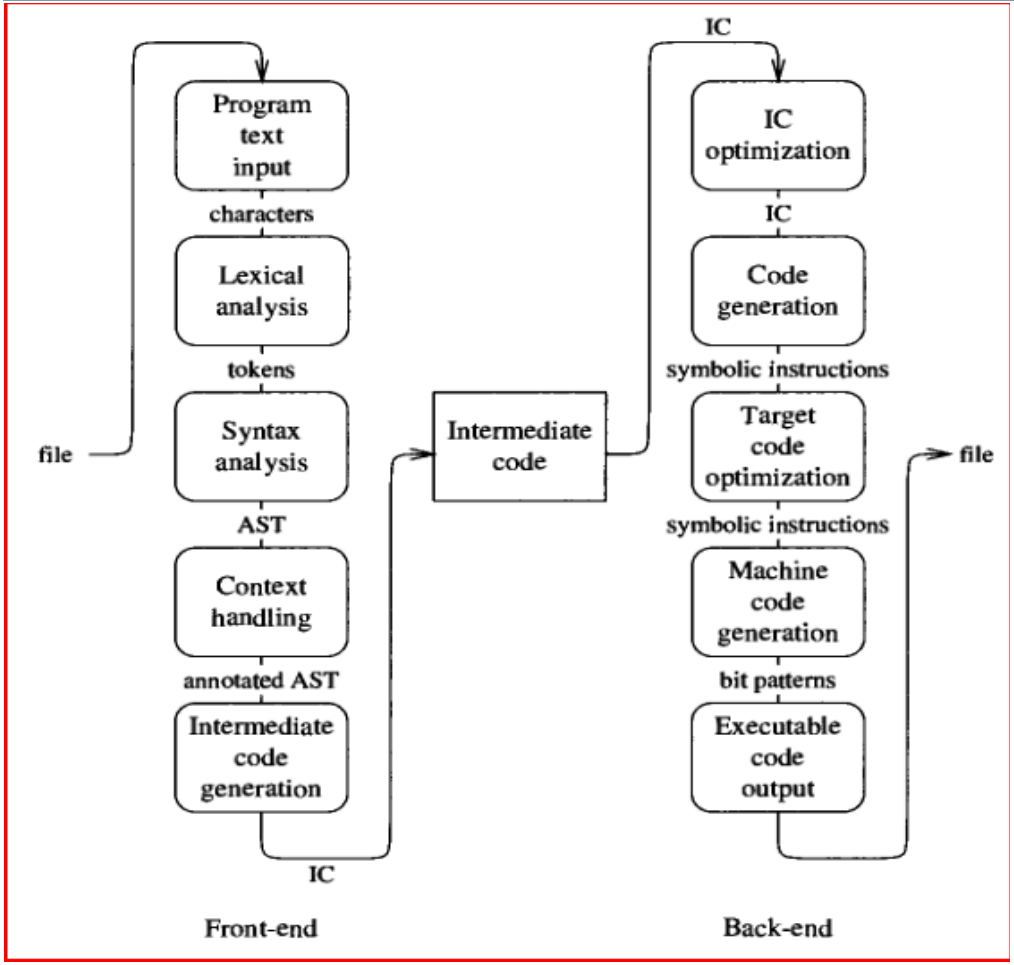
İçerik Yönetimi Modülü: programda farklı yerlerden içerik bilgilerini toplar ve sonuçları düğümlere not olarak yazar. Bu notlar daha sonra içerik kontrollerinin uygulanmasında ya da sonraki modüllere geçişte kullanılır.

Intermediate Code Modülü: AST'deki dile özgü yapıları daha genel yapılara dönüştürür. Bu genel yapılar daha sonra ara kod (**I**ntermediate **C**ode - IC)'u oluşturur. Genellikle IC ifadeleri ve denetim akışı komutlarını içerir.

IC İyileştirme Modülü: kod üretim modülünün etkinliğini arttırmak için IC üzerinde ön süreci uygular.

Kod Üretimi Modülü: AST'yi yaklaşık olarak sembolik bir formda, hedef makine komutlarının düz bir listesi olarak yeniden yazar.

Bir Derleyicinin Yapısı



Hedef Kod İyileştirme Modülü: Sembolik makine komutlarının listesini içerir ve hedef kodu iyileştirmeye çalışır. Bu işlemi hedef makineye ait özellikleri kullanarak daha hızlı, ya da daha kısa makine kodlarının kullanılmasını sağlayarak gerçekleştirir.

IC iyileştirme, kod üretimi ve hedef kod iyileştirme modülleri arasındaki görev sınırları kesin olarak belirlenmemiştir. Eğer kod üretimi oldukça iyiyse basit bir kod iyileştirme işlemi yeterli olabilmektedir.

Makine kod üretimi modülü: Sembolik makine komutlarını uygun bit örneklerine çevirir. Program kodu ve verilerinin makine adreslerine karar verir. Sabitler tablosunu ve taşıma tablosunu oluşturur.

Çalışabilir kod çıkış modülü: kodlanmış makine komutlarını, başlıkları, işletim sistemi için gerekli elemanları, sabitler tablosunu ve taşıma tablosunu çalışabilir bir kod dosyasında bir araya getirir.

Program Metnini Okuma

Program metni karakterlerden oluşur fakat programlama dili tarafından yazılan fonksiyonların doğrudan karakterleri okuma işleminde kullanılması uygun değildir. Çünkü bu fonksiyonlar genel amaçlar için kullanılırlar ve gerekenden daha yavaştırlar.

Eski derleyiciler tampon (buffer) tekniğini kullanarak program dosyasını hızlı bir şekilde okurken bellek tüketimini korumaya çalışıyorlardı. Yeni makinelerde önerilen yöntem ise tüm dosyayı bir sistem çağrısıyla okumaktır. Bu çok hızlı bir giriş yöntemidir ve gerekli bellek miktarı sağlanırken her hangi bir sorun ortaya çıkmaz.

Program Metnini Okuma

Bütün olarak okumanın avantajları;

Çeşitli boyutlardaki tokenleri yönetme kolaylığı sağlar. Bu tokenlerin birçoğu derleyici tarafından daha sonradan kullanılmak üzere depolanabilir ve boyutları önceden biliniyorsa tokenler için alan tahsisi yapılabilir.

İlk string karakterin yeri not edilebilir, sonu bulunabilir, boyutu hesaplanabilir, yer tahsisi yapılabilir ve kopyalanabilir. Eğer giriş dosyası bellekte tüm derleme boyunca tutulursa bir işaretçi ile ilk karakter ve bu karakterin uzunluğu belirlenebilir.

Problemin çıktığı noktada hata mesajları kolaylıkla elde edilebilir.

Sözcüksel Analiz (Lexical Analysis)

Yüksek seviyeli bir dille yazılan kodun çalıştırılabilmesi için en düşük seviyede bilgisayarın işleyebileceği komutlar dizisi haline getirilmesi gereklidir. Bu sebeple kaynak kodu oluşturan karakter katarı anlamlı alt parçalara ayrıştırılmalıdır. Alt parçalara **token**, bunları ayrıştırma işlemine de **lexical analiz** denir.

"Lexical" kelimesi genel anlamda "kelimeler ile ilgili" anlamına gelir. Programlama dillerinde kelimeler değişken isimlerine, sayılara vb. benzeyen nesnelerdir. Bu tür nesneler **token** olarak adlandırılırlar. Lexical analiz işleminin çıktısı olan tokenlar bir sonraki aşama olan syntax analiz bileşenine aktarılır.

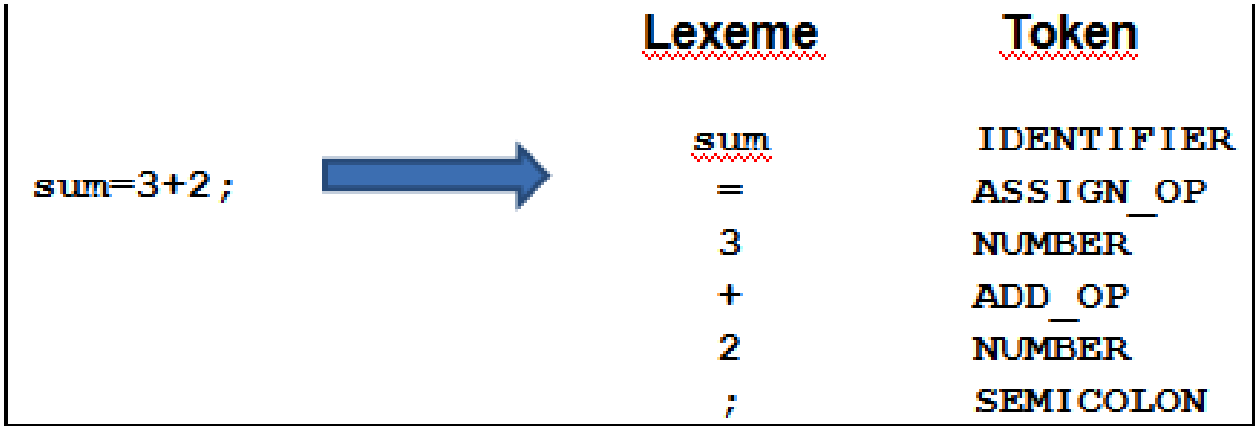
Token sözdizimsel bir kategori belirtir. Bu kategoriler doğal diller için "isim", "sıfat", "fiil" vb. olabilirken, programlama dilleri için "matematiksel sembol" veya "anahtar kelime" gibi alt ifadeler olurlar.

Sözcüksel Analiz (Lexical Analysis)

Token belirten bir alt karakter katarı **lexeme** olarak adlandırılır. Programlama dili içinde kullanılması mümkün olan tüm lexemeler tanımlanırken örüntüler (pattern) kullanılır. Örüntüler kurallı ifadeler kullanılarak tanımlanır.

Bir lexical analiz aracı, ya da kısaca lexer şu 3 işlevi gerçekleştirebilmelidir:

1. Bütün boşlukları ve açıklamaları temizlemeli,
2. Karakter katarı içindeki tüm tokenleri bulmalı,
3. Bulunan token için lexeme ve bulunduğu satır numarası gibi özellikler döndürülmelidir.



Sözcüksel Analiz (Lexical Analysis)

- Token Class (or Class)
  - In English:

    Noun, verb, adjective, ...

  - In a programming language:

    Identifier, Keywords, '(', ')', Numbers, ...

Sözcüksel Analiz (Lexical Analysis)

- Token classes correspond to sets of strings.

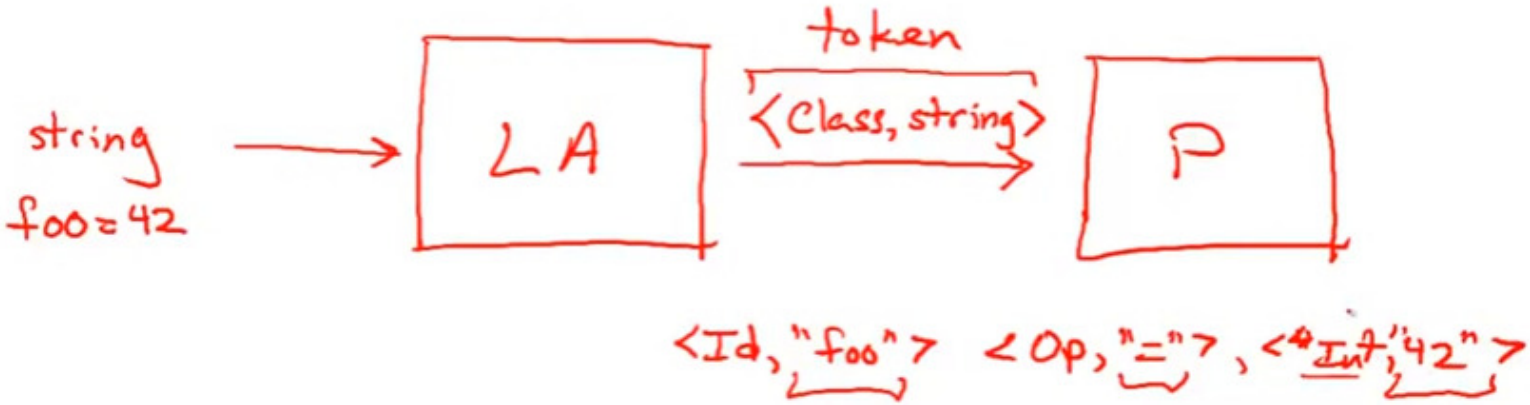
- Identifier:
  - *strings of letters or digits, starting with a letter*
- Integer: A1 Foo B17
  - *a non-empty string of digits* 0 12 001 00
- Keyword:
  - *"else" or "if" or "begin" or …*
- Whitespace:
  - *a non-empty sequence of blanks, newlines, and tabs*

if ___ (→ whitespace

Sözcüksel Analiz (Lexical Analysis)

- Classify program substrings according to role
- Communicate tokens to the parser

token class



Sözcüksel Analiz (Lexical Analysis)

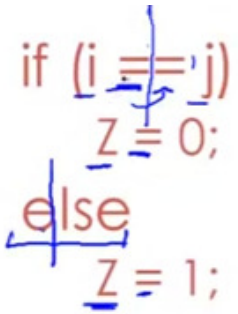
İki önemli nokta:

1. Amaç stringi parçalamaktır. Okuma soldan sağa yapılır ve bir defada bir token bulunur.

2. “Lookahead” bir tokenın nerede bittiği ve diğerinin nerede başladığına karar vermek için gereklidir.

Basit örnekte bile lookahead konuları vardır.

```
if (i == j)
    Z = 0;
else
    Z = 1;
```



Sözcüksel Analiz (Lexical Analysis)

- var < 5+3; ifadesi söz dizimsel olarak geçerli bir ifadedir ve lexical analyzer çıktısı:

variable(var), compOperator(<), intConstant(5), arithOperator(+), intConstant(3), endOfLine(;)

şeklinde olur.

- var < (-+67;); ifadesi ise söz dizimsel olarak geçersiz bir ifadedir. Fakat bu lexical analyzer'ın problemi değildir. Lexical analyzer herhangi bir hata vermeden şu çıkıtıyı üretmelidir:

variable(var), compOperator(<), leftParanthesis((), intConstant(-), arithOperator(+), intConstant(67), endOfLine(;), rightParanthesis()), endOfLine(;)

- Görüldüğü üzere lexical analyzer'ın tek kaygısı, kaynak koddaki tüm terminallerin dil tarafından tanınıp tanınmadığıdır. Bir de bunun yanında, yorum satırlarının sonlanıp sonlamadığı da parsing aşamasına gelmeden lexical analysis aşamasında belli olur.

Sözcüksel Analiz (Lexical Analysis)

Lexical analizin ana amacı bir sonraki faz olan syntax analizinin işini kolaylaştırmaktır. Teoride, lexical analiz süresince yapılan iş syntax analizin bir parçası olarak yapılabilir (basit sistemlerde uygulanmaktadır). Ancak bu işlemleri ayrı fazlara ayırmak hızı ve verimliliği artırır, karmaşıklığı azaltır.

Bir lexer'ın yazılması genelde zor değildir. Öncelikle ilk karakter okunur ve tokenin bir harf mi, bir sayı mı, bir değişken mi yoksa başka bir şey mi olduğunu anlamak için test edilir. Daha sonra sıra ile bir sonraki tokene geçilir. Ancak bu bir problemi çözmek için iyi bir yöntem değildir. Her bir olası token test edilirken girişin benzer değerleri tekrar tekrar okunmalıdır. Ayrıca yazılan bir lexer karmaşık olabilir ve sürdürülmesi zordur. Bu nedenle lexer'lar genelde bir lexer generator (lexer üreteci) ile oluşturulurlar. Benzer yöntem syntax analizi fazında da gerçekleştirilir.

Sözcüksel Analiz (Lexical Analysis)

Program okuma modülü ve Lexical Analyser bir derleyicinin yalnızca basit parçaları olmalarına rağmen birçok işi gerçekleştirirler ve ön-uç işleminde harcanan zamanın %30'u bu süreçte geçebilir.

Lexical analiz için tanımlamalar kurallı ifadeler (Regular Expressions) kullanılarak yazılırlar. Kurallı ifadeler stringlerin tanımlanması için kullanılan matematiksel bir gösterimdir. Kurallı ifadeler kullanılarak elde edilen lexer'lar "Sonlu Otomat (Finite Automat)" olarak adlandırılan basit programların bir sınıfındadırlar.

Derleme Örneği

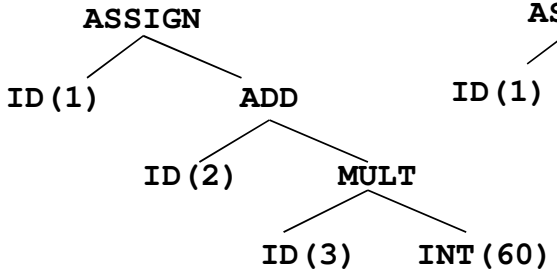
Kaynak Dil:

```
cur_time = start_time + cycles * 60
```

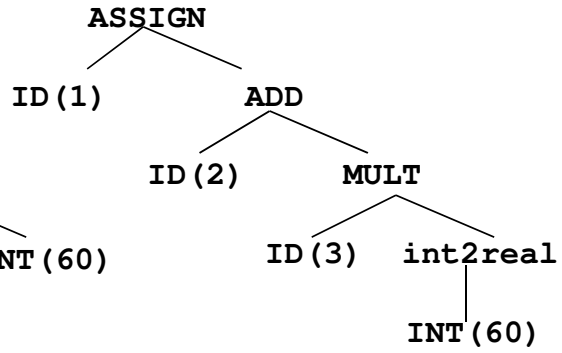
Lexical Analiz:

```
ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)
```

Sözdizim Analizi:



Anlamsal Analiz:



Derleme Örneği

Ara Kod:

```
temp1 = int2real(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimizasyon:

Adım 1:

```
temp1 = 60.0
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Adım 2:

```
temp2 = id3 * 60.0
temp3 = id2 + temp2
id1 = temp3
```

Adım 3:

```
temp2 = id3 * 60.0
id1 = id2 + temp2
```

Optimize Edilmiş Kod:

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Hedef Dil:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Kurallı İfadeler (Regular Expressions)

Tokenları belirtmek için farklı yaklaşımlar vardır. Düzgün diller en yaygın olanıdır.

“Bir tanımlayıcı; bir harf ile başlayan, harf, sayı ya da alt çizgilerin dizisidir. Burada ardışık iki alt çizgiye izin verilmez”.

Böyle bir tanımlama, o dilin kullanıcısı için yeterlidir. Fakat derleyici yapısı için amaçlanan token şekilleri **“kurallı ifadeler”** olarak adlandırılan ifadelerle daha kullanışlıdır.

Sonlu durum otomatları (FSA) ile tanımlanabilen dilleri ifade etmede kullanılırlar.

Kurallı İfadeler (Regular Expressions)

Tanım. S bir dizi karakterdir ve içindeki karakterler belirli kurallarla bir dil oluşturur.

(Σ bu dilin alfabesidir.)

- alfabe= İngilizce karakterler
- Dil = İngilizce cümleler

- İngilizce karakterlerden oluşan her string İngilizce bir cümle değildir.

Kurallı İfadeler (Regular Expressions)

- Eğer A bir kurallı ifade ise, L(A) A ile gösterilen dili belirtmek için kullanılır.
- L(“if”|”then”|”else”) dili sadece “if”, “then” ve “else” ifadelerini tanıyabilen bir dil belirtir.
- Lexical analiz işleminde tarayıcı (scanner), kaynak kodun içinde önceden düzenli ifadelerle tanımlanmış anahtar kelimeleri arar ve bunların token tiplerini belirler.

Kurallı İfadeler (Regular Expressions)

Tüm integer sabitleri ya da tüm değişken isimleri bir string grubudur. Bu grubun her bir harfi özel bir alfabeden oluşur. Böyle bir string grubu dil(language) olarak adlandırılır. Integer'lar için alfabe 0-9 arası sayılar ile oluşur ve değişken isimleri için alfabe hem harflerden hem de sayılardan oluşur (ve belki diğer karakterlerden).

Verilen bir alfabede string gruplarını kurallı ifadeler ile tanımlarız. Kurallı ifadeler insanların kullanması ve anlaması için kolay ve öz olan matematiksel bir gösterimdir.

Kurallı İfadeler (Regular Expressions)

Kurallı İfade	Dil (String Grubu)	Informal tanımlaması
a	{a}	Bu grup tek harfli string “a” dan oluşur
ε	{“ “}	Boş string içerir
s|t	L(s) U L(t)	Her iki dilden oluşan simgeler
st	{vw | v € L(s),w € L(t)}	Stringler ilk dilden bir string ve ikinci dilden bir stringin bağlanması ile oluşur
s*	{“ “}U{vw | v € L(s),w € L(s*)}	Dildeki her bir string s dilindeki her hangi bir sayıdaki stringin bağlanması ile oluşur

Kurallı İfadeler (Regular Expressions)

Atomik düzgün ifadeler tek bir karakterden oluşur. Tek karakter: ‘c’

L(‘c’) = { “c” } (her c Є Σ)

Arka arkaya Ekleme(Concatenation): AB (A ve B düzgün ifadeler olsun.)

L(AB) = { ab | a Є L(A) ve b Є L(B) }

Bileşim(Union)

L(A | B) = { s | s Є L(A) or s Є L(B) }

Örnekler:

‘if’ | ‘then‘ | ‘else’ = { “if”, “then”, “else”}

(‘0’ | ‘1’) (‘0’ | ‘1’) = { “00”, “01”, “10”, “11” }

Tekrarlama (Iteration): A*

L(A*) = { “” } U L(A) U L(AA) U L(AAA) U …

Örnekler:

‘0’* = { “”, “0”, “00”, “000”, …}

‘1’ ‘0’* = {1 ile başlayıp 0’lar ile devam eden stringler}

Epsilon: ε

L(ε) = { “ ” } $\varepsilon \neq \{\}$

Keyword: “else” or “if” or “begin” or …

‘else’ | ‘if’ | ‘begin’ | …

Integer: Boş olmayan rakamlar stringi

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

number = digit digit*

Kısaltma: A+ = A A*

Whitespace: boş olmayan boşluk, yeni satır ve tab lerden oluşan stringler

(‘ ‘ | ‘\t’ | ‘\n’)+

Kurallı İfadeler (Regular Expressions)

İki bileşik operatör vardır. Biri görünmez operatördür ki bu birbirine bağlamayı gösterir. Örneğin ab* ifadesinde a ve b arasında oluşur. İkincisi | operatörüdür. Bu operatör alternatifleri ayırır. Örneğin ab* | cd?

st'ye örnek olarak, eğer L(s), {"a", "b"} ve L(t)'de {"c", "d"} ise L(st), {"ac", "ad", "bc", "bd"} olur.

s* örnek olarak, L(s) {"a", "b"} ise L(s*), {" ", "a", "b", "aa", "ab", ba", "bb", "aaa", "aab",……..} olur.

Bir kurallı ifade **ab*** sonsuz sayıda grup üretebilir {a, ab, abb, abbb..}. Verilen bir kurallı ifade tarafından üretilmiş bir stringe sahip olduğumuz zaman söyleyebiliriz ki kurallı ifade stringle uyumludur.

Kurallı İfadeler (Regular Expressions)

a(a* U b*)b kurallı ifadesi ile önce bir a üretilir. Ardından iki durumdan birisine göre devam edilir. İstenen sayıda a üretilir veya b üretilir. Son olarak da bir b üretilir.

Diller yazılırken genellikle{“a”, “b”} yerine {a, b} şeklinde yazılırlar. “ ” yerine ise ε simgesi kullanılır. v ve w tanımlanmamış tek stringleri göstermek için kullanılırlar. Örneğin abw, ab ile başlayan her hangi bir stringi ifade eder.

Kurallı İfadeler (Regular Expressions)

Bir string *s* ve bir düzgün ifade *R* verildiğinde

$$s \in L(R)\,?$$

sorusuna cevap bulmak gerekmektedir.

1. Bir küme token seçin
   - Number, Keyword, Identifier, …

2. Her bir token için bir düzgün ifade yazın
   - Number = digit+
   - Keyword = 'if' | 'else' | …
   - Identifier = letter (letter | digit)*
   - OpenPar = '('
   - …

Kurallı İfadeler (Regular Expressions)

3. R dilini bütün tokenları kapsayacak şekilde yazın.

$$R = \text{Keyword} \mid \text{Identifier} \mid \text{Number} \mid \ldots$$
$$= R_1 \mid R_2 \mid R_3 \mid \ldots$$

If $s \in L(R)$ ➔ s bir tokenın lexeme değeridir.

- Dahası $s \in L(R_i)$ (belirli bir "i" için)
- Bu "i" bulunan tokenın hangisi olduğunu gösterir.

Kurallı İfadeler (Regular Expressions)

R = Whitespace | Integer | Identifier | ‘+’

- “f +3 +g” stringini çözümleyin
  - “f” Identifier’a dolayısıyla R’ye uyar
  - “+“ “+” ya dolayısıyla R’ye uyar
  - …
  - token-lexeme ikilileri ise

    (Identifier, “f”), (‘+’, “+”), (Integer, “3”)

    (Whitespace, “ “), (‘+’, “+”), (Identifier, “g”) olur

Kurallı İfadeler (Regular Expressions)

- Algoritmada belirsizlikler vardır.
- Örnek:

  R = Whitespace | Integer | Identifier | ‘+’
- “foo+3” çözümleyin
  - “f” Identifier’a uyar
  - Ama aynı zamanda “fo” Identifier ’a uyar, hatta “foo” da Identifier’a uyar, ama “foo+” Identifier’a uymaz
- Eğer
  - Hem $x_1 \ldots x_i \in L(R)$ ve hem de $x_1 \ldots x_K \in L(R)$ doğru ise ne kadar girdi parçası kullanılır?
  - “Maximal munch(en büyük lokma)” kuralı: R’a uyan mümkün olan en uzun stringi seçin

Kurallı İfadeler (Regular Expressions)

R = Whitespace | Integer | Identifier | ‘+’

- “=56” : çözümleyin
  - Hiçbir önek(prefix) R’a uymamaktadır: ne “=“, ne “=5”, ne de “=56”
- Çözüm:
  - Son kural olarak bütün uygun olmayan stringlere uyacak bir kural eklenir.
- Lexer araçları şu şekilde bir gösterime izin verir:

  $R = R_1 \mid ... \mid R_n \mid \text{Error}$
  - Token Error diğer tokenlardan hiçbirine uygunluk çıkmazsa kullanılır.

Öncelik Kuralları (Precedence Rules)

Kurallı ifadelerde a | ab* biçiminde farklı yapıcı sembolleri bir araya getirdiğimiz zaman farklı alt ifadelerin nasıl gruplandırılacağının önceliği net değildir. Parantezler sembollerin gruplandırılmasını belirgin hale getirmek için kullanılır. Ayrıca öncelik kuralları da kullanılır. Bu yöntem matematiksel birleşme ile aynıdır (3+4*5 ifadesinin anlamı 3'ün 4 ve 5'in çarpımına eklenmesidir).

Kurallı ifadeler için uygulanan kural ise şudur; * birbirine bağlamaya göre daha önceliklidir. Birbirine bağlama ise | sembolüne göre daha önceliklidir. Örneğin a | ab* aynı zamanda a | (a (b*)) demektir.

Öncelik Kuralları (Precedence Rules)

| operatörü ilişkisel ve değişmelidir. Birbirine bağlama ilişkiseldir ve | üzerine dağılır. Yanda bu özellik ve kurallı ifadelerin diğer matematiksel özellikleri gösterilmiştir.

$$
\begin{aligned}
(r|s)|t &= r|s|t = r|(s|t) \\
s|t &= t|s \\
s|s &= s \\
s? &= s|\varepsilon \\
(rs)t &= rst = r(st) \\
s\varepsilon &= s = \varepsilon s \\
r(s|t) &= rs|rt \\
(r|s)t &= rt|st \\
(s^*)^* &= s^* \\
s^*s^* &= s^* \\
ss^* &= s^+ = s^*s
\end{aligned}
$$

Kısa Gösterimler (Shorthands)

Kurallı ifade tanımlamasındaki yapılar sayı stringlerini ve değişken isimlerini tanımlamak için yeterliyken sıklıkla daha uygun kısa gösterimler kullanılır. Örneğin pozitif integer sabitlerini tanımlamak istiyorsak bunu bir ya da daha fazla rakam ile sağlarız.

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

Farklı sayıların geniş aralıkları yukarıdaki ifadenin kullanımını oldukça gereksiz hale getirir. Bu durum değişken isimleri kullanıldığında daha da kötü olur. Bu nedenle, harf grupları için kısa gösterimler kullanılır. Köşeli parantez içerisindeki harf dizisi bu harflerin grubunu simgeler. Örneğin [ab01], a | b | 0 | 1 için kısa gösterim olarak kullanılır.

Kısa Gösterimler (Shorthands)

Ayrıca, [0123456789]'u [0-9] olarak kısaltmak için aralık gösterimi de kullanılabilir. Bu köşeli parantez içerisinde birkaç aralık da birleştirilebilir. [a-zA-Z] örneği hem küçük hem de büyük karakterleri simgeler.

Aralıklar kullanıldığı zaman içerilen semboller için sıralanma bilgisi olmalıdır. Yukarıda kullanılan harf ve sayılar için genellikle karışıklık yoktur. Eğer [0-z] yazarsak bunun ne anlama geldiği açık değildir. Karışıklıktan kaçınmak için aralık gösterimi sadece harflerin aralıkları ve sayıların aralıkları için kullanılır.

Kısa Gösterimler (Shorthands)

Yukarıdaki örneğe tekrar dönecek olursak daha kısa olarak [0-9][0-9]* şeklinde yazılabilir.

s* , sıfır ya da daha çok s varlığını gösterdiği için sayı grupları iki kez tanımlandı. Bu tür sıfır bulunmayan tekrarlamalar oldukça sık kullanılır. Bu nedenle diğer bir kısa gösterim s^+ kullanılır. s^+ bir ya da daha çok s varlığını ifade eder. Bu gösterim ile integer tanımlaması $[0\text{-}9]^+$ olarak kısaltılabilir. Benzer bir uygulama da bir şeylerin sıfır ya da bir tane varlığına sahip olunduğunda ortaya çıkar. Bu nedenle s | ε için s? kısa gösterimi kullanılır. + ve ?'nin bağlama önceliği * ile aynıdır.

Kısa Gösterimler (Shorthands)

Aşağıda tipik programlama elemanlarının birkaç örneği gösterilmektedir.

Kelimeler: if gibi bir kelime i ve f gibi iki kurallı ifadenin birleşmesinden meydana gelen bir kurallı ifade tarafından tanımlanır.

Integer'lar: Bir integer sabiti seçimli bir işaret tarafından izlenen boş olmayan sayı gruplarından oluşur. **[+-]? [0-9]$^{+}$** . Bazı dillerde işaret ayrı bir semboldür ve sabitin kendi parçası değildir. Burada işlem işaret ve sayı arasında boşluk vererek sağlanır.

Kısa Gösterimler (Shorthands)

Float'lar: kayan noktalı bir sayı seçimli bir işarete sahiptir. Bundan sonra bir ondalık noktası tarafından izlenen sayı dizisi olarak tanımlanmış bir tam sayı kısmı ve daha sonra bir diğer sayı dizisi vardır. Bu sayı gruplarından her hangi birisi boş olabilir. Son olarak seçimli bir üs parçası vardır. Bu e ya da E harfi ve bunu takip eden (işaret seçimli) integer bir sabitten oluşur. Eğer sabit için bir üs parçası varsa tamsayı kısmı integer bir sabit olarak yazılabilir.

Örnek float sayı belirtimleri; 3.14 -3. .23 3e+4 11.22E-3

Bu ifadenin kurallı ifadesi;

$$[+-]?((([0-9]^{+}.[0-9]^{*}|.[0-9]^{+})([eE][+-]?[0-9]^{+})?)|[0-9]^{+}[eE][+-]?[0-9]^{+})$$

Kısa Gösterimler (Shorthands)

String Sabitler: bir string sabiti bir tırnak işareti ile başlar, sembol dizi ile takip edilir ve sonunda yine tırnak işareti ile sonlanır. Burada tırnak işareti arasında izin verilen semboller üzerinde bazı kısıtlamalar vardır. Örneğin, satır atla karakteri gibi. “\n\n” iki satır atla karakteri içeren bir string’dir. Aşağıdaki kurallı ifade ile bu stringler için kurallı ifade tanımlayabiliriz. Burada izin verilen semboller alfabetik ve sayısal değerlerden ve birde harf tarafından izlenen \ karakterinden oluşmuştur.

```
"([a-zA-Z0-9]|\[a-zA-Z])*"
```

Bazı Kurallı İfade Türetimleri

Kurallı İfade	Tanımladığı Küme
Φ	{ }
λ	{λ}
00+11	{ 00+11}
a(b+c)	{ab, ac}
ab*	{a, ab, abb, abbb, abbbb, …}
a(bb+cc)d*	{abb, abbd, abbdd, …, acc, accd, accdd, …}
a*	{λ, a, aa, aaa, aaaa, …}
(0+1)*	{λ, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, …}
a(b+cd*)*a	{aa, aba, acda, acdda, acddda, abca, acba, …}

Kurallı İfadelerde İstisna Karakterler

Kurallı ifadelerde, operatör olarak *, +, ? ve / karakterleri, ayırıcı olarak [,], (ve) karakterlerinin kullanımı gerçek karakterler olarak kullanımlarına engel olur.

Bir kaç yöntemle bu karakterlerin kendi yerlerine kullanılması sağlanabilir. İlki \ karakterini kendi yerine kullanılacak karakterin önünde kullanmaktır. * ifadesi yıldız işaretinin kendi yerine kullanılmasını sağlar ve \\ kendi karakterini simgeler. Diğer bir işaret “ karakteridir. Harici karakter tırnak içine alınır. “*” * işaretini simgeler, “ “ “ “ kendini simgeler.

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

Bir LA direk olarak yazılabilir ya da otomatik olarak (sonlu otomatlar kullanılarak) üretilebilir. Her iki durumda da kurallı ifadelere dayanan token belirtimi olmalıdır. Ancak kıyasla bir LA'nın yazılması daha kolaydır. Buna başlamak için en iyi yol giriş ilk karakteri üzerine bir durum yapısı oluşturmaktır.

```
letter → [a-zA-Z]
digit → [0-9]
underscore → _
letter_or_digit → letter | digit
underscored_tail → underscore letter_or_digit+
identifier → letter letter_or_digit* underscored_tail*
```

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

Bu kurallı ifadenin programlama dilinde kodlanması

```
#define is_end_of_input(ch)      ((ch) == '\0')
#define is_layout(ch)            (!is_end_of_input(ch) && (ch) <= ' ')
#define is_comment_starter(ch)   ((ch) == '#')
#define is_comment_stopper(ch)   ((ch) == '#' || (ch) == '\n')

#define is_uc_letter(ch)         ('A' <= (ch) && (ch) <= 'Z')
#define is_lc_letter(ch)         ('a' <= (ch) && (ch) <= 'z')
#define is_letter(ch)            (is_uc_letter(ch) || is_lc_letter(ch))
#define is_digit(ch)             ('0' <= (ch) && (ch) <= '9')
#define is_letter_or_digit(ch)   (is_letter(ch) || is_digit(ch))
#define is_underscore(ch)        ((ch) == '_')

#define is_operator(ch)          (strchr("+-*/", (ch)) != 0)
#define is_separator(ch)         (strchr(";,(){}", (ch)) != 0)
```

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

Örnek uygulamada LA 5 token sınıfını ayırabilir. Integer ifadeler, tek karakterli tokenler, token sınıfı ERRONEOUS ve EOF ve tanımlanan 'identifier'. Tek karakterli tokenler operatörlerdir (+, -, *, /, ;, ,, (,), {, }). Yorumlar bir # karakteri ile başlar ve # karakteri ile ya da satır sonu ile biter.

Diğer taraftan arka-uç girişte hangi digit'in sunulduğu ile ilgilidir. Bu nedenle bütün süreçlerden sonra tokenlerin muhafaza edilmesi gerekir. Bir token hakkındaki bilgiyi iki kısma ayırırız;

- token sınıfı ve
- onun gösterimi

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

```
/* Define class constants; 0-255 reserved for ASCII characters: */
#define EOF             256
#define IDENTIFIER      257
#define INTEGER         258
#define ERRONEOUS       259

typedef struct {
    char *file_name;
    int line_number;
    int char_number;
} Position_in_File;

typedef struct {
    int class;
    char *repr;
    Position_in_File pos;
} Token_Type;

extern Token_Type Token;

extern void start_lex(void);
extern void get_next_token(void);
```

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

LA'nın başlık dosyası aşağıda gösterilmiştir. Girişi yönetebilmek için yerel verinin tanımlamalarını içerir. Global bir token tanımı ve derleyiciyi başlatan start_lex() fonksiyonu.

```
#include    "input.h"           /* for get_input() */
#include    "lex.h"

/* PRIVATE */
static char *input;
static int dot;                 /* dot position in input */
static int input_char;          /* character at dot position */

#define next_char()             (input_char = input[++dot])

/* PUBLIC */
Token_Type Token;

void start_lex(void) {
    input = get_input();
    dot = 0; input_char = input[dot];
}
```

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

Get_next_token() fonksiyonu, note_token_position() fonksiyonunu çağıran token.pos alanında tanımlanmış token pozisyonunu kayıt eder.

```
void get_next_token(void) {
    int start_dot;

    skip_layout_and_comment();
    /* now we are at the start of a token or at end-of-file, so: */
    note_token_position();

    /* split on first character of the token */
    start_dot = dot;
    if (is_end_of_input(input_char)) {
        Token.class = EOF; Token.repr = "<EOF>"; return;
    }
    if (is_letter(input_char)) {recognize_identifier();}
    else
    if (is_digit(input_char)) {recognize_integer();}
    else
    if (is_operator(input_char) || is_separator(input_char)) {
        Token.class = input_char; next_char();
    }
    else {Token.class = ERRONEOUS; next_char();}

    Token.repr = input_to_zstring(start_dot, dot-start_dot);
}
```

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

Progman metni içindeki boşluklar ve yorum satırları tespit ediliyor

```
void skip_layout_and_comment(void) {
    while (is_layout(input_char)) {next_char();}
    while (is_comment_starter(input_char)) {
        next_char();
        while (!is_comment_stopper(input_char)) {
            if (is_end_of_input(input_char)) return;
            next_char();
        }
        next_char();
        while (is_layout(input_char)) {next_char();}
    }
}
```

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

```
void recognize_identifier(void) {
    Token.class = IDENTIFIER; next_char();

    while (is_letter_or_digit(input_char)) {next_char();}

    while (is_underscore(input_char)
    &&    is_letter_or_digit(input[dot+1])
    ) {
        next_char();
        while (is_letter_or_digit(input_char)) {next_char();}
    }
}

void recognize_integer(void) {
    Token.class = INTEGER; next_char();
    while (is_digit(input_char)) {next_char();}
}
```

Kurallı İfadelere Göre Bir Lexer'ın Programla Kodlanması

```
#include    "lex.h"       /* for start_lex(), get_next_token() */

int main(void) {
    start_lex();

    do {
        get_next_token();
        switch (Token.class) {
        case IDENTIFIER:    printf("Identifier"); break;
        case INTEGER:       printf("Integer"); break;
        case ERRONEOUS:     printf("Erroneous token"); break;
        case EOF:           printf("End-of-file pseudo-token"); break
        default:            printf("Operator or separator"); break;
        }
        printf(": %s\n", Token.repr);
    } while (Token.class != EOF);
    return 0;
}
```