

Compiladores: Um Guia para uma compilação simples de Rust

Eduardo Morgado

dezembro, 2019

Conteúdo:

1	Introdução	3
1.1	O que é uma linguagem de programação?	3
1.2	Dois tipos de linguagem: compilada vs interpretada	3
1.3	Objectivo	4
2	Análise	6
2.1	Análise Léxica	6
2.1.1	Implementação de um lexer	8
2.1.2	Autómato Não Determinístico Finito	9
2.1.3	Autómatos Finitos Determinísticos	10
2.1.4	Minimização de DFA	11
2.1.5	Analizador Léxico	11
2.2	Análise Sintáctica	11
2.2.1	FIRST	13
2.2.2	FOLLOW	15
2.2.2.1	Resolver restrições	15
2.2.3	Análise Left-To-Right LL(1)	18
2.2.3.1	Parsing LL(1) com recurso a tabela	18
2.2.3.2	Reescrever gramática para análise LL(1)	20
2.2.3.3	Eliminar recursividade à esquerda	21
2.2.3.4	Recursividade indirecta à esquerda	22
2.2.3.5	Factorização à esquerda	22
2.2.3.6	Construção de analisadores LL(1)	23
2.2.4	Análise Bottom-Up Left-To-Right LR	23
	Bibliografia	24

Figuras:

1	Esquema de fases de compilação	5
2	Etapas de um gerador de scanners	7
3	Construção de fragmentos NDFA a partir de expressões regulares	10
4	Árvore de derivação para expressão (3-2)*4	12
5	Gramática exemplo	17
6	Gramática para paridade de parêntesis	19

Tabelas:

1	Restrições a gramática 5	17
2	Tabela LL(1) para gramática 6	19
3	Tabela de análise LL(1) para gramática 6, com entrada ()\$ e tabela 2	21

Lista de Code Snipets

1	Lexer Simples em Haskell	7
2	Lexer Simples em flex	8
3	Gramática simples para expressões	12
4	Percorrer por Top-Down Left-To-Right (LL)	12
5	Percorrer por Top-Down Right-To-Left (RL)	13
6	Percorrer por Bottom-Up Left-To-Right (LR)	13
7	Pseudocódigo para análise LL(1) com recurso a tabela [3]	20

1 Introdução

1.1 O que é uma linguagem de programação?

Uma linguagem de programação é essencialmente uma linguagem computacional utilizada para gerar programas/*scripts* que podem ser executados por um computador [2]. Cada linguagem terá a sua própria sintaxe.

Qualquer programa executado é um conjunto de bits, um ficheiro binário, cabe à unidade de processamento, interpretar esse conjunto de bits e executar as tarefas configuradas. No entanto, para o programador, torna-se difícil interpretar e gerar código binário, como tal surgem linguagens de programação destinadas a facilitar o desenvolvimento de programas, num dos níveis mais baixos de linguagens, encontra-se o Assembler¹, outras linguagens foram construídas sobre este chegando a linguagens de alto nível como o C++ e o Kotlin por exemplo.

Para um programa numa linguagem de alto nível poder ser executado pelo processador, este tem que o poder ler, como apenas identifica código binário o programa terá de passar por um **processo de tradução**, até o ficheiro original ser traduzido num ficheiro binário, este processo de tradução designa-se por **compilação**².

O processo de tradução de um ficheiro numa linguagem de programação para um ficheiro binário tem 3 grandes etapas:

- Tradução para código intermédio;
- Tradução para código *assembly*;
- Tradução para código binário;

A compilação incide sobre as duas primeiras etapas, irão ser estudadas em maior detalhe nas próximas secções.

1.2 Dois tipos de linguagem: compilada vs interpretada

Uma dada linguagem de programação pode ser compilada ou interpretada (ou ambos), as diferenças estão na forma como estas tratam e executam os ficheiros.

Uma linguagem compilada, é uma linguagem onde, após o processo de compilação, é expressa no conjunto de instruções **específicas à máquina onde é executada**, ou seja, processadores diferentes, apresentam conjuntos de instruções diferentes (por regra geral). Dessa forma o programa gerado, fica condicionado ao tipo de máquina onde será executado.

Já uma linguagem interpretada, é uma linguagem onde as instruções **não são executadas directamente** pela máquina mas sim por um outro programa.

Ambos os dois tipos podem realizar as mesmas tarefas uma vez que são Turing *complete*, no entanto, apresentam algumas vantagens/desvantagens.

¹A linguagem de programação mais próxima ao código binário, um exemplo dessa linguagem é o **Mips**.

²Um compilador é então um tradutor de uma linguagem para outra.

Entre as vantagens de linguagens compiladas estão:

- **Velocidade:** o programa executável contém apenas o código binário para a máquina em questão, conforme a sua arquitectura, dessa forma o programa pode ser corrido um número arbitrário de vezes sem causar nenhum impacto de tempo na tradução do executável, pois esta não é necessária.
- **Memória:** programas compilados, utilizam, por norma, menos memória, pois não é necessário fazer *caching* de traduções e não é necessário armazenar um *buffer* de instruções a executar.

No entanto, linguagens compiladas apresentam alguma desvantagens:

- Qualquer alteração ao programa requer uma **nova compilação**.
- *Debugging* torna-se mais difícil.
- Menor flexibilidade, um programa executável está dependente da arquitectura da máquina, não podendo ser partilhado para outras máquinas, de arquitecturas diferentes, é sempre necessário compilar o ficheiro de código fonte.

Linguagens interpretadas fornecem uma maior flexibilidade aos programas quando comparadas com as compiladas, apresentando algumas vantagens:

- **Tipagem dinâmica.**
- Facilidade em *debugging*.

No entanto, a execução de um programa por m interpretador é muito mais ineficiente quando comparada com uma execução normal/compilada, isto acontece devido ao facto de cada instrução ser interpretada em tempo de execução ou, como nos caso de interpretadores mais modernos, o código fonte ser convertido/compilado para um código intermédio antes de ser executado.

1.3 Objectivo

Através deste documento será possível implementar um compilador simples para a linguagem de programação Rust³. Para esse objectivo serão abordadas as etapas da fase de compilação até ser produzido o código Assembler. A Figura 1 apresenta esse esquema.

³Não iremos entrar em detalhe na sintaxe do Rust e poderemos vir a alterar a sua sintaxe para uma implementação mais simples será apenas para um subconjunto que deverá incluir:

- Uma função única `fn main(){ ...} !comandos if, if else, while ! funções de input/output ! e sequências de comandos separados por ; !;`
- Atribuições de Inteiros e booleanos e expressões aritméticas/booleanas através de comando `let |;`

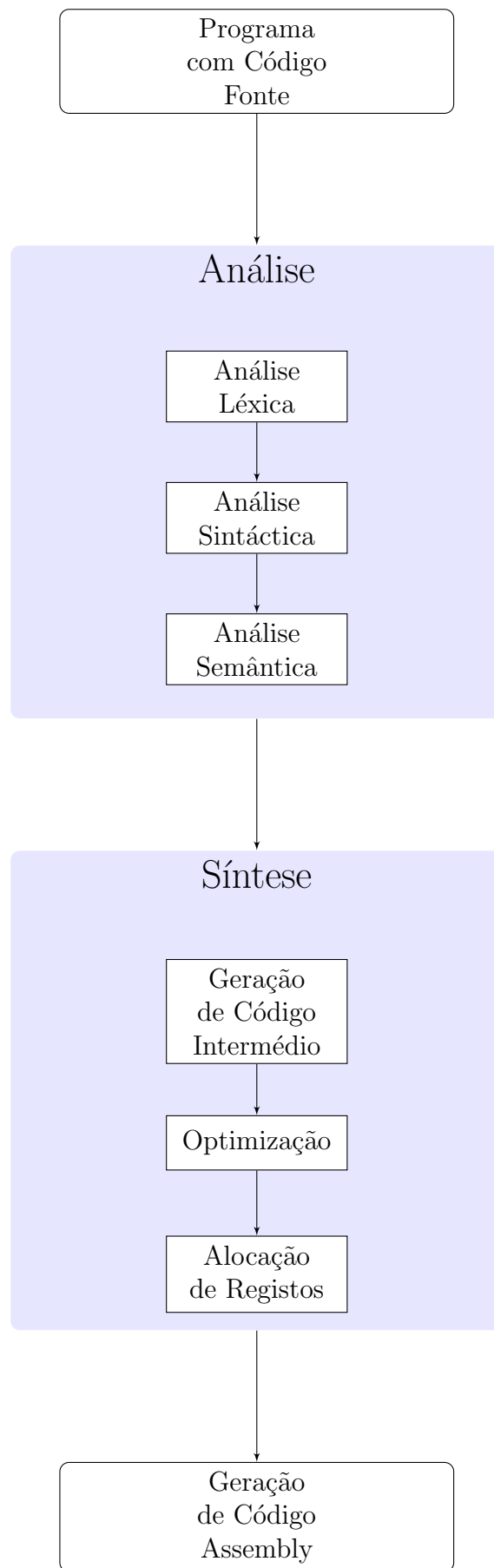


Figure 1: Esquema de fases de compilação

2 Análise

Para traduzir um programa de uma linguagem para outra, o compilador necessita de desconstruir o programa de forma a perceber a sua estrutura e o seu significado para o depois reorganizar de uma nova forma [1]. Tal como a Figura 1 mostra, o compilador possui duas fase principais, a fase de **análise** e a fase de **síntese**, sendo a fase frontal a fase de análise⁴.

Esta fase pode ainda ser dividida em 3 fases:

- **Análise léxica** (2.1): nesta fase o programa é partido em **tokens**⁵;
- **Análise sintáctica**: nesta fase faz-se *parsing* do programa e é gerada a sua árvore abstracta;
- **Análise semântica**: nesta fase a partir da árvore gerada, atribuí-se significado ao programa.

2.1 Análise Léxica

Nesta fase é gerado um analisador lexical, **lexer**, que irá, a partir do *input* inicial, gerar uma sequência de caracteres específicos, **tokens**, do programa. Para facilitar o processo, são ignorados espaços e comentários, para o consumo e escolha desses *tokens* são utilizadas **expressões regulares**.

Dependendo da linguagem de programação que faz a análise lexical, a geração do *lexer* pode ser feita mais facilmente, um exemplo seria o Haskell, ver *snippet* 1.

Quando a linguagem a analisar é relativamente simples, um exemplo seria uma linguagem apenas capaz de resolver contas aritméticas, a geração do *lexer* pode ser feita manualmente. No entanto, para linguagens com uma maior funcionalidade⁶, a geração manual do *lexer* será mais trabalhosa, para isso, actualmente existem ferramentas de **geração automática de analisadores léxicos**, através da especificação dos *tokens* da linguagem (por expressões regulares) é formado um **gerador de scanners** que irá fazer a leitura do programa e fornece o *output* ao *lexer*. Os geradores de *scanners*, por regra geral seguem estes passos:

⁴Esta fase não é apenas comum a um compilador, um interpretador **também apresenta a fase de análise**, o interpretador é a fase de análise aliada a estruturas de controlo de execução, seguindo o formato **REPL** (*read-eval-print-loop*).

Para interpretadores mais modernos, estes já possuem outras fases similares a de síntese onde convertem o programa para um código intermédio e depois executam esse código.

⁵Caracteres que podem ser vistos como **unidades na gramática** da linguagem de programação a ser analisada, uma linguagem de programação, tem um **conjunto finito** de tipos de **tokens** [1]

⁶Quanto maior a funcionalidade de uma linguagem (as acções disponibilizadas pela mesma) maior será a sua complexidade, irá ter uma maior quantidade de *tokens*.

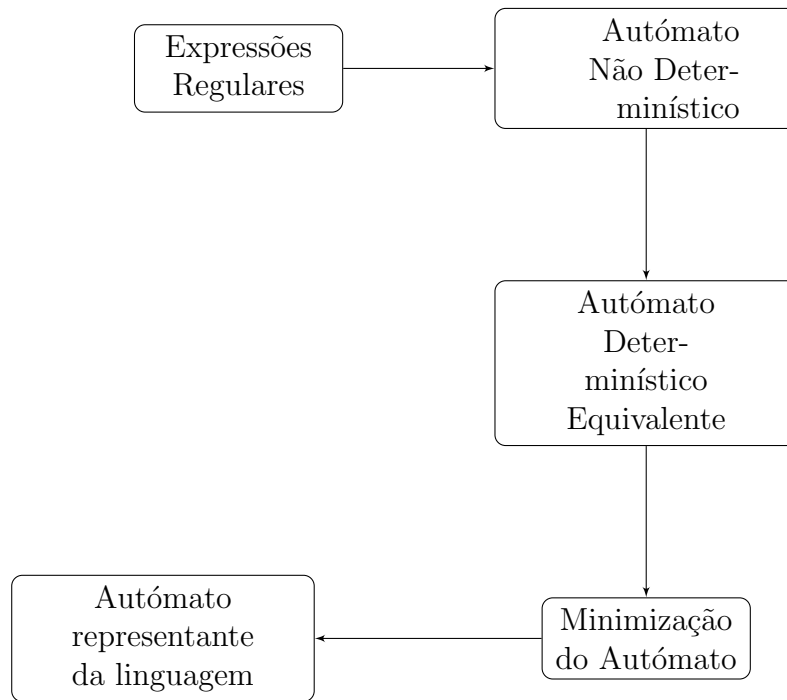


Figure 2: Etapas de um gerador de scanners

Code Snippet 1: Lexer Simples em Haskell

```

data Token = TokenInt Int
           | TokenVar String
           | TokenWhile
           | TokenIf
           | TokenAtrib
           | TokenAdd
           | ...
           | TokenMult
           | TokenOB
           | TokenCB
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
    | isSpace c = lexer cs
    | isAlpha c = lexVar(c:cs)
    | isDigit c = lexNum(c:cs)
lexer ('=:cs) = TokenAtrib : lexer(cs)
lexer ('+:cs) = TokenAdd : lexer(cs)
lexer ('*':cs) = TokenMult : lexer(cs)
lexer ('(' :cs) = TokenOB : lexer(cs)
lexer (')':cs) = TokenCB : lexer(cs)

```

```

lexNum :: String -> [Token]
lexNum cs = TokenInt (read num) : lexer rest
    where (num,rest) = span isDigit cs

```

```

lexVar :: String -> [Token]
lexVar cs = case (span cs) of
    ("while",rest) = TokenWhile : lexer rest
    ("if",rest) = TokenIf : lexer rest
    (var,rest) = TokenVar : lexer rest

```

Nota: Os geradores de *scanners* podem também fazer programas de *matching* de expressões regulares e não apenas *lexers*.

2.1.1 Implementação de um lexer

Tal como referido anteriormente, actualmente é possível gerar automaticamente o *lexer* através de programa/linguagens auxiliares, uma delas, a utilizada nesta implementação, será o

flex

Code Snippet 2: Lexer Simples em flex

```
%{
    #include <stdlib.h>
    #include <parser.h>
    int yyline = 1;
}%

%option noyywrap

%%

"//" .* \n      { /*Comment => consume*/ yyline++;}
[ \t]+          { /*Tab => consume*/}
\n             { yyline++;}
\-[0-9]+        { yylval.int_val = atoi(yytext); return INT;}

"+"            {return ADD_OP;}
"_"            {return SUB_OP;}
"*"            {return MULT_OP;}
"/"            {return DIV_OP;}
"%/"           {return MOD_OP;}

"&&"           {return AND_OP;}
"||"           {return OR_OP;}

"=="           {return EQ_OP;}
"!="           {return NOT_EQ_OP;}
">"           {return GRT_OP;}
"<"           {return LT_OP;}
">="          {return GRT_EQ_OP;}
"<="          {return LT_EQ_OP;}

"="            {return ATTR_OP;}

"main"         {return F_MAIN;}
"fn"           {return F_DEC;}

"let"          {return LET_OP;}
```



```

" if"          {return IF_OP;}
" else"        {return ELSE_OP;}

" while"       {return WHILE_OP;}
" println!"    {return PRINT_CMD;}
" read_line!"  {return READ_CMD;}

" {"           {return OPEN_BRACKET;}
" }"           {return CLOSE_BRACKET;}
" ("           {return OPEN_PARENT;}
" )"           {return CLOSE_PARENT;}

";"            {return SEMICOLON;}

[ a-zA-Z_ ] [ a-zA-Z0-9_ ] *    {
                                yylval.var_val = strdup(yytext);
                                return VARNAME;
                                }

.                               {yyerror(" Unexpected character ");}

%%

```

Este ficheiro irá percorrer o programa fornecido como *input* e irá gerar *tokens* relativos ao programa, isto após terem sido especificados (através de expressões regulares). Posteriormente, este programa será utilizado em conjunto com um *parser* para atribuir contexto ao programa.

2.1.2 Autómato Não Determinístico Finito

Para a transformação de expressões regulares em *tokens* de forma eficiente são utilizados **autómatos não determinísticos finitos** (N DFA), podendo ser utilizados para decidir se uma data *string* é membro de algum conjunto de *strings* [4].

Para isso, é definido um estado inicial e a partir deste podemos chegar a outros estados através de um caminho por *epsilon* ou pela leitura de um carácter esperado ao qual temos um caminho e um estado associado, após a leitura de todos os caracteres, é verificado se o estado actual é um estado válido, caso seja, a *string* lida encontra-se no conjunto de *strings* aceites pela linguagem definida pelo autómato [4].

Um programa(o *flex* neste caso) que verifica se uma dada *string* é aceite pelo autómato terá que **verificar todos os caminhos possíveis** até pelo menos um aceitar a *string* [4]. Este processo pode ser feito de duas formas [4]:

- **Backtracking** até ser encontrado um caminho válido;
- Percorrer todos os caminhos **simultaneamente**.

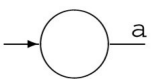
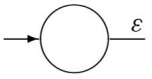
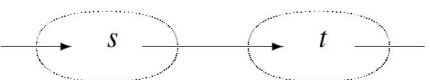
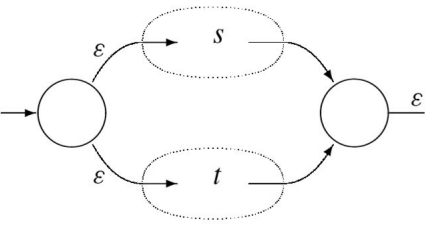
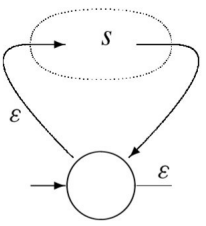
Regular expression	NFA fragment
a	
ε	
st	
$s t$	
s^*	

Figure 3: Construção de fragmentos NDFA a partir de expressões regulares

Ambas estas formas consomem demasiado tempo para tornar o autómato finito não determinístico um reconhecedor de linguagens eficiente, dessa forma, serão apenas utilizados como o **passo inicial** entre expressões regulares e o autómato determinístico finito (DFA)⁷ [4]. A Figura 3 apresenta um exemplo de construções de fragmentos de NDFA a partir de expressões regulares.

2.1.3 Autómatos Finitos Determinísticos

Este tipo de autómatos são NDFAs com certas restrições [5]:

- Não existem transições por *epsilon*;
- Transições com mesmos rótulos/nomes a partir de um mesmo estado não podem existir.

Com estas restrições, o estado e o seu próximo símbolo de *input* **identificam unicamente** a transição (daí o nome determinístico) [5]. Qualquer NDFA pode ser convertido para um **DFA equivalente**⁸.

⁷Ao utilizar o NDFA desta forma, é proporcionado uma construção mais simples e directa de um DFA.

⁸Não iremos entrar em detalhe na formulação de NDFAs e conversão para DFAs equivalentes, estes tópicos

2.1.4 Minimização de DFA

Após o autômato finito não determinístico ser convertido num autômato finito determinístico, este não é o autômato mínimo para a linguagem (existem estados desnecessários e equivalentes que devem ser removidos) [6]. A tarefa de minimização é uma tarefa relativamente simples, dessa forma, maior parte de geradores automáticos de *lexers* realizam esta operação.

2.1.5 Analisador Léxico

Vimos anteriormente um gerador automático de *lexers*, ao criar esse gerador, definimos os **tokens** que aceita de forma a gerar um **único** DFA capaz de realizar testes para todas a *tokens* ao invés de gerar um autômato para cada uma das expressões regulares [7].

Uma vez que, as *tokens* são definidas por expressões regulares, pode ser formada um expressão regular que consiste na união das linguagens definidas pelas expressões regulares individuais e o DFA gerado a partir desta expressão regular combinada irá ser capaz de verificar todos os *tokens* ao mesmo tempo [7], dessa forma, cada *token* deve poder ser distinguido.

A construção do DFA do conjunto das expressões regulares segue este algoritmo [7]:

- Construir N_{i_s} NFAs para cada n_i expressão regular;
- Marcar estados válidos de cada NDFA pelo nome das *tokens* que eles aceitam;
- Combinar os NDFAs num NDFA único, adicionando um estado iniciante com transições por *epsilon* para cada estado iniciante dos NDFAs;
- Converter o NDFA combinado num DFA;
- Cada estado de aceitação consiste num conjunto de estados NDFA, com pelo menos sendo um estado de aceitação marcado pelo nome da *token*.

O gerador automático de *lexers* gera o autômato finito determinístico mínimo correspondente a cada expressão regular.

2.2 Análise Sintáctica

A análise léxica parte o *input* em *tokens*, a análise sintáctica (*parsing*) **recombina** esses *tokens* numa estrutura que reflecte a organização dos dados, essa estrutura chama-se **árvore sintáctica abstracta** (AST)⁹ [8].

As folhas na AST são as *tokens* produzidas na análise léxica, se a árvore for percorrida/lida da esquerda para a direita a sequência de *tokens* é a mesma que a do texto de código fonte [8], sendo assim na geração da AST o único aspecto relevante será a forma como as folhas são combinadas na formatação da árvore e a nomenclatura dos nós interiores [8].

devem ser abordados numa disciplina precedente a compiladores estes

⁹Simplificada o suficiente para guardar a estrutura da linguagem do *input* em memória

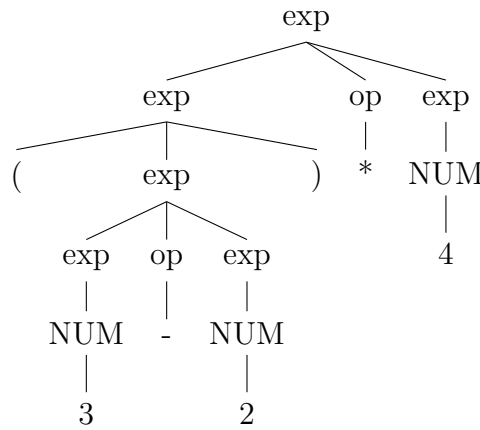


Figure 4: Árvore de derivação para expressão $(3-2)*4$

A análise sintáctica requer por sua vez métodos mais avançados quando comparada com a análise léxica, no entanto, é utilizada a mesma abordagem, a utilização de uma notação adequada para interpretação que por sua vez é transformada numa notação máquina mais eficiente, essa notação para interpretação humana são as **gramáticas independentes de contexto** podendo estas por sua vez, ser traduzidas para **autómatos pilha**¹⁰ [8] A estratégia utilizada para percorrer a AST define o algoritmo de geração do autômato pilha, existem **duas** estratégias principais ou **técnicas de parsing**:

- **Top-Down** (complexidade exponencial):
 - **Left-To-Right (LL)**;
 - **Right-To-Left (RL)**;
- **Bottom-Up** (complexidade linear):
 - **Left-To-Right (LR / SLR)**;
 - **Right-To-Left (RR)**;

Por exemplo, o *snippet* 3 apresenta uma gramática para expressões aritméticas simples e a Figura 4 representa a árvore de derivação para a expressão $(3-2)*4$.

Code Snippet 3: Gramática simples para expressões

```

exp -> exp op exp      op -> + | - | * | / | %
      | (exp)
      | NUM
NUM -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Aplicando as estratégias para percorrer a árvore de derivação, são geradas as seguintes sequências de produções.

Code Snippet 4: Percorrer por Top-Down Left-To-Right (LL)

```
exp => exp op exp => (exp) op exp
```

¹⁰Autómatos similares aos DFAs podendo ainda utilizar uma pilha, permitindo contagem e utilização de símbolos externos à máquina

```

=> ( exp op exp ) op exp => ( NUM op exp ) op exp
=> ( NUM - exp ) op exp => ( NUM - NUM ) op exp
=> ( 3 - 2 ) op exp => ( 3 - 2 ) * exp
=> ( 3 - 2 ) * NUM => ( 3 - 2 ) * 4

```

Code Snippet 5: Percorrer por Top-Down Right-To-Left (RL)

```

exp => exp op exp => exp op NUM
=> exp * 4 => ( exp ) * 4
=> ( exp op exp ) * 4 => ( exp op NUM ) * 4
=> ( exp - 2 ) * 4 => ( NUM - 2 ) * 4
=> ( 3 - 2 ) * 4

```

Code Snippet 6: Percorrer por Bottom-Up Left-To-Right (LR)

```

1)  (
2)  exp —— NUM —— 3
3)  op —— -
4)  exp —— NUM —— 2
5)  )
6)  op —— *
7)  exp —— NUM —— 4
8)  exp

```

Tal como observado, a árvore sintáctica para o *input* fornecido, é geradas através da procura de **derivações** do *input* a partir do símbolo inicial da gramática [9]. Esta procura pode ser feita de forma exaustiva através de tentativas e com validação da escolha, estes métodos são métodos de **análise preditiva**, os *parsers* por ele gerados geram a árvore sintáctica a partir da raiz até às folhas, dessa forma, são denominados *parsers* **Top-Down** [9].

Uma procura alternativa é feita através da pesquisa no *input* de partes que possam fazer *match* com as primeiras *tokens* nas regras da gramática procedendo depois (através do consumo do *input*) à filtragem das regras e construindo a árvore sintáctica, após todo o *input* ser consumido, a árvore sintáctica estará construída, uma vez que, irão ser escolhidas as regras que levarão a produções correctas, este método é denominado de **análise determinística**, os *parsers* por ele gerados são *parsers* **Bottom-Up** [9].

2.2.1 FIRST

Construir a AST para uma dada *string* resume-se à escolha das sequências de produções que irão consumir a dada *string* de forma correcta, até se chegar a um **símbolo terminal**. Para gramáticas com produções iniciais com vários símbolos não-terminais, a escolha é feita baseada no símbolo a ser avaliado na iteração corrente, ou seja, cada produção poderá derivar um conjunto de *strings* com símbolos iniciais disjuntos, basta comparar esses símbolos com o símbolo desejado e se houver *match* a produção que deriva o conjunto é escolhida, caso contrário e caso exista uma

produção vazia essa é escolhida [10].

A função que, dada uma sequência de símbolos iniciais a produções, retorna o conjunto de símbolos iniciais das *strings* derivadas por cada símbolo de cada produção é denominada por **FIRST** [10]. Dessa forma:

$$\gamma \in \mathbf{FIRST}(\alpha) \text{ sse } \alpha \Rightarrow * \gamma\beta$$

Para ser possível calcular FIRST é necessário recorrer a uma função auxiliar *nullable* que, para uma dada sequência α de símbolos da gramática, indica se é possível derivar a **string** vazia/epsilon (ε) [10].

$$\alpha \in \text{Nullable}(\alpha) \text{ sse } \alpha \Rightarrow \varepsilon$$

Uma produção onde existe pelo menos um símbolo terminal no resultado da produção (lado direito) não é *Nullable*, se a produção do lado direito começar por um símbolo terminal, o conjunto *FIRST* consiste apenas nesse símbolo. As regras gerais para o cálculo de *Nullable* são as seguintes [10]:

- $\text{Nullable}(\varepsilon) = \text{true}$
- $\text{Nullable}(a) = \text{false}$
- $\text{Nullable}(\alpha\beta) = \text{Nullable}(\alpha) \wedge \text{Nullable}(\beta)$
- $\text{Nullable}(N) = \text{Nullable}(\alpha_1) \vee \dots \vee \text{Nullable}(\alpha_n),$
onde produções para N são: $N \longrightarrow \alpha_1, \dots, N \longrightarrow \alpha_n$

Onde a é símbolo terminal, N é símbolo não terminal, α e β representam sequências de símbolo gramaticais e ε representa a palavra vazia.

As regras gerais para o cálculo de *FIRST* são as seguintes [10]:

- $\text{FIRST}(\varepsilon) = \emptyset$
- $\text{FIRST}(a) = \{a\}$
- $\text{FIRST}(\alpha\beta) = \begin{cases} \text{FIRST}(\alpha) \cup \text{FIRST}(\beta) & \text{se } \text{Nullable}(\alpha) \\ \text{FIRST}(\alpha) & \text{se } \alpha \text{ não for } \text{Nullable} \end{cases}$
- $\text{FIRST}(N) = \text{FIRST}(\alpha_1) \cup \dots \cup \text{FIRST}(\alpha_n)$
onde as produções de N são $N \longrightarrow \alpha_1, \dots, N \longrightarrow \alpha_n$

Onde a é terminal, N é não terminal e α e β são sequências de símbolos gramaticais e ε representa a palavra vazia.

2.2.2 FOLLOW

Com *FIRST* é possível escolher produções com símbolos não terminais, permitindo a escolha de **até uma produção anulável** [10], no entanto, para linguagens mais complexas, podem apresentar várias produções anuláveis que terão que ser consideradas, para isso são utilizados funções/conjuntos **FOLLOW** para símbolos não terminais [11].

Se S for símbolo inicial da gramática, $\alpha \in \text{FOLLOW}(N)$ sse $S \Rightarrow \alpha N \alpha \beta$

Ou seja, α está no conjunto $\text{FOLLOW}(N)$ se numa dada altura durante a derivação, α segue a regra N [11]. Ao contrário de $\text{FIRST}(N)$ que representa uma propriedade para as produções na regra N , $\text{FOLLOW}(N)$ representa as produções que **podem directa ou indirectamente utilizar a regra N** [11]. Para ser possível analisar produções para a terminação de *strings* é necessário verificar se $S \Rightarrow \alpha N$, ou seja, se existem derivações onde a regra N pode ser atingida no final da *string* [11], esta verificação pode ser facilitada adicionando uma nova produção à gramática:

$$S' \rightarrow S\$$$

sendo S' o novo símbolo inicial da gramática, dessa forma:

$$\$ \in \text{FOLLOW}(N) \text{ sse } S' \Rightarrow \alpha N, \text{ ou seja, quando } S \Rightarrow \alpha N$$

A forma mais fácil de calcular o *FOLLOW* é gerar um **conjunto de restrições** resolvidas pela resolução de m número mínimo de conjuntos que satisfaçam as restrições. Uma produção $M \rightarrow \alpha N \beta$ gera a restrição $\text{FIRST}(\beta) \subseteq \text{FOLLOW}(N)$, uma vez que β pode seguir N , caso β seja *Nullable* ($\text{Nullable}(\beta)$), a produção gera outra restrição, $\text{FOLLOW}(M) \subseteq \text{FOLLOW}(N)$ ¹¹ [11].

Se o lado direito (resultante do \rightarrow) duma produção apresentar vários símbolos não terminais, são adicionadas restrições para todas as ocorrências em conjuntos $\text{beta}\alpha$ (β sequência de símbolos ou regra gramatical e α o símbolo não terminal)¹² [11].

2.2.2.1 Resolver restrições

Para resolver as restrições devemos considerar inicialmente conjuntos *FOLLOW* vazios para todos os não terminais [11].

- Restrições da forma $\text{FIRST}(\beta) \subseteq \text{FOLLOW}(N)$:

- (a) Calcular $\text{FIRST}(\beta)$;
- (b) Adicionar $\text{FIRST}(\beta)$ a $\text{FOLLOW}(N)$;

Depois de resolver estas restrições podemos avançar para as próximas.

- Restrições da forma $\text{FOLLOW}(M) \subseteq \text{FOLLOW}(N)$:

- (a) Adicionar todos os elementos em $\text{FOLLOW}(M)$ ao conjunto $\text{FOLLOW}(N)$;

¹¹Se um símbolo $c \in \text{FOLLOW}(M)$, existe uma derivação $S' \Rightarrow \gamma M c \delta$, uma vez que, $M \rightarrow \alpha N \beta$ e $\text{Nullable}(\beta)$, podemos desenvolver $\gamma M c \delta \Rightarrow \gamma \alpha N c \delta$, logo $c \in \text{FOLLOW}(N)$ [11].

¹²Por exemplo, a produção $A \rightarrow B c B$ gera a restrição $\{c\} \subseteq \text{FOLLOW}(B)$ e a restrição $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$ através da separação do primeiro B e do segundo, respectivamente [11].

- (b) Repetir o passo anterior até resolver todas as restrições desta forma;

Para calcular os conjuntos *FOLLOW* da gramática são aplicados os seguintes passos [11]:

1. Adicionar nova regra à gramática, $S' \rightarrow S\$$ onde S' passa a ser o novo símbolo da gramática (deixando de ser S);
2. Para cada não terminal N , localizar todas as suas ocorrências em resultados de produções na gramática. Para cada ocorrência realizar os seguintes passos:
 - 2.1. Seja β (podendo ser vazio) o resto do resultado da produção depois de N , ou seja para uma produção M (com M possivelmente igual a N) fica na forma $M \rightarrow \alpha N \beta$, se o resultado de uma produção contiver múltiplas ocorrências de N , estas devem ser separadas.
 - 2.2. Seja $m = FIRST(\beta)$ e se β não for vazio é adicionada a restrição $m \subseteq FOLLOW(N)$ ao conjunto de restrições.
 - 2.3. Se $Nullable(\beta)$ (caso β seja vazio, também será *Nullable*), todos os símbolos não terminais que enunciam regras (lado esquerdo de produção) que sejam diferentes de N iram adicionar uma nova restrição $FOLLOW(M) \subseteq FOLLOW(N)$ ao conjunto de restrições.
3. Resolver as restrições através dos seguintes passos:
 - 3.1. Para qualquer símbolo não terminal diferente de S' , considerar conjuntos $FOLLOW(N)$ vazios.
 - 3.2. Para cada restrição $m \subseteq FOLLOW(N)$ gerada no ponto 2.1, adicionar os valores de m em $FOLLOW(N)$.
 - 3.3. Para cada restrição da forma $FOLLOW(M) \subseteq FOLLOW(N)$ adicionar os conteúdos de $FOLLOW(M)$ a $FOLLOW(N)$, repetir este passo até ser atingido um ponto de paragem.

Consideremos a seguinte gramática na figura 5 como exemplo, irá ser adicionada uma nova produção $T' \rightarrow T\$$ de forma a lidar com condições de terminação. A tabela 1 apresenta as restrições para a gramática 5, é gerada da seguinte forma (aplicando os passos 2.1, 2.2 e 2.3 dos passos de resolução de conjuntos *FOLLOW*, apresentados anteriormente, aos símbolos não terminais T e R) [11]:

- Para T:
 - i Produção $T' \rightarrow T\$$, $\beta = \$$, $m = FIRST(\beta)$ e $FIRST(\$) = \{\$ \}$ pelo que $\{\$ \} \subseteq FOLLOW(T)$.
 - ii Produção $T \rightarrow aTc$, $\beta = c$, $m = FIRST(\beta) = FIRST(c) = \{c\}$, pelo que, $\{c\} \subseteq FOLLOW(T)$.
- Para R:
 - i Produção $T \rightarrow R$, aplicar passo 2.3, pelo que, $FOLLOW(T) \subseteq FOLLOW(R)$.
 - ii Produção $R \rightarrow$, β vazio, não é adicionada restrição.
 - iii Produção $R \rightarrow RbR$, $\beta = bR$, aplicar regra 2.1, separar ocorrências, $\beta_1 = b$, $\beta_2 = R$, $FIRST(\beta_1) = FIRST(b) = \{b\}$, ao β_2 aplicar regra 2.3, pelo que $\{b\} \subseteq FOLLOW(R)$, $FOLLOW(R) \subseteq FOLLOW(R)$.

$$\begin{aligned}
T &\longrightarrow R \\
T &\longrightarrow aTc \\
R &\longrightarrow \\
R &\longrightarrow RbR
\end{aligned}$$

Figure 5: Gramática exemplo

Produções	Restrições
$T' \longrightarrow T\$$	$\{\$ \} \subseteq FOLLOW(T)$
$T \longrightarrow R$	$FOLLOW(T) \subseteq FOLLOW(R)$
$T \longrightarrow aTc$	$\{c\} \subseteq FOLLOW(T)$
$R \longrightarrow$	
$R \longrightarrow RbR$	$\{b\} \subseteq FOLLOW(R), FOLLOW(R) \subseteq FOLLOW(R)$

Table 1: Restrições a gramática 5

Após gerar a tabela de restrições e para ser possível calcular os conjuntos de *FOLLOW* da gramática 5 devem primeiro ser calculados os conjuntos *FIRST* [11]:

$$FIRST(T') = FIRST(T) \cup FIRST(\$)$$

$$FIRST(\$) = \{\$ \}$$

$$FIRST(T) = FIRST(R) \cup FIRST(aTc)$$

$$FIRST(R) = FIRST(RbR)$$

$$FIRST(aTc) = FIRST(a) \text{ (a não é Nullable)}$$

$$FIRST(a) = \{a\}$$

$$FIRST(c) = \{c\}$$

$$FIRST(RbR) = FIRST(R) \cup FIRST(b) \cup FIRST(R)$$

$$FIRST(b) = \{b\}$$

Para inicializar os conjuntos *FOLLOW*, aplicar a regra 3.2 e utilizar as restrições que utilizam elementos de conjuntos *FIRST*, ou seja restrições $\{\$ \} \subseteq FOLLOW(T)$, $\{c\} \subseteq FOLLOW(T)$, $\{b\} \subseteq FOLLOW(R)$ [11]. Dessa forma,

$$FOLLOW(T) \supseteq \{\$, c\}$$

$$FOLLOW(R) \supseteq \{b\}$$

Pode depois ser aplicada a regra 3.3 à restrição $FOLLOW(T) \subseteq FOLLOW(R)$, adicionar os conteúdos de $FOLLOW(T)$ a $FOLLOW(R)$:

$$FOLLOW(T) \supseteq \{\$, c\}$$

$$FOLLOW(R) \supseteq \{\$, c, b\}$$

Estando todas as restrições satisfeitas, pode ser calculado os conjuntos $FOLLOW$:

$$FOLLOW(T) = \{\$, c\}$$

$$FOLLOW(R) = \{\$, c.b\}$$

2.2.3 Análise Left-To-Right LL(1)

Anteriormente, foi visto como escolher produções com base nos conjuntos $FIRST$ e $FOLLOW$, ou seja, uma produção $N \rightarrow \alpha$ é escolhida para símbolo de entrada c se [12]:

- $c \in FIRST(\alpha)$ ou,
- $Nullable(\alpha) \wedge c \in FOLLOW(N)$

Caso seja possível, através destas regras, escolher sempre uma produção de forma **única**, o método de análise é **LL(1)**¹³. Este tipo de *parsing* apresenta dois métodos de implementação, apenas será analisado o segundo, o método com recurso a tabela.

2.2.3.1 Parsing LL(1) com recurso a tabela

Neste método a selecção de produções é definida numa tabela, um programa **não recursivo** utiliza essa tabela e um **pilha** para realizar a análise. A tabela apresenta correspondências de símbolos terminais (colunas) para símbolos terminais (linhas), para cada par terminal/não terminal, apresenta a produção a ser escolhida (se existir) para o não terminal caso o *lookahead* seja o terminal da coluna em questão, ou seja, a produção $N \rightarrow a$ está na tabela T na posição $T[N, a]$ se $a \in FIRST(\alpha)$ ou se $Nullable(\alpha) \wedge a \in FOLLOW(N)$.

Iremos considerar e utilizar a gramática 6 para a construção da tabela. A geração da tabela ocorre da seguinte forma:

1. Inserir uma nova regra $S' \rightarrow S\$$ à gramática e considerar o símbolo S' como o novo símbolo inicial
2. Gerar tabela de restrições, com restrições:
 - $\{\$\} \subseteq FOLLOW(S)$
 - $\{)\} \subseteq FOLLOW(S)$

¹³O primeiro L indica a direcção de leitura (*left-to-right*), o segundo indica a ordem de derivação (esquerda) e 1 indica que o algoritmo irá sempre manter um símbolo *lookahead* [12].

$$\begin{aligned} S &\longrightarrow (S)S \\ S &\longrightarrow \varepsilon \end{aligned}$$

Figure 6: Gramática para paridade de parêntesis

3. Calcular conjunto *FOLLOW* para símbolos não terminais, $FOLLOW(S) = \{ \$,) \}$
4. Considerar símbolos de entrada: (,) e \$
5. Gerar tabela *M* com colunas (,) e \$ e linha *S*, calcular $M[N_i, t_j]$ para *i* símbolos não terminais e *j* símbolos terminais.
6. Calcular entradas $M[S, (]$, $M[S,)]$ e $M[S, \$]$:
 - 6.1. $M[S, (] = S \longrightarrow (S)S$
 - 6.1.1. Testando para $N = S$, $\alpha = (S)S$ e $c = ($
 - 6.1.2. $c \in FIRST(\alpha) \equiv (\in FIRST((S)S)$, pelo que a produção $S \longrightarrow (S)S$ é escolhida
 - 6.2. $M[S,)] = S \longrightarrow \varepsilon$
 - 6.2.1. Testando para $N = S$, $\alpha = (S)S$ e $c =)$
 - 6.2.1.1. $) \notin FIRST((S)S)$
 - 6.2.1.2. $Nullable((S)S) = false$, esta produção não é escolhida
 - 6.2.2. Testando para $N = S$, $\alpha = \varepsilon$ e $c =)$
 - 6.2.2.1. $) \notin FIRST(\varepsilon)$
 - 6.2.2.2. $Nullable(\alpha) = Nullable(\varepsilon) = true \wedge) \in FOLLOW(S)$, pelo que a produção $S \longrightarrow \varepsilon$ é escolhida
 - 6.3. $M[S, \$] = S \longrightarrow \varepsilon$
 - 6.3.1. Testando para $N = S$, $\alpha = (S)S$ e $c = \$$
 - 6.3.1.1. $\$ \notin FIRST((S)S)$
 - 6.3.1.2. $Nullable((S)S) = false$, esta produção não é escolhida
 - 6.3.2. Testando para $N = S$, $\alpha = \varepsilon$ e $c = \$$
 - 6.3.2.1. $\$ \notin FIRST(\varepsilon)$
 - 6.3.2.2. $Nullable(\alpha) = Nullable(\varepsilon) = true \wedge \$ \in FOLLOW(S)$, pelo que a produção $S \longrightarrow \varepsilon$ é escolhida
7. Adicionar as novas entradas para obter a tabela 2

	()	\$
S'	$S \longrightarrow (S)S$	$S \longrightarrow \varepsilon$	$S \longrightarrow \varepsilon$

Table 2: Tabela LL(1) para gramática 6

Após gerar a tabela, o programa de análise 7 irá utilizar uma pilha que, em qualquer momento apresenta o pedaço da derivação que ainda não teve *match* com o texto de entrada, caso a pilha fique vazia a análise termina [3]. Se a pilha não estiver vazia:

- Se o topo da pilha contiver um símbolo terminal, esse símbolo é satisfeito com o de entrada e caso sejam iguais é feito um *pop* da stack.
- Se o topo da pilha for um símbolo não terminal irá ser consultada a tabela LL(1) com base no símbolo da próxima entrada (*lookahead*), caso a entrada para esse símbolo não exista, é reportado um erro uma vez que a derivação não é a correcta. Se a tabela apresentar uma entrada para o *lookahead* é feito *pop* do símbolo não terminal da pilha e é inserida o resultado da produção (lado direito) da entrada escolhida na tabela, os símbolos da produção são inseridos de forma a que o primeiro símbolo seja também o topo da pilha.

A tabela 3 apresenta a análise para o texto de entrada $()\$$ para a gramática ?? com recurso à tabela LL(1) 2.

Code Snippet 7: Pseudocódigo para análise LL(1) com recurso a tabela [3]

```
stack := empty ; push(S', stack)
while stack not empty do
    if top(stack) is terminal then
        match(top(stack)) ; pop(stack)
    else if table(top(stack), next) is empty then
        reportError
    else
        rhs := rightHandSide(table(top(stack), next)) ;
        pop(stack) ;
        pushList(rhs, stack)
```

Este algoritmo apenas verifica se o texto de entrada é válido para a gramática, ou seja se pertence à linguagem, podendo, a partir dele, construir a AST para a linguagem, quando um símbolo não terminal é adicionado à pilha, se a árvore estiver vazia este é adicionado à árvore (ou à posição correta se não estiver vazia), quando o símbolo não terminal é substituído por um dos resultados das suas produções (lado direito), são gerados nós para cada um dos símbolos do resultado da produção e esses nós são adicionados como filhos do nó com o símbolo a ser substituído [3].

Nota: Quando um dado símbolo permite várias escolhas de produções de símbolos não terminais N , existe um **conflito** desse símbolo para o não terminal N [13]. Os conflitos podem ser causados por **gramáticas ambíguas** (podem existir gramáticas não ambíguas que causem conflitos), sendo assim, é necessário **resolver a ambiguidade**, no entanto, existem linguagens com gramáticas independentes de contexto não ambíguas que não conseguem gerar uma tabela LL(1) sem conflitos, essas linguagens são denominadas **non-LL(1)** [13].

2.2.3.2 Reescrever gramática para análise LL(1)

Os métodos utilizados para reescrever a gramática são a **eliminação de recursividade** à esquerda e a **factorização** à esquerda.

Stack	Input	Action
S \$	()\$	$S \rightarrow (S)S$
(S) S \$	()\$	match
S) S \$)\$	$S \rightarrow \varepsilon$
) S \$)\$	match
S \$	\$	$S \rightarrow \varepsilon$
\$	\$	ACCEPT (pilha vazia)

Table 3: Tabela de análise LL(1) para gramática 6, com entrada ()\$ e tabela 2

2.2.3.3 Eliminar recursividade à esquerda

Uma produção é recursiva à esquerda se tem a forma $N \rightarrow N\alpha$ onde N é símbolo não terminal e α é uma seqüências de símbolos gramaticais. Para resolver este tipo de recursividade produções $N \rightarrow N\alpha \mid \beta$ são transformadas em produções $N \rightarrow \beta N'$ onde $N' \rightarrow \alpha N' \mid \varepsilon$.

A seguinte gramática

$$\begin{aligned}
xp &\rightarrow \text{Exp} + \text{Exp}_2 \\
\text{Exp} &\rightarrow \text{Exp} - \text{Exp}_2 \\
\text{Exp} &\rightarrow \text{Exp}_2 \\
\text{Exp}_2 &\rightarrow \text{Exp}_2 * \text{Exp}_3 \\
\text{Exp}_2 &\rightarrow \text{Exp}_2 / \text{Exp}_3 \\
\text{Exp}_2 &\rightarrow \text{Exp}_3 \\
\text{Exp}_3 &\rightarrow \text{num} \\
\text{Exp}_3 &\rightarrow (\text{Exp})
\end{aligned}$$

$$\begin{array}{l}
\text{Exp} \longrightarrow \text{Exp}_2 \text{Exp}_* \\
\text{Exp}_* \longrightarrow +\text{Exp}_2 \text{Exp}_* \\
\text{Exp}_* \longrightarrow -\text{Exp}_2 \text{Exp}_* \\
\text{Exp}_* \longrightarrow \varepsilon \\
\text{Exp}_2 \longrightarrow \text{Exp}_3 \text{Exp}_{2*} \\
\text{Exp}_{2*} \longrightarrow *\text{Exp}_3 \text{Exp}_{2*} \\
\text{Exp}_{2*} \longrightarrow /\text{Exp}_3 \text{Exp}_{2*} \\
\text{Exp}_{2*} \longrightarrow \varepsilon \\
\text{Exp}_3 \longrightarrow \text{num} \\
\text{Exp}_3 \longrightarrow (\text{Exp})
\end{array}$$

Pode ser reescrita como

2.2.3.4 Recursividade indirecta à esquerda

Existem dois tipos de recursão indirecta

1. Existem produções recursivas à direita mutuamente

$$\begin{array}{l}
N_1 \longrightarrow N_2 \alpha_1 \\
N_2 \longrightarrow N_3 \alpha_2 \\
\vdots \\
N_{k-1} \longrightarrow N_k \alpha_{k-1} \\
N_k \longrightarrow N_1 \alpha_k
\end{array}$$

2. Existe uma produção $N \longrightarrow \alpha N \beta$ onde $\text{Nullable}(\alpha)$.
3. Uma combinação de 1. e 2.

Uma gramática é directa ou indirectamente recursiva à esquerda se existe uma sequência de derivação não vazia $N \Longrightarrow N\alpha$, ou seja, se um símbolo não terminal deriva uma sequência de símbolos gramaticais iniciados pelo mesmo símbolo não terminal [14]. Caso exista recursividade indirecta à esquerda, a gramática deve ser reescrita de forma a tornar a recursividade directa [14].

2.2.3.5 Factorização à esquerda

Se duas produções do mesmo símbolo terminal começam com a mesma sequência de símbolos, irá ocorrer **sobreposição de conjuntos FIRST** [14]. a gramática deve ser reescrita de forma a tornar essas produções em apenas uma contendo o **prefixo comum** e utilizando uma nova regra auxiliar.

Por exemplo, a gramática representando condicionais,

$$\text{if_cond} \longrightarrow \text{IF}(\text{exp}) \text{ cmd} \mid \text{IF}(\text{exp}) \text{ cmd} \text{ ELSE } \text{cmd}$$

pode ser factorizada na gramática

$$\begin{array}{l}
\text{if_cond} \longrightarrow \text{IF}(\text{exp}) \text{ cmd } \text{else_part} \\
\text{else_part} \longrightarrow \text{ELSE } \text{cmd} \mid \varepsilon
\end{array}$$

2.2.3.6 Construção de analisadores LL(1)

Aplicar os seguintes passos [14]:

1. Eliminar ambiguidade na gramática
2. Eliminar recursão à esquerda
3. Realizar factorização à esquerda, caso seja necessário
4. Adicionar uma nova inicial regra $S' \rightarrow S\$$ à gramática, sendo S' o novo símbolo inicial
5. Calcular FIRST para todas as produções
6. Calcular Follow para todos os símbolos não terminais
7. Para símbolos não terminais N e símbolo de entrada c escolher a produção $N \rightarrow \alpha$ quando:
 - (a) $c \in FIRST(\alpha)$ ou
 - (b) $Nullable(\alpha) \wedge c \in FOLLOW(N)$

Guardar esta escolha numa tabela para fácil aplicação do algoritmo.

2.2.4 Análise Bottom-Up Left-To-Right LR

Ficamos na página 97

Bibliografia

- [1] Appel Andrew. “**Modern compiler implementation in C**”. In: Cambridge University Press, 1998. Chap. 2 Lexical Analysis, pp. 16–17, 30–35.
- [2] Computer Hope. *Programming language*. 2019. URL: <https://www.computerhope.com/jargon/p/programming-language.htm>. (acedido pela última vez a 18 de dezembro de 2019).
- [3] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2.11.2 Table-Driven LL(1) Parsing.
- [4] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 1.2 Nondeterministic Finite Automata.
- [5] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 1.4 Deterministic Finite Automata.
- [6] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 1.7 Minimisation of DFAs.
- [7] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 1.8 Lexers and Lexer Generators.
- [8] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2 Syntax Analysis.
- [9] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2.5 Syntax Analysis.
- [10] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2.7 Nullable and FIRST.
- [11] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2.9 FOLLOW.
- [12] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2.11 LL(1) Parsing.
- [13] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2.11.3 Conflicts.
- [14] Torben Ægidius Mogense. “Introduction to Compiler Design”. In: Springer, 2011. Chap. 2.12 Rewriting a Grammar for LL(1) Parsing.