Compiladores:

Apontamentos sobre linguagens livres de contexto

(ano letivo de 2022-2023)

Artur Pereira, Miguel Oliveira e Silva

Maio de 2023

Comp 2023

Nota prévia

Este documento representa apenas um compilar de notas sobre a matéria teórico-prática lecionada na unidade curricular "Compiladores", sem pretensões, por isso, de ser um texto exaustivo. No caso do último capítulo, na realidade, é apenas um enumerar das secções do *Dragon Book* usadas na produção dos *slides*. A sua leitura deve apenas ser encarada como ponto de partida e nunca como único elemento de estudo. Os *slides*, em muitos aspetos, estão mais completos que estes apontamentos.

Comp 2023

Conteúdo

Comp 2023

Comp 2023

Capítulo 1

Gramática livre de contexto

Considere uma estrutura G, definido sobre o alfabeto $T = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, com as regras de rescrita

$$\begin{array}{rcl} S & \rightarrow & \varepsilon \\ & | & X\ S \\ X & \rightarrow & \texttt{a} \\ & | & \texttt{c} \\ & | & \texttt{a b c} \end{array}$$

Que palavras se podem gerar a partir de S? Podem gerar-se 0 ou mais concatenações de X, sendo cada X substituível por `a`, `c` ou `abc`. Ou seja, pode gerar-se a mesma linguagem que a descrita pela expressão regular $e = \texttt{(a|c|abc)}^*$. Embora a linguagem seja regular, a gramática não o é. A produção $S \rightarrow X\ S$ viola a definição de gramática regular, porque tem dois símbolos não terminais.

Considere agora a gramática G

$$\begin{array}{rcl} S & \rightarrow & \varepsilon \\ & | & \texttt{a}\ S\ \texttt{b} \end{array}$$

definida sobre o alfabeto $T = \{\texttt{a}, \texttt{b}\}$. Qual é a linguagem L descrita pela gramática G? A partir de S podem gerar-se as palavras ε, `ab`, `aabb`, $\cdots$, ou seja $L = \{\texttt{a}^n\texttt{b}^n \mid n \geq 0\}$. Pode provar-se que esta linguagem não é regular. Por conseguinte, não é possível definir uma expressão regular, gramática regular ou autómato finito que a represente.

Exemplo 1.1

Considere sobre o alfabeto $T = \{\texttt{a}, \texttt{b}\}$, a gramática

$$\begin{array}{rcl} S & \rightarrow & \texttt{a}\ S \\ & | & \texttt{a}\ X \\ X & \rightarrow & \texttt{a b} \\ & | & \texttt{a}\ X\ \texttt{b} \end{array}$$

Comp 2023

Esta gramática gera a linguagem $L = \{\mathtt{a}^m\mathtt{b}^n \mid n > 0 \wedge m > n\}$. O símbolo X gera a linguagem $\mathtt{a}^n\mathtt{b}^n$, com $n > 0$. O símbolo S gera $\mathtt{a}^k$, com $k > 0$, seguido de X. A conjugação resulta nas palavras $\mathtt{a}^k\mathtt{a}^n\mathtt{b}^n$, com $k, n > 0$.

Exemplo 1.2

Considere sobre o alfabeto $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, a gramática

$$\begin{array}{rcl} S & \rightarrow & \mathtt{c} \\ & \mid & \mathtt{a}\ S\ \mathtt{a} \\ & \mid & \mathtt{b}\ S\ \mathtt{b} \end{array}$$

Esta gramática gera a linguagem $L = \{w\mathtt{c}w^R \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$, onde w^R representa a palavra w com os símbolos colocados por ordem inversa.

As linguagens dos dois últimos exemplos também não são regulares, não podendo, por isso, ser descritas por expressões regulares. Correspondem a linguagens designadas **livres de contexto**, ou **independentes do contexto**. As gramáticas adequadas para descrever estas linguagens têm todas as suas produções da forma $A \rightarrow \beta$, em que A é um símbolo não terminal e β é uma sequência constituída por zero ou mais símbolos terminais ou não terminais.

No exemplo seguinte apresenta-se uma gramática **dependente do contexto**.

Exemplo 1.3

Considere sobre o alfabeto $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, a gramática

$$\begin{array}{rcl} S & \rightarrow & \mathtt{a}\ S\ B\ C \\ & \mid & \mathtt{a}\ \mathtt{b}\ C \\ C\ B & \rightarrow & B\ C \\ \mathtt{b}\ B & \rightarrow & \mathtt{b}\ \mathtt{b} \\ \mathtt{b}\ C & \rightarrow & \mathtt{b}\ \mathtt{c} \\ \mathtt{c}\ C & \rightarrow & \mathtt{c}\ \mathtt{c} \end{array}$$

Esta gramática gera a linguagem $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n > 0\}$. Note que, por exemplo, na produção $\mathtt{b}\,B \rightarrow \mathtt{b}\,\mathtt{b}$ o símbolo não terminal B apenas se pode transformar em $\mathtt{b}$ se tiver um $\mathtt{b}$ imediatamente à sua esquerda.

1.1 Definição de gramática livre de contexto

Formalmente, uma gramática livre de contexto é um quádruplo $G = (T, N, P, S)$, onde

- T é um conjunto finito não vazio de símbolos terminais;

Comp 2023

- N, sendo $N \cap T = \emptyset$, é um conjunto finito não vazio de símbolos não terminais;
- P é um conjunto de produções, cada uma da forma $\alpha \rightarrow \beta$, onde
  - $\alpha \in N$
  - $\beta \in (T \cup N)^*$
- $S \in N$ é o símbolo inicial.

Nas produções, α e β são designados por **cabeça da produção** e **corpo da produção**, respetivamente. Note que relativamente à definição de gramática regular a diferença está na definição de β. No caso das gramáticas dependentes do contexto, como é o caso da do exemplo 1.3, a cabeça das produções (α) deixa de ter de ser necessariamente um único símbolo não terminal.

1.2 Derivação

$\mathcal{D}$ Dada uma produção $u \rightarrow v$ e uma palavra $\alpha u \beta$, chama-se **derivação direta** à rescrita de $\alpha u \beta$ em $\alpha v \beta$, denotando-se

$$\alpha u \beta \Rightarrow \alpha v \beta$$

Em algumas situações, será usada o termo **passo de derivação** como sinónimo de derivação direta.

$\mathcal{D}$ Chama-se **derivação** a uma sucessão de zero ou mais derivações diretas, denotando-se

$$\alpha \Rightarrow^* \beta$$

ou, equivalentemente,

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n = \beta$$

onde n é o comprimento da derivação.

- A notação $\alpha \Rightarrow^n \beta$ é usada para representar uma derivação de comprimento n
- A notação $\alpha \Rightarrow^+ \beta$ é usada para representar uma derivação de comprimento não nulo

$\mathcal{D}$ Dada uma gramática $G = (T, N, P, S)$ e uma palavra $u \in (T \cup N)^+$, o conjunto das **palavras derivadas** a partir de u é representado por

$$D(u) = \{v \in T^* \ : \ u \Rightarrow^* v\}$$

Comp 2023

$\mathcal{D}$ A **linguagem gerada pela gramática** $G = (T, N, P, S)$ é representada por

$$L(G) = D(S) = \{v \in T^* \ : \ S \Rightarrow^* v\}$$

Nas gramáticas livres de contexto, em geral, em cada passo de uma derivação estão envolvidos vários símbolos não terminais. Como as produções têm a forma $\alpha \to \beta$, com $\alpha \in N$ e $\beta \in (T \cup N)^*$, β pode conter vários símbolos não terminais, que serão introduzidos na derivação se a produção for utilizada.

A gramática seguinte, definida sobre p alfabeto $T = \{\mathtt{a}, \mathtt{b}\}$, é livre de contexto e gera a linguagem $L = \{w \in T^* \mid \#(\mathtt{a}, w) = \#(\mathtt{b}, w)\}$

$$\begin{array}{lcl} S & \to & \mathtt{a}\ B \mid \mathtt{b}\ A \\ A & \to & \mathtt{a} \mid \mathtt{a}\ S \mid \mathtt{b}\ A\ A \\ B & \to & \mathtt{b} \mid \mathtt{b}\ S \mid \mathtt{a}\ B\ B \end{array}$$

A palavra `aabbab` tem 4 derivações possíveis, das quais são apresentadas duas.

1. $\underline{S} \Rightarrow \mathtt{a}\underline{B} \Rightarrow \mathtt{aa}\underline{B}B \Rightarrow \mathtt{aab}\underline{B} \Rightarrow \mathtt{aabb}\underline{S} \Rightarrow \mathtt{aabba}\underline{B} \Rightarrow \mathtt{aabbab}$
2. $\underline{S} \Rightarrow \mathtt{a}\underline{B} \Rightarrow \mathtt{aa}B\underline{B} \Rightarrow \mathtt{aa}B\mathtt{b}\underline{S} \Rightarrow \mathtt{aa}B\mathtt{ba}\underline{B} \Rightarrow \mathtt{aa}\underline{B}\mathtt{bab} \Rightarrow \mathtt{aabbab}$

Usou-se o sublinhado para destacar o símbolo não terminal expandido em cada derivação direta.

A derivação 1 designa-se por **derivação à esquerda**, porque em cada passo se expande o símbolo não-terminal mais à esquerda. A derivação 2 designa-se por **derivação à direita**, porque em cada passo se expande o símbolo não-terminal mais à direita.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.3, "Derivations".

1.3 Operações sobre gramáticas livres de contexto

A classe das gramáticas livres de contexto é fechada sobre as operações de reunião, concatenação e fecho, mas não o é sobre as operações de intersecção e complementação.

1.3.1 Reunião

Sejam $G_1 = (T, N_1, P_1, S_1)$ e $G_2 = (T, N_2, P_2, S_2)$ duas gramáticas livres de contexto que geram as linguagens L_1 e L_2, respetivamente. A gramática $G = (T, N, P, S)$, onde

$S \notin (T \cup N_1 \cup N_2)$;
$N = N_1 \cup N_2 \cup \{S\}$
$P = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}$
gera a linguagem $L = L_1 \cup L_2$.

Comp2025

1.3.2 Concatenação

Sejam $G_1 = (T, N_1, P_1, S_1)$ e $G_2 = (T, N_2, P_2, S_2)$ duas gramáticas livres de contexto que geram as linguagens L_1 e L_2, respetivamente. A gramática $G = (T, N, P, S)$, onde

$S \notin (T \cup N_1 \cup N_2)$;
$N = N_1 \cup N_2 \cup \{S\}$
$P = P_1 \cup P_2 \cup \{S \to S_1 \ S_2\}$
gera a linguagem $L = L_1 \cdot L_2$.

1.3.3 Fecho de Kleene

Seja $G_1 = (T, N_1, P_1, S_1)$ uma gramática livre de contexto que gera a linguagem L_1. A gramática $G = (T, N, P, S)$, onde

$S \notin (T \cup N_1)$;
$N = N_1 \cup \{S\}$
$P = P_1 \cup \{S \to \varepsilon, S \to S_1 \ S\}$
gera a linguagem $L = {L_1}^*$.

1.3.4 Intersecção e complementação

Se L_1 e L_2 são duas linguagens livres de contexto, e, por conseguinte, descritíveis por gramáticas livres de contexto, a sua intersecção pode não o ser. Já se disse atrás que a linguagem

$$L = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \geq 0\}$$

não é livre de contexto. No entanto, ela pode ser obtida por intersecção das linguagens $L_1 = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^j \mid i, j \geq 0\}$ e $L_2 = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^j \mid i, j \geq 0\}$ que o são.

Sabe-se que $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Então, se a intersecção de linguagens livres de contexto pode resultar numa linguagem que não o é, o mesmo pode acontecer com a complementação.

1.4 Árvore de derivação

Será que as várias derivações de uma mesma palavra são diferentes? Poderão ter interpretações diferentes? Veja-se com um exemplo que de facto podem. Considere a gramática

Comp 2023

$S \rightarrow S$ `+` $S \mid S$ `.` $S \mid \neg S \mid$ `(` S `)` $\mid$ `0` $\mid$ `1`

e compare as duas derivações esquerdas seguintes da palavra `1+1.0`.

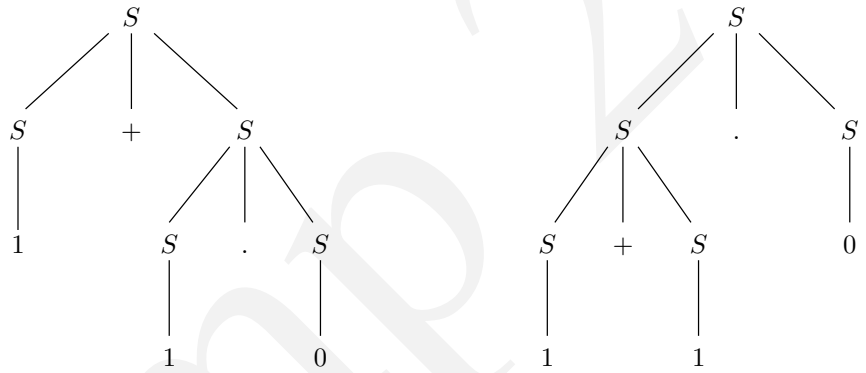
Derivação 1:

$$\underline{S} \Rightarrow \underline{S}+S \Rightarrow \texttt{1}+\underline{S} \Rightarrow \texttt{1}+\underline{S}.S \Rightarrow \texttt{1}+\texttt{1}.\underline{S} \Rightarrow \texttt{1}+\texttt{1}.\texttt{0}$$

Derivação 2:

$$\underline{S} \Rightarrow \underline{S}.S \Rightarrow \underline{S}+S.S \Rightarrow \texttt{1}+\underline{S}.S \Rightarrow \texttt{1}+\texttt{1}.\underline{S} \Rightarrow \texttt{1}+\texttt{1}.\texttt{0}$$

Serão estas derivações equivalentes? Para responder a esta pergunta vão-se representar as derivações usando um outro formalismo. A **árvore de derivação** (*parse tree*) é um mecanismo de representação de uma derivação que capta a interpretação dada nessa derivação. Veja-se as duas derivações anteriores representadas de forma arbórea.



A árvore da esquerda corresponde à derivação 1 e a da direita à 2. Vê-se claramente que as duas árvores têm interpretações distintas: a da esquerda é equivalente a ter-se `1+(1.0)`, enquanto que a direita é equivalente a `(1+1).0`. Em termos de álgebra booleana as duas expressões são bastante diferentes.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.4, "Parse trees and derivations".

1.4.1 Ambiguidade

Diz-se que uma palavra é derivada **ambiguamente** se possuir duas ou mais árvores de derivação distintas. Na gramática anterior a palavra `1+1.0` é gerada ambiguamente. Diz-se que uma gramática é **ambígua** se possuir pelo menos uma palavra gerada ambiguamente. A gramática anterior é ambígua.

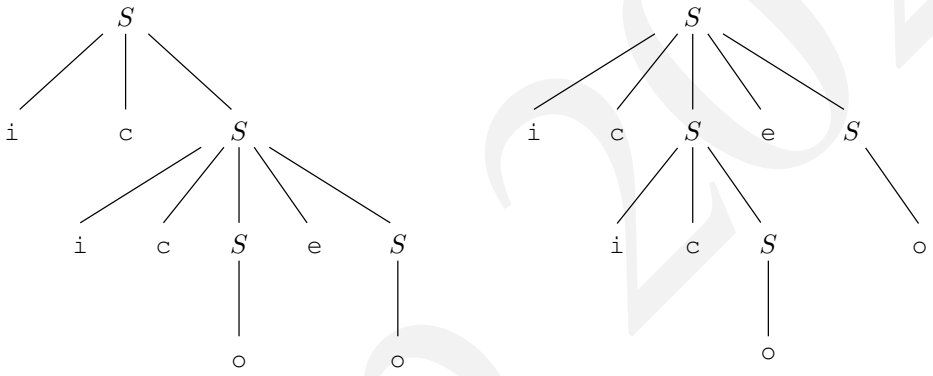
Frequentemente é possível definir-se uma gramática não ambígua que gere a mesma linguagem que uma ambígua. A gramática anterior pode ser rescrita por

$$
\begin{array}{lcl}
S & \rightarrow & T \mid S \texttt{ + } T \\
T & \rightarrow & F \mid T \texttt{ . } F \\
F & \rightarrow & K \mid \neg\ K \\
K & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{(}\ S\ \texttt{)}
\end{array}
$$

que não é ambígua e gera exatamente a mesma linguagem. A gramática

$$S \rightarrow \texttt{o} \mid \texttt{i c}\ S \mid \texttt{i c}\ S\ \texttt{e}\ S$$

é uma abstração da instrução `if-then-else` e possui ambiguidade. A palavra `icicoeo` tem duas árvores de derivação possíveis, representadas abaixo.



A árvore da esquerda corresponde à interpretação dada na linguagem C, em que o `e` (*else*) está associado ao `i` (*if*) mais à direita. A árvore da direita associa o `e` ao `i` mais à esquerda.

É possível por manipulação gramatical obter-se uma gramática equivalente sem ambiguidade. A gramática seguinte não é ambígua e descreve a mesma linguagem que a anterior.

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{o} \mid \texttt{i c}\ S \mid \texttt{i c}\ S'\ \texttt{e}\ S \\
S' & \rightarrow & \texttt{o} \mid \texttt{i c}\ S'\texttt{e}\ S'
\end{array}
$$

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.5, "Ambiguity".

Linguagens inerentemente ambíguas

Há linguagens **inerentemente ambíguas**, no sentido em que é impossível definir-se uma gramática não ambígua que gere essa linguagem. É, por exemplo, o caso da linguagem

$$L = \{\texttt{a}^i\texttt{b}^j\texttt{c}^k \mid i = j \vee j = k\}$$

1.5 Limpeza de gramáticas

1.5.1 Símbolos produtivos e não produtivos

Seja $G = (T, N, P, S)$ uma gramática qualquer. Um símbolo não terminal A diz-se **produtivo** se for possível transformá-lo num expressão contendo apenas símbolos terminais. Ou seja, A é produtivo se

$$A \Rightarrow^* u \quad \wedge \quad u \in T^*$$

Caso contrário, diz-se que A é **improdutivo**. Uma gramática é improdutiva se o seu símbolo inicial for improdutivo.

Sobre o alfabeto $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, considere a gramática

$$\begin{array}{lcl} S & \rightarrow & \mathtt{a\ b} \mid \mathtt{a}\ S\ \mathtt{b} \mid X \\ X & \rightarrow & \mathtt{c}\ X \end{array}$$

S é produtivo, porque

$$S \Rightarrow \mathtt{ab} \quad \wedge \quad \mathtt{ab} \in T^*$$

Em contrapartida, X é improdutivo.

$$X \Rightarrow \mathtt{c}X \Rightarrow \mathtt{cc}X \Rightarrow^* \mathtt{c}\cdots\mathtt{c}X$$

Por mais que se tente é impossível transformar X numa sequência de símbolos terminais.

Seja $G = (T, N, P, S)$ uma gramática qualquer. O conjunto dos seus símbolos produtivos, N_p, pode ser obtido por aplicação das seguintes regras construtivas

if $(A \rightarrow \alpha) \in P$ **and** $\alpha \in T^*$ **then** $A \in N_p$
if $(A \rightarrow \alpha) \in P$ **and** $\alpha \in (T \cup N_p)^*$ **then** $A \in N_P$

A 1ª regra é um caso particular da 2ª, pelo que poderia ser retirada. Optou-se por a incluir porque torna a leitura mais fácil.

Começando com um N_p igual ao resultado da aplicação da 1ª regra a todas as produções da gramática e extendendo depois esse conjunto por aplicação sucessiva da 2ª regra obtem-se o conjunto de todos os símbolos produtivos de G. O algoritmo seguinte executa esse procedimento.

Algoritmo 1.1 (Cálculo dos símbolos produtivos)

let $N_p = \emptyset$, $P_p = P$
repeat
    nothingAdded = TRUE
    foreach $(A \rightarrow \alpha) \in P_p$ **do**

```
            if α ∈ (T ∪ N_p)* then
                if A ∉ N_p then
                    N_p = N_p ∪ {A}
                    nothingAdded = FALSE
                P_p = P_p \ {A → α}
    until nothingAdded or N_p = N
```

Nele, N_p representa o conjunto dos símbolos produtivos já identificados e P_p o conjunto das produções contendo símbolos ainda não identificados como produtivos. Se numa iteração nenhum símbolo for marcado como produtivo o algoritmo pára, sendo o conjunto dos símbolos produtivos o conjunto N_p tido nesse momento. Obviamente que o algoritmo também pára, se no fim de uma iteração, $N_p = N$, isto é, se todos os símbolos foram marcados como produtivos.

1.5.2 Símbolos acessíveis e não acessíveis

Seja $G = (T, N, P, S)$ uma gramática qualquer. Um símbolo terminal ou não terminal x diz-se **acessível** se for possível transformar S (o símbolo inicial) numa expressão que contenha x. Ou seja,

$$S \Rightarrow^* \alpha x \beta$$

Caso contrário, diz-se que x é **inacessível**.

Considere a gramática

$$\begin{array}{lcl} S & \rightarrow & \varepsilon \mid \texttt{a}\ S\ \texttt{b} \mid \texttt{c}\ C\ \texttt{c} \\ C & \rightarrow & \texttt{c}\ S\ \texttt{c} \\ D & \rightarrow & \texttt{d}\ X\ \texttt{d} \\ X & \rightarrow & C\ C \end{array}$$

É impossível transformar S numa expressão que contenha D, d, ou X, pelo que estes símbolos são inacessíveis. Os restantes são acessíveis.

Seja $G = (T, N, P, S)$ uma gramática qualquer. O conjunto dos seus símbolos acessíveis, V_A, pode ser obtido por aplicação das seguintes regras construtivas

$S \in V_A$
if $A \rightarrow \alpha B \beta \in P$ **and** $A \in V_A$ **then** $B \in V_A$

Começando com $V_A = \{S\}$ e aplicando sucessivamente a 2ª regra até que ela não acrescente nada a V_A obtém-se o conjunto dos símbolos acessíveis. O algoritmo seguinte executa esse procedimento. Nele, V_A representa o conjunto dos símbolos acessíveis já identificados e N_X o conjunto dos símbolos não terminais acessíveis já identificados mas ainda não processados. No fim, quando N_X for o conjunto vazio, V_A contém todos os símbolos acessíveis.

Algoritmo 1.2 (Cálculo dos símbolos acessíveis)

```
let V_A = {S},  N_X = V_A
repeat
    let A = um elemento de N_X
    N_X = N_X \ {A}
    foreach (A → α) ∈ P do
        foreach x in α do
            if x ∉ V_A then
                V_A = V_A ∪ {A}
                if x ∈ N then
                    N_X = N_X ∪ {A}
until N_X = ∅
```

1.5.3 Gramáticas limpas

Numa gramática os símbolos inacessíveis e os símbolos improdutivos são **símbolos inúteis**, porque não contribuem para as palavras que a gramática pode gerar. Se tais símbolos forem removidos obtem-se uma gramática equivalente, em termos da linguagem que descreve. Diz-se que uma gramática é **limpa** se não possuir símbolos inúteis.

Para limpar uma gramática deve-se começar por a expurgar dos símbolos improdutivos. Só depois se devem remover os inacessíveis.

1.6 Transformações em gramáticas livres de contexto

Em muitas situações práticas — algumas serão abordadas nos capítulos seguintes — é necessário transformar uma gramática numa outra que seja equivalente e goze de determinada propriedade. Apresentam-se a seguir algumas dessas transformações.

1.6.1 Eliminação de produções-ε

Uma **produção-ε** é uma produção do tipo $A \to \varepsilon$, para um qualquer símbolo não terminal A. Se L é uma linguagem livre de contexto tal que $\varepsilon \notin L$, é possível descrever L por uma gramática livre de contexto sem produções-ε. Se assim é então tem de ser possível transformar uma gramática que descreva uma linguagem L e possua produções-ε numa outra equivalente que as não possua.

Considere a gramática

$$\begin{array}{lcl} I & \to & \texttt{0}\ I \mid \texttt{1}\ P \\ P & \to & \varepsilon \mid \texttt{0}\ P \mid \texttt{1}\ I \end{array}$$

Comp 2023

que descreve a linguagem L formada pelas palavras definidas sobre o alfabeto $\{0,1\}$, com número ímpar de $1s$. Claramente, a palavra vazia não pertence a L porque não tem número ímpar de uns. Mas, a gramática contém a produção $P \to \varepsilon$. Então, de acordo com o dito anteriormente, existe uma gramática equivalente que não tem produções-ε.

A existência de tal produção na gramática anterior significa que as produções $I \to 1P$ e $P \to 0P$ podem produzir as derivações $I \Rightarrow 1$ e $P \Rightarrow 0$, respetivamente. Mas, estas derivações podem ser contempladas acrescentando as produções $I \to 1$ e $P \to 0$ à gramática, tornando desnecessária a produção-ε. Na realidade a gramática

$$\begin{array}{lcl} I & \to & 0\ I \mid 1\ P \mid 1 \\ P & \to & 0\ P \mid 0 \mid 1\ I \end{array}$$

é equivalente à anterior e não possui produções-ε.

Em geral, o papel da produção $A \to \varepsilon$ sobre uma produção $B \to \alpha A \beta$ pode ser representado pela inclusão da produção $B \to \alpha\beta$. Assim a eliminação das produções-ε de uma gramática pode ser obtido por aplicação do algoritmo seguinte

Algoritmo 1.3 (Eliminação de produções-ε, 1ª versão)

```
foreach A → ε do
    foreach B → αAβ do
        add B → αβ to P.
    remove A → ε from P.
```

O algoritmo anterior pode introduzir novas produções-ε na gramática. Se $B \to A$ for uma produção da gramática, a eliminação da produção $A \to \varepsilon$ introduz a produção $B \to \varepsilon$. A algoritmo deve ser alterado de modo a acautelar estas situações.

...

1.6.2 Eliminação de recursividade à esquerda

Diz-se que uma gramática é **recursiva à esquerda** se possuir um símbolo não terminal A que admita uma derivação do tipo $A \Rightarrow^+ A\gamma$, ou seja, que seja possível, em um ou mais passos de derivação, transformar A numa expressão que tem A no início.

A gramática seguinte é recursiva à esquerda

$$\begin{array}{lcl} E & \to & X\ T \\ X & \to & \varepsilon \mid E\ \texttt{+} \\ T & \to & \texttt{a} \mid \texttt{b} \mid \texttt{(}\ E\ \texttt{)} \end{array}$$

A derivação $E \Rightarrow X\ T \Rightarrow E\ \texttt{+}\ T$ mostra que é possível transformar E numa expressão com E à esquerda. Logo, esta gramática tem recursividade à esquerda associada ao símbolo não terminal E.

Se a obtenção da expressão começada por A se faz em apenas um passo de derivação, então diz-se que a recursividade é **imediata**. Esta última situação só ocorre se a gramática possuir uma ou mais produções do tipo $A \rightarrow A\ \alpha$.

No gramática seguinte a recursividade à esquerda é imediata.

$$\begin{array}{lcl} E & \rightarrow & T \mid E\ \texttt{+}\ T \\ T & \rightarrow & \texttt{a} \mid \texttt{b} \mid \texttt{(}\ E\ \texttt{)} \end{array}$$

A eliminação de recursividade imediata à esquerda fazer-se com um algoritmo bastante simples. Considere que $A \rightarrow \beta$ e $A \rightarrow A\ \alpha$, onde A é um símbolo não terminal e α e β sequências de zero ou mais símbolos terminais ou não terminais, são duas produções de uma gramática qualquer. Será possível substituir as duas produções por outras que não possuam recursividade à esquerda e produzam uma gramática equivalente? Para responder a esta pergunta observem-se as palavras que se podem obter a partir de A. Numa derivação de um passo obtem-se $A \Rightarrow \beta$. Numa de dois passos obtem-se $A \Rightarrow A\alpha \Rightarrow \beta\alpha$. Numa de n passos, com $n > 0$, obtem-se $A \Rightarrow \beta\alpha^{n-1}$. Mas estas palavras também podem ser obtidas com as produções

$$\begin{array}{lcl} A & \rightarrow & \beta\ X \\ X & \rightarrow & \varepsilon \mid \alpha\ X \end{array}$$

que não possui recursividade à esquerda. A nova gramática continua a ser recursiva. Na realidade, não pode deixar de o ser, a recursividade passou para à direita.

O algoritmo anterior pode ser facilmente adaptado a situações em que possa haver mais do que uma produção a introduzir a recursividade imediata à esquerda. Considere que

$$A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \beta_m \mid A\ \alpha_1 \mid A\ \alpha_2 \mid \cdots \mid A\ \alpha_n$$

são as produções de uma gramática com A à cabeça. As palavras que se podem gerar com estas produções são as mesmas que se podem gerar com as produções seguintes e que não possuem recursividade à esquerda.

$$\begin{array}{lcl} A & \rightarrow & \beta_1\ X \mid \beta_2\ X \mid \cdots \beta_m\ X \\ X & \rightarrow & \varepsilon \mid \alpha_1\ X \mid \alpha_2\ X \mid \cdots \mid \alpha_n\ X \end{array}$$

Se a recursividade à esquerda não é imediata o algoritmo de eliminação é um pouco mais complexo.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.3.3, "Elimination of left recursion".

1.6.3 Fatorização à esquerda

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.3.4, "Left factoring".

1.7 Os conjuntos `first`, `follow` e `predict`

A construção de reconhecedores sintáticos (*parsers*) — apresentados nos capítulos seguintes — é auxiliada por três funções associadas às gramáticas. Estas funções são os conjuntos **`first`**, **`follow`** e **`predict`**.

1.7.1 O conjunto `first`

Seja $G = (T, N, P, S)$ uma gramática qualquer e α uma sequência de símbolos terminais e não terminais, isto é, $\alpha \in (T \cup N)^*$. O conjunto **`first`**(α) contém todos os símbolos terminais que podem aparecer como primeira letra de sequências obtidas a partir de α por aplicação de produções da gramática. Ou seja, um símbolo terminal x pertence a **`first`**(α) se e só se $\alpha \Rightarrow^* x\beta$, com β qualquer. Adicionalmente, considera-se que ε pertence ao conjunto **`first`**(α) se $\alpha \Rightarrow^* \varepsilon$.

Algoritmo 1.4 (Conjunto `first`)

`first`$(\alpha)\,\{$
  `if` $(\alpha = \varepsilon)$ ***`then`***
    `return` $\{\varepsilon\}$
  $h =$ ***`head`*** (α) *# com* $|h| = 1$
  $\omega =$ ***`tail`*** (α) *# tal que* $\alpha = h\,\omega$
  `if` $(h \in T)$ ***`then`***
    `return` $\{h\}$
  `else`
    `return` $\bigcup_{(h \to \beta_i) \in P}$ ***`first`***$(\beta_i\,\omega)$
  $\}$

Note que no último **`return`** o argumento do **`first`** é $\beta_i\,\omega$, concatenação dos β_i (que vêm dos corpos das produções começadas por h) com o ω (que é o **`tail`** do α

1.7.2 O conjunto `follow`

O conjunto **`follow`** está relacionado com os símbolos não terminais de uma gramática. Seja $G = (T, N, P, S)$ uma gramática e A um elemento de N ($A \in N$). O conjunto **`follow`**(A) contém todos os símbolos terminais que podem aparecer imediatamente à direita de A num processo derivativo qualquer. Formalmente, **`follow`**$(A) = \{a \in T \mid S \Rightarrow^* \gamma A a \psi\}$, com α e β quaisquer.

O cálculo dos conjuntos **`follow`** dos símbolos não terminais da gramática $G = (T, N, P, S)$ baseia-se na aplicação das 4 regras seguintes, onde $\supseteq$ significa "contém".

Comp 2023

1. $\$ \in \texttt{follow}(S)$.
2. se $(A \to \alpha\ B) \in P$, então $\texttt{follow}(B) \supseteq \texttt{follow}(A)$.
3. se $(A \to \alpha\ B\ \beta) \in P$ e $\varepsilon \notin \texttt{first}(\beta)$, então $\texttt{follow}(B) \supseteq \texttt{first}(\beta)$.
4. se $(A \to \alpha B \beta) \in P$ e $\varepsilon \in \texttt{first}(\beta)$, então $\texttt{follow}(B) \supseteq ((\texttt{first}(\beta) - \{\varepsilon\}) \cup \texttt{follow}(A))$.

A primeira regra é óbvia. Sendo o símbolo inicial da gramática, S representa as palavras da linguagem. Logo \$ vem a seguir.

A segunda regra diz que se $A \to \alpha\ B$ é uma produção da gramática e $x \in \texttt{follow}(A)$, então $x \in \texttt{follow}(B)$. Considere, por hipótese, que $x \in \texttt{follow}(A)$. Então, pela definição de conjunto **follow**, $S \Rightarrow^* \gamma A x \psi$. Logo, sendo $A \to \alpha B$ uma produção da gramática, tem-se que $S \Rightarrow^* \gamma \alpha B x \psi$, ou seja, $x \in \texttt{follow}(B)$.

A terceira regra diz que se $A \to \alpha\ B\ \beta$ é uma produção da gramática, com $\varepsilon \notin \texttt{first}(\beta)$, e $x \in \texttt{first}(\beta)$, então $x \in \texttt{follow}(B)$. Considere, por hipótese, que $x \in \texttt{first}(\beta)$. Então, pela definição de conjunto **first**, $\beta \Rightarrow^* x\ \gamma$ e, consequentemente, $A \Rightarrow^* \alpha\ B\ x\ \gamma$, ou seja, $x \in \texttt{follow}(B)$.

Finalmente, a quarta e última regra diz que se $A \to \alpha\ B\ \beta$ é uma produção da gramática, com $\varepsilon \in \texttt{first}(\beta)$, e $x \in (\texttt{first}(\beta) - \{\varepsilon\}) \cup \texttt{follow}(A)$, então $x \in \texttt{follow}(B)$. Esta regra pode ser entendida cruzando as duas regras anteriores. Considere-se os elementos de $\texttt{first}(\beta)$ diferentes de ε. Pela regra 3, pertencem ao $\texttt{follow}(B)$. Se β se transforma em ε, então $A \Rightarrow^* \alpha\ B$ e, pela regra 2, se $x \in \texttt{follow}(A)$, então $x \in \texttt{follow}(B)$.

1.7.3 O conjunto `predict`

O conjunto **predict** aplica-se às produções de uma gramática e envolve os conjuntos **first** e **follow**. É dado pela seguinte equação.

$$\texttt{predict}(A \to \alpha) = \begin{cases} \texttt{first}(\alpha) & \varepsilon \notin \texttt{first}(\alpha) \\ (\texttt{first}(\alpha) - \{\varepsilon\}) \cup \texttt{follow}(A) & \varepsilon \in \texttt{first}(\alpha) \end{cases}$$

Capítulo 2

Análise sintática descendente

NOTA PRÉVIA: *Este capítulo não está completo e não foi devidamente revisto, pelo que, por um lado, há partes omissas e, por outro lado, pode conter falhas.*

Dada uma gramática $G = (T, N, P, S)$ e dada uma palavra $u \in T^*$, $u \in L(G)$ se e só se existir uma derivação que produza u a partir de S, isto é, se $S \Rightarrow^* u$. Um **reconhecedor sintático** da gramática G é um mecanismo que responde à pergunta "$u \in L(G)$?", tentando produzir a derivação anterior. Para o fazer, o reconhecedor pode partir de S e tentar chegar a u ou partir de u e tentar chegar a S. No primeiro caso diz-se que o reconhecedor é **descendente**, porque o seu procedimento corresponde à geração da árvore de derivação da palavra u de cima (raiz) para baixo (folhas).

O papel da análise sintática é definir procedimentos que permitam contruir reconhecedores sintáticos a partir da gramática. Por exemplo, considere a linguagem L descrita pela gramática G seguinte.

$$S \rightarrow \texttt{a}\ S\ \texttt{b} \mid \texttt{c}\ S \mid \varepsilon$$

Será que a palavra $\texttt{acacbb} \in L$? Pertencerá se $S \Rightarrow^* \texttt{acacbb}$. Na verdade pertence porque

$$S \Rightarrow \texttt{a}S\texttt{b} \Rightarrow \texttt{ac}S\texttt{b} \Rightarrow \texttt{aca}S\texttt{bb} \Rightarrow \texttt{acac}S\texttt{bb} \Rightarrow \texttt{acacbb}$$

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4, "Top-down parsing".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.3, "LL(1) grammars".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.1, "Predictive parsing".

Comp 2023

2.1 Reconhecimento preditivo

Mas como deterministicamente produzir a derivação anterior se houver duas ou mais produções com o mesmo símbolo à cabeça? Por exemplo, considerando o caso anterior, como saber que se deve optar por expandir o S usando $S \rightarrow \texttt{a}\ S\ \texttt{b}$ e não $S \rightarrow \texttt{c}\ S$ ou $S \rightarrow \varepsilon$? A solução baseia-se na observação antecipada dos próximos símbolos à entrada (*lookahead*). No exemplo, se o próximo símbolo à entrada for um `a` expande-se usando a produção $S \rightarrow \texttt{a}\,S\,\texttt{b}$; se for um `c` usa-se $S \rightarrow \texttt{c}\,S$; se for `b` usa-se $S \rightarrow \varepsilon$. Se a entrada se esgotou, o que é representado considerando que a próxima entrada é um \$, também se expande usando a produção $S \rightarrow \varepsilon$.

Pode-se então definir uma tabela que para cada símbolo não terminal da gramática e para cada valor do *lookahead* indica qual a produção da gramática que deve ser usada na expansão. O profundidade da observação antecipada (número de símbolos de *lookahead*) pode ser qualquer, embora apenas a profundidade 1 será usada neste documento. Para o exemplo anterior a tabela assume a forma

	lookahead			
symbol	`a`	`b`	`c`	\$
S	$S \rightarrow \texttt{a}\,S\,\texttt{b}$	$S \rightarrow \varepsilon$	$S \rightarrow \texttt{c}\,S$	$S \rightarrow \varepsilon$

Na tabela anterior, o preenchimento das colunas `a` e `c` são óbvias, visto que as produções associadas começam pelo próprio símbolo do *lookahead*. Nas colunas `b` e \$ tal não acontece. O preenchimento da tabela de decisão baseia-se nos conjuntos **`predict`**, apresentados na secção 1.7, e faz-se usando o algoritmo seguinte:

Algoritmo 2.1 (Preenchimento da tabela de decisão para *lookahead* 1)

```
foreach (A → α) ∈ P
    foreach a ∈ predict(A → α)
        add (A → α) to T[A, a]
```

As células da tabela que fiquem vazias representam situações de erro sintático. As células da tabela que fiquem com dois os mais produções representam situações de não determinismo: com base no *lookahead* usado não é possível escolher que produção usar na expansão. Uma gramática diz-se **LL(1)** se na tabela de decisão, para um *lookahead* de profundidade 1, não houver células com mais que uma produção. Equivalentemente, uma gramática diz-se LL(1) se para todas as produções com o mesmo símbolo à cabeça os seus conjuntos **`predict`** são dijuntos entre si.

Exemplo 2.1

Calcule a tabela de decisão de um reconhecedor preditivo para a gramática seguinte.

$$
\begin{array}{lcl}
S & \rightarrow & A\ B \\
A & \rightarrow & \varepsilon \mid \mathtt{a}\ A \\
B & \rightarrow & \varepsilon \mid \mathtt{b}\ B
\end{array}
$$

Resposta:

(Deixo ao cuidado do leitor o cálculo dos conjuntos **`predict`**.)

$$
\begin{array}{l}
\texttt{predict}(S \rightarrow A\ B) = \{\mathtt{a}, \mathtt{b}, \$\} \\
\texttt{predict}(A \rightarrow \varepsilon) = \{\mathtt{b}, \$\} \\
\texttt{predict}(A \rightarrow \mathtt{a}A) = \{\mathtt{a}\} \qquad l \\
\texttt{predict}(B \rightarrow \varepsilon) = \{\$\} \\
\texttt{predict}(B \rightarrow \mathtt{b}B) = \{\mathtt{b}\}
\end{array}
$$

	lookahead		
symbol	a	b	$
S	$S \rightarrow A\ B$	$S \rightarrow A\ B$	$S \rightarrow A\ B$
A	$A \rightarrow \mathtt{a}\ A$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B		$B \rightarrow \mathtt{b}\ B$	$B \rightarrow \varepsilon$

2.2 Reconhecedores recursivo-descendentes

O reconhecimento preditivo, sintetizado na tabela de decisão apresentada acima, permite construir programas de reconhecimento, reconhecedores (ou *parsers* na terminologia em inglês). Uma solução para a construção do reconhecedor é baseada numa estrutura na qual cada símbolo não terminal da gramática dá origem a uma função, possivelmente recursiva.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 2.4.2, "Recursive-descent parsing".

Comp 2023

Exemplo 2.2

Considere que com base na gramática e na tabela de decisão do exemplo anterior se contrói o programa seguinte.

```
function eat(token)
    if lookahead = token then
        lookahead := nextToken().
    else
        REJECT.

function S()
    A(), B().

function A()
    case lookahead in
        a :  eat(a), A(), return .
        b, $ :  return .

function B()
    case lookahead in
        a :  REJECT.
        b :  eat(b), B(), return .
        $ :  return .

program Parser()
    lookahead := nextToken().  S().
    if lookahead = $ then
        ACCEPT.
    else
        REJECT.
```

Se executar o programa `Parser` quando a entrada é `aab` verificará que após o retorno da função S o *lookahead* é igual a \$, indicando que a palavra é válida. Mas a palavra aba é rejeitada, porque durante a execução a função B vai ser invocada numa altura em que o *lookahead* é igual a `a`. (Confirme.)

2.3 Reconhecedores descendentes não recursivos

Uma solução alternativa para implementar o reconhecedor preditivo usa uma pilha (*stack*) para reter o estado no processo de reconhecimento e usa a tabela de decisão para decidir como evoluir no processo de reconhecimento.

Seja $G = (T, N, P, S)$ uma gramática independente do contexto, que se assume seja LL(1). Seja M a tabela de decisão de G para um *lookahead* de profundidade 1. Finalmente, considere que dispõe de uma

Comp 2023

pilha onde pode armazenar elementos do conjunto $Z = T \cup N \cup \{\$\}$ e que pode ser manipulada com as funções **push**, **pop** e **top**, que, respetivamente, coloca uma sequência de símbolos na pilha, retira o símbolo no topo da pilha e mostra qual o símbolo no topo sem o retirar. O programa seguinte é um reconhecedor das palavras da gramática G

```
program Parser()
    push(S $).
    lookahead = getToken().
    forever
        z := top().
        if z ∈ T then
            if z ≠ lookahead then
                REJECT.
            elseif z = $ then
                ACCEPT.
            else (* z = lookahead ∧ z ≠ $ *)
                pop(), lookahead = getToken().
        else (* z ∈ N *)
            α := M(z, lookahead).
            if α = ∅ then
                REJECT.
            else
                pop(), push(α).
```

A evolução do programa anterior no processo de reconhecimento pode ser captado por uma tabela onde se mostre a cada passo os estados da pilha e da entrada e a ação tomada. Se se considerar a gramática e a tabela de decisão do exemplo 2.1, a execução do programa anterior sobre a palavra `aab` resulta na seguinte tabela. Na coluna da pilha, o símbolo mais à direita é o que está no topo e, na coluna entrada, o símbolo mais à esquerda é o *lookahead*.

Comp 2023

Pilha	Entrada	Ação
$ S	a a b $	**pop**(), **push**(A B)
$ B A	a a b $	**pop**(), **push**(a A)
$ B A a	a a b $	**pop**(), lookahead = getToken()
$ B A	a b $	**pop**(), **push**(a A)
$ B A a	a b $	**pop**(), lookahead = getToken()
$ B A	b $	**pop**()
$ B	b $	**pop**(), **push**(b B)
$ B b	b $	**pop**(), lookahead = getToken()
$ B	$	**pop**()
$	$	ACCEPT

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.4, "Nonrecursive predictive parsing".

2.4 Implementação de gramáticas de atributos

Os reconhecedores recursivo-descendentes (ver secção 2.2) permitem implementar facilmente gramáticas de atributos do tipo L (ver secção 4.2.2). As funções recursivas podem ter parâmetros de entrada e de saída. O valor de retorno pode também funcionar como parâmetro de saída. Os primeiros permitem passar informação da função chamadora para a função chamada, o que corresponde, na árvore de derivação, a definir atributos herdados dos nós filhos com base em atributos do nó pai. Os segundos permitem passar informação da função chamada para a função chamadora, o que corresponde ao suporte de atributos sintetizados e de atributos herdados de nós filhos com base em atributos de nós filhos à esquerda na árvore de derivação.

Comp 2023

Capítulo 3

Análise sintáctica ascendente

NOTA PRÉVIA: *Este capítulo não está completo e não foi devidamente revisto, pelo que, por um lado, há partes omissas e, por outro lado, pode/deve conter falhas.*

Considere a gramática

$$
\begin{array}{lcl}
D & \rightarrow & T\ L\ \texttt{;} \\
T & \rightarrow & \texttt{i} \mid \texttt{r} \\
L & \rightarrow & \texttt{v} \mid L\ \texttt{,}\ \texttt{v}
\end{array}
$$

que representa uma declaração de variáveis *a la* C. Como reconhecer a palavra "$u = \texttt{i v , v ;}$" como pertencente à linguagem gerada pela gramática dada? Se u pertence à linguagem gerada pela gramática, então $D \Rightarrow^* u$. Tente-se chegar lá andando no sentido contrário ao de uma derivação, ie. de u para D.

$$
\begin{array}{lll}
 & \texttt{i v , v ;} & \\
\Leftarrow & T\ \texttt{v , v ;} & \text{(por aplicação da regra } T \rightarrow \texttt{i}) \\
\Leftarrow & T\ L\ \texttt{, v ;} & \text{(por aplicação da regra } L \rightarrow \texttt{v}) \\
\Leftarrow & T\ L\ \texttt{;} & \text{(por aplicação da regra } L \rightarrow L\ \texttt{, v}) \\
\Leftarrow & D & \text{(por aplicação da regra } D \rightarrow T\ L\ \texttt{;})
\end{array}
$$

Colocando ao contrário

$$D \Rightarrow T\ L\ \texttt{;} \Rightarrow T\ L\ \texttt{, v ;} \Rightarrow T\ \texttt{v , v ;} \Rightarrow \texttt{i v , v ;}$$

vê-se que corresponde a uma derivação à direita. A tabela seguinte mostra como, na prática, se realiza esta (retro)derivação.

Comp 2023

pilha	entrada	ação
$	`i v , v ; $`	deslocamento
$ `i`	`v , v ; $`	redução por $T \rightarrow$ `i`
$ T	`v , v ; $`	deslocamento
$ T `v`	`, v ; $`	redução por $L \rightarrow$ `v`
$ T L	`, v ; $`	deslocamento
$ T L `,`	`v ; $`	deslocamento
$ T L `, v`	`; $`	redução por $L \rightarrow L$ `, v`
$ T L	`; $`	deslocamento
$ T L `;`	$	redução por $D \rightarrow T\ L$ `;`
$ D	$	aceitação

Inicialmente, o topo da pilha apenas possui um símbolo especial, representado por um $, que indica, quando no topo, que a pilha está vazia. A entrada possui a palavra a reconhecer seguida também de um símbolo especial, aqui também representado por um $, que indica fim da entrada. Em cada ciclo realiza-se, normalmente, uma operação de deslocamento ou de redução. A operação de **deslocamento** (no inglês, *shift*) transfere o símbolo não terminal da entrada para o topo da pilha. A operação de **redução** (no inglês, *reduce*) substitui os símbolos do topo da pilha que correspondem ao corpo de uma produção da gramática pela cabeça dessa regra.

Se se atingir uma situação em que a entrada apenas possui o símbolo $ e a pilha apenas possui o símbolo $ e o símbolo inicial da gramática, tal como acontece na tabela anterior, a palavra é reconhecida como pertencendo à linguagem descrita pela gramática. Caso contrário, a palavra é rejeitada.

Veja-se a reação deste procedimento a uma entrada errada, por exemplo a palavra `i v v ;`.

pilha	entrada	ação
$	`i v v ; $`	deslocamento
$ `i`	`v v ; $`	redução por $T \rightarrow$ `i`
$ T	`v v ; $`	deslocamento
$ T `v`	`v ; $`	rejeição

Com T `v` na pilha e `v` na entrada é impossível chegar-se à aceitação. Porque se se reduzir `v` para L ficar-se-ia com um $T\ L$ na pilha e v na entrada, que não pertence ao conjunto **`follow`**(L). Mais à frente voltaremos a este assunto.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.1, "Reductions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.2, "Handle pruning".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.3, "Shift-reduce parsing".

3.1 Conflitos

O procedimento acabado de descrever pode acarretar situações ambíguas chamadas **conflitos**. Considere a gramática

$$\begin{array}{rcl} S & \rightarrow & \texttt{i c } S \\ & | & \texttt{i c } S \texttt{ e } S \\ & | & \texttt{a} \end{array}$$

e execute o procedimento de reconhecimento com a palavra `i c i c a e a`. Obtém-se

pilha	entrada	ação
\$	i c i c a e a \$	deslocamento
\$ i	c i c a e a \$	deslocamento
\$ i c	i c a e a \$	deslocamento
\$ i c i	c a e a \$	deslocamento
\$ i c i c	a e a \$	deslocamento
\$ i c i c a	e a \$	redução por $S \rightarrow$ a
\$ i c i c S	e a \$	conflito: redução usando $S \rightarrow$ i c S
		ou deslocamento para tentar $S \rightarrow$ i c S e S ?

Na última linha é possível reduzir-se por aplicação da regra $S \rightarrow$ i c S ou deslocar-se o e para tentar posteriomente a redução pela regra $S \rightarrow$ i c S e S. Trata-se de um conflito deslocamento-redução (*shift-reduce conflict*). Perante este tipo de conflitos, ferramentas como o *bison* optam pelo deslocamento, mas pode não ser a mais adequada.

Também pode haver conflitos entre reduções (*reduce-reduce conflict*). Considere a gramática

$$\begin{array}{rcl} S & \rightarrow & A \\ & | & B \\ A & \rightarrow & \texttt{c} \\ & | & A \texttt{ a} \\ B & \rightarrow & \texttt{c} \\ & | & B \texttt{ b} \end{array}$$

e a palavra `c`. O procedimento de reconhecimento produz

pilha	entrada	ação
\$	c \$	deslocamento
\$ c	\$	conflito: redução usando $A \rightarrow$ c ou $B \rightarrow$ c ?

Na última linha é possível reduzir-se por aplicação das regras $A \to \texttt{c}$ ou $B \to \texttt{c}$. Perante este tipo de conflitos, ferramentas como o *bison* optam pela produção que aparece primeiro. Neste caso é irrelevante, mas pode não ser o adequado.

Veja-se agora a situação de um falso conflito. Considere a gramática

$$\begin{aligned} S &\to \texttt{a} \mid \texttt{(}\ S\ \texttt{)} \mid \texttt{a}\ P \mid \texttt{(}\ S\ \texttt{)}\ S \\ P &\to \texttt{(}\ S\ \texttt{)} \mid \texttt{(}\ S\ \texttt{)}\ S \end{aligned}$$

e reconheça-se a palavra "a(a)a".

pilha	entrada	ação
\$	a (a) a \$	deslocamento
\$ a	(a) a \$	redução usando $S \to \texttt{a}$
		deslocamento para tentar $S \to \texttt{a}\ P$?

Considerar a redução corresponde a realizar a retro-derivação

$$\texttt{a (a) a} \Leftarrow S\ \texttt{(a) a}$$

que não faz sentido porque $\texttt{(} \notin \texttt{follow}(S)$. Não há, portanto, conflito, sendo realizado o deslocamento.

pilha	entrada	ação
\$	a (a) a \$	deslocamento
\$ a	(a) a \$	deslocamento
\$ a (	a) a \$	deslocamento
\$ a (a	) a \$	redução por $S \to \texttt{a}$
\$ a (S	) a \$	deslocamento
\$ a (S)	a \$	deslocamento, porque $\texttt{a} \notin \texttt{follow}(S), \texttt{follow}(P)$
\$ a (S) a	\$	redução por $S \to \texttt{a}$
\$ a (S) S	\$	redução por $P \to \texttt{(}\ S\ \texttt{)}\ S$
\$ a P	\$	redução por $S \to \texttt{a}\ P$
\$ S	\$	aceitação

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.4, "Conflicts during shift-reduce parsing".

Pode-se alterar uma gramática de modo a eliminar a fonte de conflito. Considerando que se pretendia optar pelo deslocamento, a gramática seguinte gera a mesma linguagem e está isenta de conflitos.

$$\begin{aligned} S &\to \texttt{a} \\ &\mid \texttt{i c}\ S \\ &\mid \texttt{i c}\ S'\ \texttt{e}\ S \\ S' &\to \texttt{a} \\ &\mid \texttt{i c}\ S'\ \texttt{e}\ S' \end{aligned}$$

3.2 Construção de um reconhecedor ascendente

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.6, "Introduction to LR parsing: Simple LR".

O procedimento de reconhecimento apresentado atrás é um mecanismo iterativo. Em cada passo, a operação a realizar — deslocamento, redução, aceitação ou rejeição — depende da configuração nesse momento. Uma *configuração* é formada pelo conteúdo da pilha mais a parte da entrada ainda não processada. A pilha é conhecida — na realidade, é preenchida pelo procedimento de reconhecimento —, mas a entrada é desconhecida, conhecendo-se apenas o próximo símbolo (*lookahead*). Então a decisão a tomar só pode basear-se no conteúdo da pilha e no *lookahead*.

Mas, se se quiser construir um reconhecedor apenas com capacidade de observar o topo da pilha, uma pilha onde se guardam os símbolos terminais e não terminais tem pouco interesse. Deve-se guardar um símbolo que represente tudo o que está para trás.

Como definir esses símbolos?

A associação de um símbolo diferente por cada configuração da pilha não serve porque a pilha pode, em geral, crescer de forma não limitada. Os símbolos a colocar na pilha devem representar estados no processo de deslocamento-redução. O alfabeto da pilha representa assim o conjunto de estados nesse processo de reconhecimento.

Cada estado representa um conjunto de itens. O item de uma produção representa uma fase no processo de obtenção dessa produção. É representado por uma produção com um ponto ($\cdot$) numa posição do seu corpo. Por exemplo, a produção $A \rightarrow B_1\ B_2\ B_3$, produz 4 itens, a saber

$$\begin{aligned} A &\rightarrow \cdot B_1\ B_2\ B_3 \\ A &\rightarrow B_1 \cdot B_2\ B_3 \\ A &\rightarrow B_1\ B_2 \cdot B_3 \\ A &\rightarrow B_1\ B_2\ B_3 \cdot \end{aligned}$$

A produção $A \rightarrow \varepsilon$ produz um único item

$$A \rightarrow \cdot$$

Um item representa o quanto de uma produção já foi obtido e, simultaneamente, o quanto falta obter. Por exemplo, $A \rightarrow B_1 \cdot B_2\ B_3$, significa que já foi obtido algo corresponente a B_1, faltando obter o correspondente a $B_2\ B_3$. Se o ponto($\cdot$) se encontra à direita, então poder-se-á reduzir $B_1\ B_2\ B_3$ a A.

3.2.1 Construção da coleção de conjuntos de itens

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.6.2, "Items and the LR(0) automaton".

Considere a gramática

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & \texttt{a} \mid \texttt{(}\ E\ \texttt{)} \end{array}$$

O estado inicial (primeiro elemento da coleção de conjunto de itens) contém o item

$$Z_0 \;=\; \{S \rightarrow \cdot\, E\,\$\}$$

Este conjunto tem de ser fechado. O facto de o ponto ($\cdot$) se encontrar imediatamente à esquerda de um símbolo não terminal, significa que para se avançar no processo de reconhecimento é preciso obter esse símbolo. Isso é considerado juntando ao conjunto Z_0 os itens iniciais das produções cuja cabeça é E. Fazendo-o, Z_0 passa a

$$Z_0 \;=\; \{\, S \rightarrow \cdot\, E\,\$\,\} \cup \{\, E \rightarrow \cdot\texttt{a}\,,\; E \rightarrow \cdot\,\texttt{(}\ E\ \texttt{)}\,\}$$

Se nos novos elementos adicionados ao conjunto voltasse à acontecer de o ponto ($\cdot$) ficar imediatamente à esquerda de outros símbolos não terminais o processo deve ser repetido para esses símbolos.

O estado Z_0 pode evoluir por ocorrência de um E, um `a` ou um `(`. Correspondem aos símbolos que aparecem imediatamente à direita do ponto ($\cdot$), e produzem 3 novos estados

$$\begin{array}{lcl} Z_1 & = & \delta(Z_0, E) = \{\, S \rightarrow E \cdot \$\,\} \\ Z_2 & = & \delta(Z_0, \texttt{a}) = \{\, E \rightarrow \texttt{a}\cdot\,\} \\ Z_3 & = & \delta(Z_0, \texttt{(}) = \{\, E \rightarrow \texttt{(}\cdot\, E\ \texttt{)}\,\} \cup \{\, E \rightarrow \cdot\texttt{a}\,,\; E \rightarrow \cdot\,\texttt{(}\ E\ \texttt{)}\,\} \end{array}$$

Note que Z_3 foi estendido pela função de fecho, uma vez que o ponto ficou imediatamente à esquerda de um símbolo não terminal (E). Z_1 representa um situação de aceitação se o símbolo à entrada (*lookahead*) for igual a \$ e de erro caso contrário. Z_2 representa uma possível situação de redução pela regra $E \rightarrow \texttt{a}$. Esta redução só faz sentido se o símbolo à entrada (*lookahead*) for um elemento do conjunto **`follow`**(E). Caso contrário corresponde a uma situação de erro. Finalmente, Z_3 pode evoluir por ocorrência de um E, um `a` ou um `(`, que correspondem aos símbolos que aparecem imediatamente à direita do ponto ($\cdot$). Estas evoluções são indicadas a seguir

$$\begin{array}{rcl} Z_4 & = & \delta(Z_3, E) = \{\, E \rightarrow \texttt{(}\ E \cdot\ \texttt{)}\,\} \\ \delta(Z_3, \texttt{a}) & = & Z_2 \\ \delta(Z_3, \texttt{(}) & = & Z_3 \end{array}$$

Apenas um novo estado foi gerado (Z_4). Este estado apenas evolui por ocorrência de `)`.

$$Z_5 \;=\; \delta(Z_4, \texttt{)}) = \{\, E \rightarrow \texttt{(}\ E\ \texttt{)}\cdot\,\}$$

Pondo tudo agrupado, a coleção de conjunto de itens é

$$\begin{array}{lcl} Z_0 & = & \{\, S \rightarrow \cdot\, E\,\$\,\} \cup \{\, E \rightarrow \cdot\texttt{a}\,,\; E \rightarrow \cdot\,\texttt{(}\ E\ \texttt{)}\,\} \\ Z_1 & = & \delta(Z_0, E) = \{\, S \rightarrow E \cdot \$\,\} \\ Z_2 & = & \delta(Z_0, \texttt{a}) = \{\, E \rightarrow \texttt{a}\cdot\,\} \\ Z_3 & = & \delta(Z_0, \texttt{(}) = \{\, E \rightarrow \texttt{(}\cdot\, E\ \texttt{)}\,\} \cup \{\, E \rightarrow \cdot\texttt{a}\,,\; E \rightarrow \cdot\,\texttt{(}\ E\ \texttt{)}\,\} \\ Z_4 & = & \delta(Z_3, E) = \{\, E \rightarrow \texttt{(}\ E \cdot\ \texttt{)}\,\} \\ Z_5 & = & \delta(Z_4, \texttt{)}) = \{\, E \rightarrow \texttt{(}\ E\ \texttt{)}\cdot\,\} \end{array}$$

3.2.2 Tabela de decisão para um reconhecimento ascendente

A coleção de conjuntos de itens (conjunto de estados) fornece a base para a construção de uma tabela usada no algoritmo de reconhecimento. A tabela de decisão é uma matriz dupla, em que as linhas são indexadas pelo alfabeto da pilha (coleção de conjuntos de itens) e as colunas são indexadas pelos símbolos terminais e não terminais da gramática. Representa simultaneamente duas funções, designadas ACTION e GOTO. A função ACTION tem como argumentos um estado (símbolo da pilha) e um símbolo terminal (incluindo o $) e define a ação a realizar. Pode ser uma de *shift*, *reduce*, *accept* ou *error*. A função GOTO mapeia um estado e um símbolo não terminal num estado. É usada após uma operação de redução.

Veja-se um exemplo. Considerando a gramática e a coleção de conjunto de itens anteriores, obtém-se a seguinte tabela de decisão.

	a	(	)	$	E
Z_0	**shift** Z_2	**shift** Z_3			Z_1
Z_1				**accept**	
Z_2			**reduce** $E \rightarrow$ a	**reduce** $E \rightarrow$ a	
Z_3	**shift** Z_2	**shift** Z_3			Z_4
Z_4			**shift** Z_5		
Z_5			**reduce** $E \rightarrow$ (E)	**reduce** $E \rightarrow$ (E)	

- $Z = \{Z_0, Z_1, Z_2, Z_3, Z_4, Z_5\}$ é o alfabeto da pilha e foi obtida calculando a coleção de conjuntos de itens.
- **shift** Z_i, com $Z_i \in Z$, representa um deslocamento, no qual é consumido o símbolo à entrada e é feito o empilhamento (*push*) do símbolo Z_i.
- **reduce** $A \rightarrow \alpha$, onde $A \rightarrow \alpha$ é uma produção da gramática, representa uma redução, na qual são retirados da pilha tantos símbolo quantos os símbolo do corpo da regra.
- Os símbolos $Z_i \in Z$ na última coluna, representam os símbolos a empilhar após uma redução.
- **accept** representa a aceitação.
- As células vazias representam situações de erro.

3.2.3 Algoritmo de reconhecimento

O algoritmo seguinte mostra como se usa a tabela anterior. Nele, `top`, `push` e `pop` são funções de manipulação da pilha, com os significados habituais, e `lookahead` e `adv` são funções de manipulação da entrada que, respetivamente, devolve o próximo símbolo terminal à entrada e consome um símbolo.

Algoritmo 3.1

```
push(Z_0)
forever
    if (top() == Z_1 && lookahead() == $)
        aceita a entrada como pertencendo à linguagem.
    acc = ACTION[top(),lookahead()]
    if (acc is shift Z_i)
        adv(); push(Z_i);
    else if (acc is reduce A → α)
        pop |α| símbolos; push(GOTO[top(),A]);
    else
        rejeita a entrada
```

A aplicação deste algoritmo à palavra "`((a))`" resulta na tabela seguinte. No preenchimento dessa tabela, optou-se por separar em duas linhas as operações de *pop* e *push* das ações de redução. Desta forma fica mais claro que o símbolo a empilhar resulta do símbolo no topo da pilha após os *pops*.

pilha	**entrada**	**ação**
Z_0	`( ( a ) ) $`	**shift** Z_3
$Z_0\ Z_3$	`( a ) ) $`	**shift** Z_3
$Z_0\ Z_3\ Z_3$	`a ) ) $`	**shift** Z_2
$Z_0\ Z_3\ Z_3\ Z_2$	`) ) $`	**reduce** $E \rightarrow$ `a`
$Z_0\ Z_3\ Z_3$	`) ) $`	push Z_4
$Z_0\ Z_3\ Z_3\ Z_4$	`) ) $`	**shift** Z_5
$Z_0\ Z_3\ Z_3\ Z_4\ Z_5$	`) $`	**reduce** $E \rightarrow$ `( E )`
$Z_0\ Z_3$	`) $`	push Z_4
$Z_0\ Z_3\ Z_4$	`) $`	**shift** Z_5
$Z_0\ Z_3\ Z_4\ Z_5$	$	**reduce** $E \rightarrow$ `( E )`
Z_0	$	push Z_1
$Z_0\ Z_1$	$	**accept**

Na redução com a produção $E \rightarrow$ `a` foi feito o `pop` de 1 símbolo (número de símbolos no corpo da produção), ficando, em consequência, um Z_3 no topo da pilha. O Z_4 que foi empilhado logo a seguir corresponde a table$[Z_3, E]$. Nas duas reduções com a produção $E \rightarrow$ `( E )` são feitos o `pop` de 3 símbolos, ficando, em consequência, um Z_3 no topo da pilha, no primeiro caso, e um Z_0 no segundo.

Capítulo 4

Gramática de atributos

NOTA PRÉVIA: *Este capítulo é apenas um enumerado das secções do livro de referência (*Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition;*) que cobrem a matéria sobre gramáticas de atributos.*

4.1 Definição de gramática de atributos

No contexto destes apontamentos considera-se *gramáticas de atributos* o que no livro de referência se designa por *syntax-directed definitions*.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1, "Syntax-directed definitions".

4.1.1 Atributos herdados e atributos sintetizados

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1.1, "Inherited and synthesized attributes".

4.1.2 Construção de gramáticas de atributos

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1.2, "Evaluating an SDD at the nodes of a parse tree".

Comp 2023

4.2 Ordem de avaliação dos atributos

4.2.1 Grafo de dependências

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.1, "Dependency graphs".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.2, "Ordering the evaluation of attributes".

4.2.2 Tipos de gramáticas de atributos

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.3, "S-attributed definitions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.4, "L-attributed definitions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.5, "Semantic rules with controlled side effects".

Comp 2023