***Parsing*, Compiladores, Marco Cristo**

Gramática: quádrupla (N, T, P, S) onde N são as variáveis (não terminais), T são os símbolos (terminais) mais o símbolo λ (*string* nula), P são as regras N = (N ∪ T)\* e S é a regra inicial.

Gramáticas regulares são aquelas em que as regras têm a forma A = Bx | xB | x | λ.

São reconhecidas por autômatos finitos determinísticos (AFDs). Contudo, construções comuns em linguagens de programação, como:

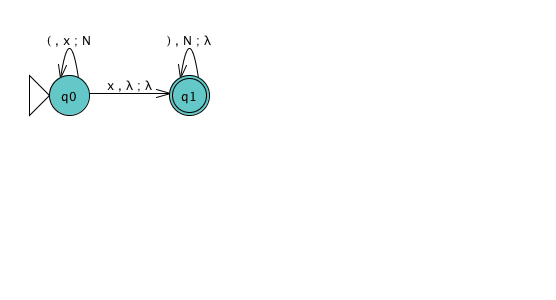
E = x | “(” E “)” .

Não podem ser reconhecidas por gramáticas regulares, pois geram uma linguagem não regular. Não há um AFD que possa reconhecer E.

*Solução*: usar gramáticas livres de contexto!

GLC são gramáticas com regras da forma A = S, onde S é uma sentença que combina terminais e não terminais (em EBNF, ainda inclui meta-símbolos {...}, (...), [...] e |) e A é não terminal.

E pode agora ser reconhecida por um autômato de pilha (aceita por sentença consumida e pilha vazia):



CFGs têm limitações. O mais importante é que linguagens de programação, em geral, são sensíveis ao contexto[[1]](#footnote-1), como vemos no exemplo abaixo em C++:

int x = 0; int a = x + 1; **ok**

string x = “teste”; int a = x + 1; **Erro, pois x é string (contexto) e +1 vale para números**

De fato, linguagens têm muitas *condições contextuais*, como por exemplo:

|  |  |  |  |
| --- | --- | --- | --- |
| Comando = | Designador | “=” | Exp “;” |
|  | Variável,  Elemento de vetor, campo de objeto |  | Compatível com tipo do designador |

A solução com GSC é pouco usada por levar a algoritmos de *parsing* muito complexos. Solução comum é usar GLC em análise sintática e verificar condições contextuais como parte da análise semântica.

**Um primeiro algoritmo de parsing**

A solução mais simples de parsing é uma análise da esquerda para direita em que o casamento é feito considerando o próximo caractere mais à esquerda da regra. Por exemplo, parse a sentença xyyz usando a gramática G:

G: S = A .

A = x B .

B = z .

B = y B .

|  |  |  |  |
| --- | --- | --- | --- |
| S = A .  A = x B .  B = z .  B = y B . | S = | A  xB  B  yB  B  yB  B  z  aceita | *esq* 🡪 *dir*  xyyz  xyyz  yyz  yyz  yz  yz  z  z |

Note que sentenças xxx e xxxy não seriam aceitas.

Este parser é chamado LL(K), com K = 1, ou seja, LL(1) ou LL1. L de left to right, L de leftmost non-terminal e K=1 é o número de tokens para ver adiante (look-ahead tokens).

Este é um parsing facílimo de implementar. Por exemplo, dada a gramática:

X = a Y c .

Y = b b .

|  |  |  |  |
| --- | --- | --- | --- |
| Parse() {  Scan()  X()  Check(EOF)  } | X() {  Check(a)  Y()  Check(c)  }  Y() {  Check(b)  Check(b)  } | Check(x) {  if sym == x {  Scan()  return t  } else  ERROR  } | Scan() {  t = la  la = Scanner.next()  sym = la.tipo  } |

Cada não terminal é implementado como uma função. Chamadas recursivas ao não terminal são simples chamadas recursivas à função correspondente. Logo, a tradução é direta. Checar um item consiste basicamente em ver se próximo token é o token esperado. Se não for, ele não é consumido.

Finalmente, note que o parsing para EBNF é tão simples quanto o BNF. Eis exemplos de códigos-base:

|  |  |  |
| --- | --- | --- |
| X = a Y | Y b.  Y = c | d. | X = [ a b ] c. | X = a { Y } b.  Y = c | d. |
| X() {  if sym == a {  Check(a)  Y()  } elif sym in First(Y)  Y()  Check(b)  } else  ERROR  } | X() {  if sym == a {  Check(a)  Check(b)  }  Check(c)  } | X() {  Check(a)  while sym in First(Y)  Y()  Check(b)  } |

Mas há um preço a se pagar pela simplicidade de checar apenas 1 token a frente. Por exemplo, veja o parsing de xxxz para a gramática abaixo:

|  |  |  |  |
| --- | --- | --- | --- |
| S = A .  A = x B .  A = x C .  B = x B .  B = y .  C = x C .  C = z . | S = | A  xB  B  xB  B  xB  B  erro | xxxz  xB ou xC -> escolhe xB  xxxz  xxz  xxz  xz  xz  z |

Problema! A sentença xxxz não foi reconhecida, embora fosse válida (caso fosse aplicada a regra A = xC, por exemplo). Soluções possíveis:

1. Backtracking (caro – tenho que sempre voltar ao último ponto com alternativas possíveis) e
2. Evitar condições ambíguas (o que é feito mais comumente, mas nem sempre possível, como mostram os códigos em Fortran: DO 10 I = 1,2 versus DO 10 I = 1.2. Note que não há como compreender a instrução até observamos 4 tokens a frente)

Para linguagens modernas, é comum optar por evitar construções ambíguas (Ex: gramáticas do Python, Ruby, etc). Mas como fazer isso?

Se considerarmos uma gramática BNF, podemos evitar ambiguidades se projetarmos regras que atendam a duas restrições, conhecidas como restrições LL1 R1 e R2. Para entender estas restrições, vamos introduzir duas funções:

*First*: (N ∪ T)\* 🡪 T\*, conjunto dos primeiros símbolos de S, S ∈ (N ∪ T)\*

Definição: a ∈ First(A) se A =>\* a S, a ∈ T, S ∈ (N ∪ T)\* e A ∈ N

(=>\* indica zero a várias derivações)

Ex: First(A = aB) = {a}

First(A = λ) = {λ}

Definição: a ∈ First(w) se w =>\* a S, w ∈ (N ∪ T)\*

Ex: First(A = ab) = {a}

First(Ca) = First(C).

Se C = λ, First(C) = a

*Follow*: N 🡪 T\*, conjuntos dos símbolos que seguem A, A ∈ N

Definição: a ∈ Follow(A) se S =>\* X Aa Y

Ex: Dado S 🡪\* X A Y, Follow(A) = First(Y)

Dado S 🡪\* X A, Follow(A) = Follow(S)

Dadas as funções First e Follow, as restrições LL1 R1 e R2 são descritas a seguir.

**Restrição R1:** duas sentenças alternativas não podem ter o primeiro símbolo em comum

A = S1 | S2 | ... | Sn .

First(Sj) ∩ First(Sk) é vazia para j != k

Ex:

A = a B | b C | c D . não viola!

A = a B | a C | b D . viola! First(a B) ∩ First(a C) = {a} != vazio.

Neste segundo caso, o parser não saberia que regra aplicar (a B *ou* a C)

Ex:

1. Ek = ak Ek

First(Ek) = First(ak Ek) = {ak}

1. Ek = Bk Ej

Bk = ak1 | ak2 | ... | akn

First(Ek) = First(Bk Ej) = First(ak1) ∪ First(ak2) ∪ ... ∪ First(akn)

Se akj = λ, First(Ej) deve ser incluído. Se akj = λ *e* First(Ej) = λ, λ deve ser incluído

*Exercício*: A viola R1 na gramática a seguir?

S = A .

A = B . 1

A = C . 2

B = x B . 3

B = y . 4

C = x C . 5

C = z . 6

First(A1) = First(B) = First(x B) ∪ First(y) = {x, y}

First(A2) = First(C) = First(x C) ∪ First(z) = {x, z}

First(A1) ∩ First(A2) = {x} que não é vazio

Logo, A viola R1.

*Exercício*: B viola a restrição 1 em G’, dada a seguir?

S = A .

A = B x . 1

B = x y . 2

B = λ . 3

First(B1) = First(x y) = {x}

First(B2) = First(λ) = {λ}

Intersecao de Firsts é vazia (logo, não viola R1)

Contudo, veja o que ocorre no parsing da string x:

|  |  |  |  |
| --- | --- | --- | --- |
| S = A .  A = B x .  B = x y .  B = λ. | S = | A  Bx  xyx  erro | x  x  x |

Note que se usasse B = λ, teria dado certo. Mas como λ é um token nulo, ele não pode ser usado no casamento. Logo, há problemas quando regras alternativas geram λ.

**Restrição R2:** Se uma produção A é formada pela disjunção de alternativas em que ao menos uma produz lambda, nenhum símbolo que segue A pode ser também um dos primeiros símbolos gerados por A

A = S1 | S2 | ... | Sn . com Sk = λ para algum k

First(Sj) ∩ Follow(A) é vazia para j != k ou

First(A) ∩ Follow(A) é vazia

Note que gramática G’ viola Restrição R2. Ou seja:

First(B1) = First(x y) = {x} = Follow(B)

Ex:

A = BD | CB . (1,2)

B = x B z | y | λ . (3,4,5)

C = w | z . (6,7)

D = x | z . (8,9)

OBS: Note que em análise de Follow, será comum garantir que um símbolo $ segue a primeira regra da gramática. Assim, a gramática anterior será re-escrita como:

S = A $ . $ indica o fim da gramática

A = BD | CB . (1,2)

B = x B z | y | λ . (3,4,5)

C = w | z . (6,7)

D = x | z . (8,9)

Todos os NTs (A, B, C e D) têm alternativas e B gera λ.

B, C e D claramente não violam R1. Mas, e A?

First(A1) = First(BD) = First(B) ∪ First(D) (pois B gera λ) = First(x B z) ∪ First(y) ∪ First(λ) ∪ First(D) = {x, y, z, λ}

First(A2) = First(CB) = First(C) = {w, z}

First(A1) ∩ First(A2) = {z} != vazio

Logo, viola!

E R2 para B?

First(B) = {x, y, λ}

Follow(B)

= note que B aparece na direita de 1, 2 e 3

= First(D) ∪ Follow(A) ∪ First(z) = {x, $, z}

First(B) ∩ Follow(B) = {x} != vazia 🡪 VIOLA!

Mas e para o caso do EBNF, como fazer?

Podemos sempre (1) usar regras específicas para EBNF ou (2) converter EBNF para BNF. O caso (1) é mais direto, porém mais complexo, uma vez que requer a checagem de mais que duas restrições. São elas:

I – Alternativas

S = A | B | C . First(A) ∩ First(B) = vazio

First(A) ∩ First(C) = vazio

First(B) ∩ First(C) = vazio

S = ( A | λ ) B . First(A) ∩ First(B) = vazio

S = ( A | λ ) . First(A) ∩ Follow(A) = vazio

II – Opcionais

S = [ A ] B . First(A) ∩ First(B) = vazio

S = [ A ] . First(A) ∩ Follow(A) = vazio

III – Iteração

S = { A } | B . First(A) ∩ First(B) = vazio

S = { A } . First(A) ∩ Follow(A) = vazio

*Obs*: note que o gerador de compilador deve checar e informar violações para você. Assim, não vamos focar muito em restrições LL1 para EBNF. Em sala de aula, vamos apenas lidar com restrições LL1 para BNF.

Como exemplo da solução (2), vejamos uma conversão de EBNF para BNF:

A = B [ E ] C .

D = F { G } H .

Isso é equivalente a:

A = B E’ C .

D = F G’ H .

E’ = E | λ.

G’ = G G’ | λ.

Como esta gramática está em BNF, podemos usar as mesmas restrições R1 e R2.

Note que tipicamente espera-se que uma sentença de entrada termine em um marcador de fim. Logo, é comum a gramática ser completada com este marcador. Veja o exemplo:

S = A .

A = x y .

A = λ .

Note que não é possível determinar o Follow de A. Mas como espera-se que a sentença de entrada termine em um marcador de fim, a gramática pode ser modificada para indicar isto:

S = A $ .

A = x y .

A = λ .

Neste caso, o Follow de A é $.

**Exercícios**

1) A gramática dada a seguir é LL1?

S = A .

A = B (x | z) | (w | z) B .

B = x B z | { y } .

Transformando, temos:

S = A $ .

A = B x | B z | w B | z B .

B = x B z | B’

B’ = y B’ | λ .

A viola R1.

First(Bz) = First(B) ∪ {z}

= First(x B z) ∪ First(B’) ∪ {z} = {x, z} ∪ First(B’) = {x, z, y, λ}

First(zB) = {z}

Logo, First(Bz) e First(zB) não tem interseção nula!

A viola R2 pois First(B) tem interseção com Follow(B). Veja abaixo:

Resumindo:

First Follow

A {w, z, x, y} {$}

B’ {y, λ} {x, z, $}

B {x, y, λ} {x, z, $}

2) A gramática abaixo é LL1?

A = a

| B C d.

B = [b] a.

C = c {d}.

Transformando, temos:

A = a

| B C d .

B = B’ a .

B’ = b | λ .

C = c D’ .

D’ = d D’ | λ .

Verificando A:

First(A1) = {a}

First(A2) = First(B C d) = First(B) = First(B’) U First(a)/pois B’ gera λ = {a, b}

Logo, A viola R1 pois a interseção não é vazia.

Verificando D’:

First(D’) = {d, λ}

Follow(D’) = Follow(D’)/recursão U Follow(C) = Follow(C) = {d}

Logo, D’ viola R2 pois a interseção não é vazia.

Não é, pois viola R1 e R2.

Como mudar a gramática para ser LL1?

A = a | B C d.

B = [b] a.

C = c {d}.

Observe que C gera cd\*. E C é apenas invocada a partir de A = B C d. Logo, nesta gramática, o trecho C d corresponde a cd\*d que é equivalente a cdd\*. Isso pode ser escrito como:

A = a | B C.

B = [b] a.

C = c d {d}.

Convertendo para BNF e inserindo um EOF, temos ($):

S' = A $.

A = a | B C.

B = B' a.

B’ = b | λ .

C = c d D'.

D’ = d D’ | λ .

Como não mudamos o início de A, ela continua violando R1. Vejamos D' em relação a R2:

First(D’) = {d, λ}

Follow(D’) = Follow(D’) U Follow(C) = Follow(C) = {$}

Logo, D’ não viola R2 pois a interseção é vazia.

Para corrigir a violação de R1 em A, podemos usar fatoração (mais a seguir):

Logo, A = a | B C pode ser reescrita como:

A = a | B C.

A = a | B’ a C. usando B = B’ a.

A = a | (b | λ) a C. usando B’ = b | λ.

A = a | b a C | a C.

A = a (λ | C) | b a C.

A = a A’ | b a C. onde A’ = λ | C.

Assim, temos:

S' = A $.

A = a A’ | b a C.

A’ = λ | C.

C = c d D'.

D’ = d D’ | λ .

Note agora que nenhuma regra viola R1.

**Casos práticos**

A seguir, descrevemos algumas técnicas de transformação de gramáticas, de forma a evitar violações de LL1.

***Fatoração***

RepeatSt = “REPEAT” SequenceSt “UNTIL” Cond

| “REPEAT” SequenceSt “FOREVER”

Após fatorar os elementos que violam LL1, temos:

RepeatSt = “REPEAT” SequenceSt RepeatTail

RepeatTail = “UNTIL” Cond

| “FOREVER”

Nem sempre aplicamos a fatoração. Veja o caso:

St = IfSt | OtherSt .

IfSt = “IF” Cond “THEN” St

| “IF” Cond “THEN” St “ELSE” St .

Após fatoração, temos:

St = IfSt | OtherSt . 1, 2

IfSt = “IF” Cond “THEN” St IfTail . 3

IfTail = “ELSE” St | λ . 4, 5

Viola R1? Não

Viola R2? Sim!

Note que IfTail gera λ. Logo, First(IfTail) não pode ter interseção com Follow(IfTail). Mas....

First(IfTail) = First(“ELSE” St) = {“ELSE” , λ }

Follow(IfTail) = Follow(IfSt)/3 = Follow(St)/1 = First(IfTail)/3 = {“ELSE”, λ }

Solução:

1. use uma estrura não redundante (Ex: if...elif...else em python, if...elseif...else...endif do Ada).
2. Ignore a redundância, assuma que o parsing vai considerar que o else atual pertence ao último else (comportamento default de parser em profundidade) e assuma que o programador sabe disso (esta, de fato, é a convenção em muitas linguagens como C, Java, etc).

***Eliminação de Recursão à Esquerda***

Note que R1 não permite recursão à esquerda. Por exemplo:

A = B | A B.

First(A1) = First(B)

First(A2) = First(A B) = First(A) = First(A1) ∪ First(A2) que não pode ter interseção nula com First(B).

Portanto, recursão à esquerda deve ser substituída por recursão à direita, sem violar R1 (pois a solução trivial A = B | B A, viola R1). Para tanto, observe que a forma A = A X | Y gera a linguagem L(Y { X }). Logo pode ser escrita como:

A = Y Z .

Z = X Z | λ .

Onde não há qualquer violação! Logo, A = B | B A pode ser reescrita como:

A = B Z .

Z = B Z | λ .

Ex: elimine a recursão à esquerda da seguinte gramática.

A = A b | A c | d | e .

A deve gerar a linguagem L((d | e) { b | c }). Logo pode ser escrita como:

A = d R | e R .

R = b R | c R | λ .

**Mais exercícios**

1) Elimine a recursão na seguinte gramática:

S = A a | b .

A = A c | S d | λ .

Primeiro, substituímos S para eliminar interações entre regras:

S = A a | b .

A = A c | A a d | b d | λ .

Agora eliminamos as recursões à esquerda de A. Note que A gera (bd | λ) (c | a d)\*. Logo:

S = A a | b .

A = b d A' | A'.

A' = c A' | a d A' | λ .

2) Converta para BNF:

Exp = ["+"|"-"] Termo { ("+"|"-") Termo }.

Termo = Fator { ("\*"|"/") Fator }.

Fator = numero | ident | "(" Exp ")".

Exp = Sinal Termo RE.

Sinal = "+" | "-" | λ.

RE = "+" Termo RE

| "-" Termo RE

| λ.

Termo = Fator RT.

RT = "\*" RT

| /" RT

| λ.

Fator = numero | ident | "(" Exp ")".

3) Mude para BNF sem recursao

E = E (“+” | “-”) T | T.

T = T (“\*” | “/”) F | F.

F = num | “(” E “)”.

BNF ->

E = E S T | T

T = T P F | F.

F = num | “(” E “)”

S = “+” | “-”

P = “\*” | “/”

Sem E-Recursão ->

E = T E’

E’ = λ | S T E’

T = F T’

T’ = λ | P F T’

F = num | “(” E “)”

S = “+” | “-”

P = “\*” | “/”

3) Corrija a seguinte gramática (com ajuda do COCO):

COMPILER Block

CHARACTERS

upper = 'A'..'Z'.

lower = 'a'..'z'.

letter = upper + lower.

TOKENS

id = letter {letter}.

IGNORE '\n' + '\t' + '\r'

PRODUCTIONS

Block = "BEGIN" StSeq "END".

StSeq = St {";" St}.

St = [ Block | Decl | Assign | WhileSt | DoSt | IfSt ].

Decl = id {"," id} ":" Type.

IfSt = "IF" Exp ["THEN"] StSeq ["ELSE" StSeq].

Exp = id.

WhileSt = "WHILE" Exp "DO" SqSeq "END".

DoSt = "DO" StSeq "WHILE" Exp.

Assign = id ":=" Exp.

Type = "INTERGER" | "BOOLEAN" | "CHAR".

END Block.

O COCO deve informar que StSeq e St geram λ, e que há violações de LL1 em StSeq (";"), St ("WHILE" e id) e IfSt ("ELSE"). Pode corrigi-las?

1. Gramáticas Sensíveis ao Contexto (GSCs) são aquelas cujas regras têm a seguinte forma: x A = S, onde x é um símbolo não terminal (que pode designar o contexto) e S é uma sentença. Se não há restrições em nenhum dos lados da regra (ou seja, pode haver sentenças à esquerda e à direita, ex: x A = x B y C), a gramática é dita irrestrita (GI). GIs são reconhecidas por máquinas de Turing e GSCs por máquinas de Turing com fita finita. [↑](#footnote-ref-1)