Capítulo 5

Análise Sintáctica Ascendente

Como já foi referido anteriormente os reconhecedores descendentes estão limitados às gramáticas que satisfzem a condição $LL(1)$, esta limitação faz com que os reconhecedores descendentes não possam ser utilizados em muitas situações.

Os reconhecedores ascendentes não têm esta limitação sendo a sua aplicabilidade muito maior. Podemos dizer que:

(+) são muito poderosos;

(−) são trabalhosos de implementar.

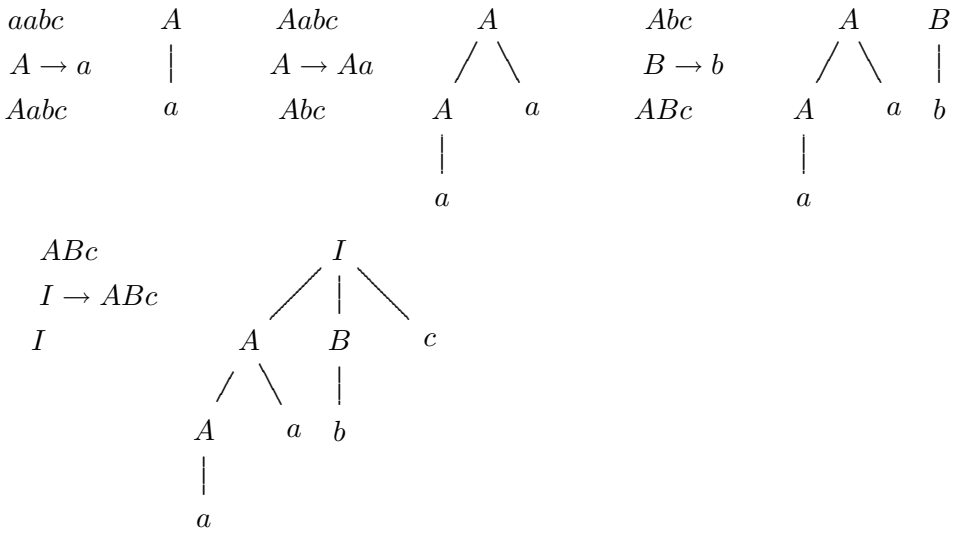
5.1 Princípios Gerais

Os reconhecedores ascendentes são assim designados dado que a árvore de reconhecimento é construída de forma ascendente, desde as folhas até à raiz. Para uma dada frase f o processo de construção da árvore de reconhecimento designa-se por *redução de* f.

A construção da árvore é feita através de uma sucessão de reduções em que, em cada uma delas, se substitue uma dada sub-frase que corresponde ao lado direito de um produção pelo correspondente lado esquerdo.

Exemplo 5.1 *Considere a seguinte gramática* $G = (\{a, b, c\}, \{I, A, B\}, I, P)$ *com* P *o seguinte conjunto de produções* $P = \{I \to ABc \mid B; A \to Aa \mid a; B \to b\}$. *Então a frase* $f = aabc$ *pode ser reduzida através da seguinte sequência de reduções:*

[1](Versão 1.14)



Esta árvore de derivação corresponde a seguinte derivação:

$$I \overset{I \to ABc}{\Longrightarrow} ABc \overset{B \to b}{\Longrightarrow} Abc \overset{A \to Aa}{\Longrightarrow} Aabc \overset{A \to a}{\Longrightarrow} aabc$$

é de notar que é feita uma derivação pela direita.

5.2 Análise Sintáctica Ascendente *LR*

O reconhecimento ascendente tal como foi indicado corresponde à inversão de uma derivação pela direita, daí vem a designação de reconhecedores LR.

L – «left scan» – leitura da esquerda para a direita.

R – «rightmost derivation» – derivação pela direita.

Esta estratégia de redução não está só por si isenta de problemas. Por exemplo para a frase anterior podiamos ter a seguinte árvore de derivação.

$aabc$	A	$Aabc$	A	A	$AAbc$	A	A	B
$A \to a$	$\vert$	$A \to a$	$\vert$	$\vert$	$B \to b$	$\vert$	$\vert$	$\vert$
$Aabc$	a	$AAbc$	a	a	$AABc$	a	a	b

não sendo possível acabar a construção da árvore de derivação. Assim como nas estratégias de reconhecimento descendentes interessa-nos considerar os casos em que não haja ambiguidade na construção da árvore de derivação, tal é possível desde que se escolha a sub-frase a derivar de forma apropriada.

Definição 5.1 (Redex) *Designa-se por* Redex *de uma frase, a uma sua sub-frase que está de acordo como o lado direito de uma produção, e cuja redução ao não terminal do lado esquerdo dessa produção é possível encadear numa derivação pela direita.*

Antes de vermos como é então possível de forma automática determinar tais sub-frases de forma a ser capaz de construir um AFD para o reconhecimento ascendente vejamos como tal técnica nos permitiria implementar uma reconhecedor.

Um modo eficiente e adequado de implementar um reconhecedor ascendente consiste na utilização de uma pilha para guardar os estados de reconhecimento. Vejamos para o caso anterior como se processaria o reconhecimento da frase f.

Temos os seguintes passos:

Desloca os símbolos de f para a pilha até encontrar um redex;

Reduz pela produção correspondente substituíndo na pilha os símbolos que constituêm o redex pelo lado esquerdo da produção;

Aceita/Erro: prosegue até chegar a uma situação de erro ou de reconhecimento da frase.

Pilha	Frase	Acção
	$aabc$	desloca
a	abc	reduz $(A \rightarrow a)$
A	abc	desloca
Aa	bc	reduz $(A \rightarrow Aa)$
A	bc	desloca
Ab	c	reduz $(B \rightarrow b)$
AB	c	desloca
ABc		reduz $(I \rightarrow ABc)$
I		aceita

Dado que as duas únicas acções, além das situações de erro e de aceitação, são as acções de deslocar um símbolo da frase para a pilha, e de reduzir uma sub-frase, é usual designar este tipo de reconhecedor por *reconhecedor desloca-reduz*.

Embora intuitivamente seja fácil determinar quais as acções a executar pelo reconhecedor é necessário ter um processo de automatizar essa escolha, tal automatização pode ser feita por introdução de um AFD. Os estados desse autómato vão conter a informação necessária para decidir entre as acções de desloca e reduz.

O AFD será representado através de duas tabelas:

- Tabela de Transições de Estado

$$TT = \underline{\texttt{tabela}}\ [Q, (T \cup N)]\ \underline{\texttt{de}}\ Q \cup \{\texttt{erro}\}$$

- Tabela de Acções

$$TA = \underline{\texttt{tabela}}\ [Q, T]\ \underline{\texttt{de}}\ P \cup \{\texttt{desloca,aceita,erro}\}$$

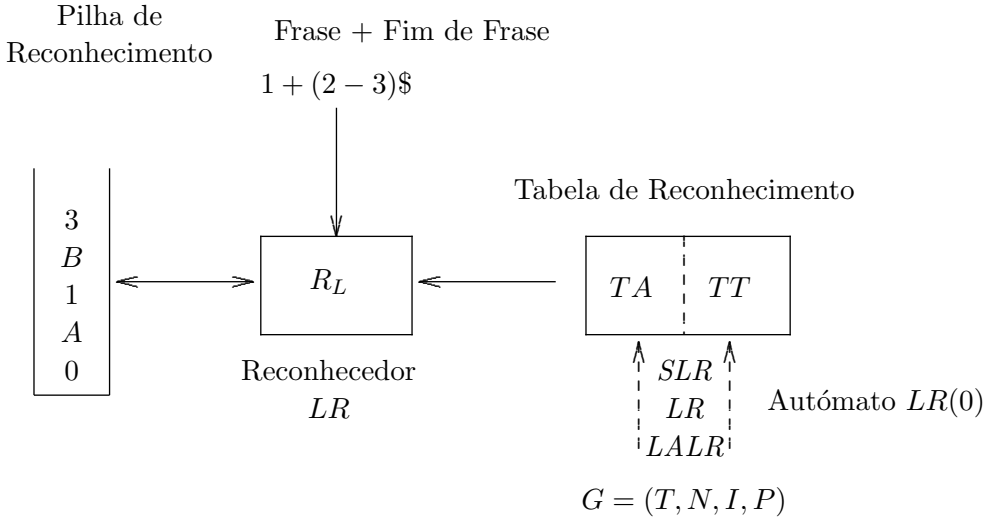
A construção destas tabelas pode ser feita de várias formas, sendo as três mais conhecidas as seguintes.

Método *SLR* (Simple LR) É o método mais fácil de implementar, é também o menos poderoso não sendo possível em alguns casos construir uma tabela de reconhecimento.

Método *LR* É o método originalmente proposto por D. Knuth, é muito poderoso, é também o mais complicado de implementar e o que necessita de mais recursos.

Método *LALR* (≪Lookahead≫ LR) É um método intermédio entre os dois anteriores tanto em poder expressivo como na dificuldade de implementação. O *Bison* gera reconhecedores $LALR$.

A estrutura genérica de um reconhecedor LR é a seguinte.



5.3 Construção de Tabelas *SLR*

A ideia central do método LR consiste na construção, a partir de uma dada Gramática Independente do Contexto, de um AFD, designado autómato $LR(0)$, cujos estados contêm a informação necessária para o reconhecedor decidir entre as acções *desloca* e *reduz*. Posteriormente representa-se esse autómato numa tabela, a Tabela de Transições, calculando-se de seguida a Tabela de Acções.

Façamos este estudo com o auxílio do tratamento de um caso concreto. Seja G uma gramática independente do contexto definida do seguinte modo:

$$G = (\{a, b\}, \{I, A\}, I, \{I \to Aa \mid b; A \to Aa \mid \varepsilon\})$$

Como o algoritmo de reconhecimento vai usar uma pilha auxiliar torna-se necessário extender a linguagem em questão com um novo símbolo que indique o fim da frase a reconhecer. A introdução do novo símbolo vai provocar outras alterações à gramática, a saber:

- introdução de um novo símbolo inicial I';
- introdução de um novo símbolo terminal \$, sendo este o símbolo que vai assinalar o fim da frase;
- introdução de uma nova produção, $I' \to I\$$.

Para o nosso exemplo teríamos:

$$G' = (\{a, b\}, \{I', I, A\}, I', \{I' \to I\$; I \to Aa \mid b; A \to Aa \mid \varepsilon\})$$

5.3.1 Construção da Tabela de Transições

A Tabela de Transições constroí-se a partir do AFD para a gramática que se está a considerar. Começa-se por construir um AFND o qual é depois convertido para AFD.

Vejamos então como construir o AFND $A = (V, Q, I, F, \delta)$ com:

- $V = N \cup T$;
- Q é um conjunto finito de estados cuja forma de construção vai ser discutida de seguida;
- I, o estado inicial;
- $F \in Q$, é o conjunto de estados terminais;
- δ é a função de transições de estado cuja definição vai ser discutida de seguida.

Os estados do autómato representam situações de reconhecimento, isto é, indicam o que já foi «visto» de uma frase num dado ponto do processo de reconhecimento. Os elementos de Q são então pares definidos do seguinte modo:

$$Q = \{(p, n) \in P \times \mathbb{N} \mid p = A \to \beta \in P \wedge n \leq |\beta| + 1\}$$

em geral utiliza-se uma notação abreviada colocando um meta-símbolo na posição de reconhecimento, isto é, na posição dada por $n \in \mathbb{N}$ coloca-se (por exemplo) um '.' (ponto).

Considerando a produção $p = I \to Aa$ temos os seguintes pares $(p, 1)$, $(p, 2)$, e $(p, 3)$, os quais usando a representação abreviada seriam representados por:

- $I \to .Aa$, espera-se encontrar na entrada uma frase derivável de Aa;
- $I \to A.a$, encontrou-se uma frase derivável de A e espera-se encontrar na entrada uma frase derivável de a;
- $I \to Aa.$, pode-se efectuar a redução.

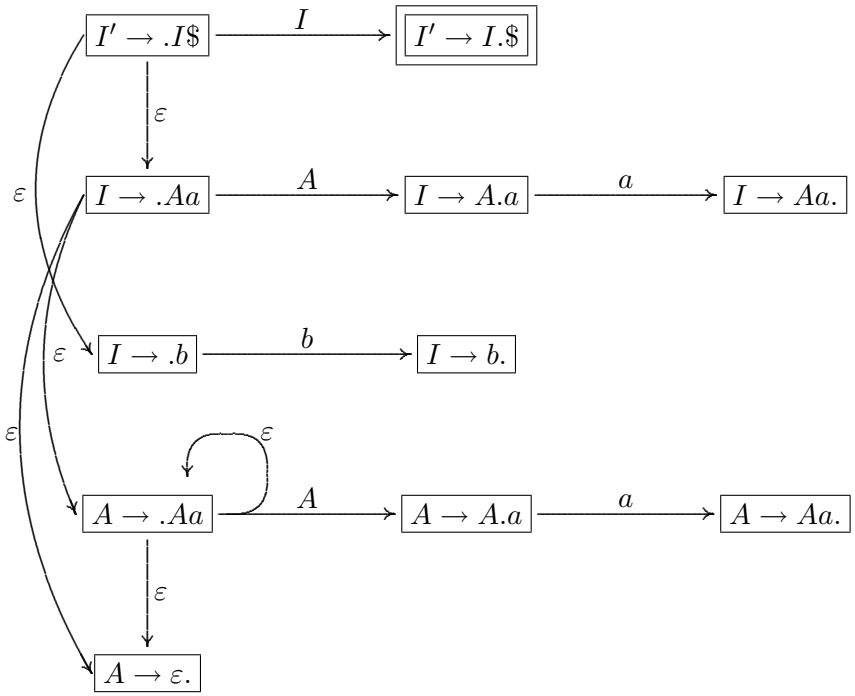
A função de transição ($\delta : (Q \times V) \to Q$) vai-nos descrever os diferentes estágios de reconhecimento sendo construída a partir dos estados tendo em conta as duas seguintes situações:

$$\boxed{A \to \alpha.X\beta} \xrightarrow{X} \boxed{A \to \alpha X.\beta}$$

$$\boxed{A \to \alpha.X\beta} \xrightarrow{\varepsilon} \boxed{X \to .\gamma}$$

- Espera-se reconhecer uma sub-frase derivável de $X\beta$, se se encontra uma sub-frase derivável de X então transita-se para um novo estado em que se espera reconhecer β.
- Espera-se reconhecer uma sub-frase derivável de $X\beta$, então se $X \to \gamma$ é um produção da gramática, pode-se também esperar ver uma sub-frase derivável de γ, transitando para esta nova regra através de uma transição ϵ.

Para o exemplo que se está a considerar, e aplicando as regras descritas acima, ter-se-ia o seguinte AFND.

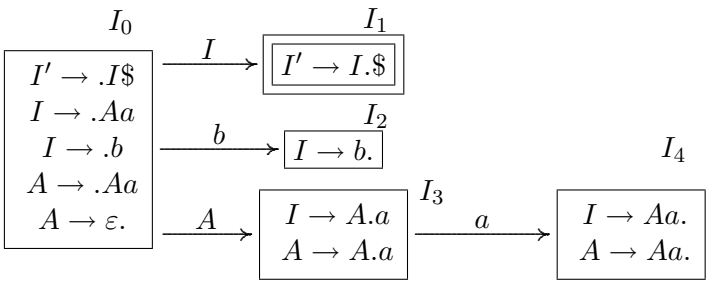


A conversão do AFND num AFD é feito eliminando as transições ε, o que é feito calculando o fecho-ε dos estados como já foi estudado no capítulo 3.

Para o exemplo que se está a estudar ter-se-ia:

$$
\begin{array}{lcl}
I_0 & = & \varepsilon - \text{fecho}([I' \to .I\$]) \\
 & = & [I' \to .I\$, I \to .Aa, I \to .b, A \to .Aa, A \to \varepsilon.] \\
I_1 = \delta'(I_0, I) & = & \varepsilon - \text{fecho}(\bigcup_{q \in I_0} \delta(q, I)) = [I' \to I.\$] \\
I_2 = \delta'(I_0, b) & = & \varepsilon - \text{fecho}(\bigcup_{q \in I_0} \delta(q, b)) = [I \to b.] \\
I_3 = \delta'(I_0, A) & = & \varepsilon - \text{fecho}(\bigcup_{q \in I_0} \delta(q, A)) = [I \to A.a, A \to A.a] \\
I_4 = \delta'(I_3, A) & = & \varepsilon - \text{fecho}(\bigcup_{q \in I_3} \delta(q, a)) = [I \to Aa., A \to Aa.]
\end{array}
$$

Graficamente temos:



Temos então a seguinte Tabela de Transições:

		$T \cup N$					
		a	b	\$	I'	I	A
Q	I_0	—	I_2	—	—	I_1	I_3
	I_1	—	—	—	—	—	—
	I_2	—	—	—	—	—	—
	I_3	I_4	—	—	—	—	—
	I_4	—	—	—	—	—	—

5.3.2 Construção da Tabela de Acções

A construção da Tabela de Acções é feita tendo em conta as seguintes regras (método *SLR*), nas quais se vai designar por $ta[i, j]$ o elemento de TA na linha i e coluna j:

- Se $A \rightarrow \alpha.a\beta \in I_i$ com $a \in T$, e $(<I_i, a>, I_j) \in \delta$ então $ta[i, a] =$ desloca$_j$;
- Se $A \rightarrow \alpha. \in I_i$ e $A \neq I'$ então para todo o $a \in$ Follow(A), tem-se que $ta[i, a] =$ reduz$_{A \rightarrow \alpha}$;
- Se $I' \rightarrow I.\$ \in I_i$ então $ta[i, \$] =$ aceita.

Todas as restantes entradas da tabela correspondem a terminações sem sucesso, isto é, situações de erro.

Continuando com o exemplo que temos vindo a tratar tem-se: Follow$(I) = \{\$\}$, Follow$(A) = \{a\}$ e fazendo (para simplificar) as seguintes associações: $r_1 = I' \rightarrow I\$$; $r_2 = I \rightarrow Aa$; $r_3 = I \rightarrow b$; $r_4 = A \rightarrow Aa$; $r_5 = A \rightarrow \epsilon$, temos então a seguinte tabela como Tabela de Acções (TA) e Tabela de Transições (TT) para o reconhecedor *SLR*:

		TA			TT					
		T			$T \cup N$					
		a	b	\$	a	b	\$	I'	I	A
Q	I_0	r_5	d_2	—	—	I_2	—	—	I_1	I_3
	I_1	—	—	aceita	—	—	—	—	—	—
	I_2	—	—	r_3	—	—	—	—	—	—
	I_3	d_4	—	—	I_4	—	—	—	—	—
	I_4	r_4	—	r_2	—	—	—	—	—	—

5.3.3 Construção de um Reconhecedor *LR*

A construção de um Reconhecedor *LR* passa por implementar a estruturas auxiliares necessárias para guardar a informação referente às tabelas de transições e de acções, à pilha auxiliar e ao canal de entrada.

No canal de entrada o reconhecedor têm uma sequência de palavras:

$$t_i t_{i+1} \dots t_n\$$$

Na pilha de reconhecimento são armazenadas as seguintes informações

$$\boxed{q_0 X_1 q_1 X_2 \dots X_m q_m} \leftarrow \text{topo}$$

com: q_i — estados do autómato
X_i — símbolos de gramática

O autómato usa o estado que está no topo da pilha e a próxima palavra a ser reconhecida como forma de indexar a tabela de reconhecimento e desta forma decidir qual a acção a tomar.

Para cada uma das acções possíveis (desloca, reduz, aceita, erro) é necessário efectuar um dado conjunto de passos.

- Se $TA[q_m, t_i] = \text{desloca}_j$ então o reconhecedor efectua um deslocamento da palavra de entrada t_i e do estado $TT[q_m, t_i] = q$ para a pilha.

  pilha
  $$\boxed{q_0 X_1 q_1 X_2 \dots X_m q_m t_i q}$$

  entrada
  $$t_{i+1} \dots t_n\$$$

- Se $TA[q_m, t_i] = r_i$, com $r_i = X \rightarrow \beta$, então efectua-se a redução pela regra correspondente, como $|\beta| = n$ retiram-se $2n$ símbolos da pilha (n estados e n símbolos), colocando-se de seguida o símbolo do lado esquerdo da produção r_i, e um novo estado que se obtêm da tabela de transições através do estado que ficou no topo da pilha q_{m-n} e do novo símbolo a introduzir X. Supondo $TT[q_{m-n}, X] = q$ tem-se:

  pilha
  $$\boxed{q_0 X_1 q_1 X_2 \dots X_m q_{m-n} X q}$$

  entrada não é alterada.

- Se $TA[q_m, t_i] =$ aceita, então o reconhecedor termina com sucesso.

- Se $TA[q_m, t_i] =$ erro, então o reconhecedor termina com erro.

Inicialmente a pilha tem somente o estado inicial do AFD e a entrada tem a frase a reconhecer terminada com \$.

Retomando o exemplo pretende-se verificar que $f = aaa \in L_G$.

Pilha	Entrada	Acções
I_0	$aaa\$$	$TA[I_0, a] = r_5,\ r_5 = A \to \varepsilon,\ \lvert\varepsilon\rvert = 0,$ $TT[I_0, A] = I_3$
I_0AI_3	$aaa\$$	$TA[I_3, a] = d_4,\ TT[I_3, a] = I_4$
$I_0AI_3aI_4$	$aa\$$	$TA[I_4, a] = r_4,\ r_4 = A \to Aa,\ \lvert Aa\rvert = 2,$ $TT[I_0, A] = I_3$
I_0AI_3	$aa\$$	$TA[I_3, a] = d_4,\ TT[I_3, a] = I_4$
$I_0AI_3aI_4$	$a\$$	$TA[I_4, a] = r_4,\ r_4 = A \to Aa,\ \lvert Aa\rvert = 2,$ $TT[I_0, A] = I_3$
I_0AI_3	$a\$$	$TA[I_3, a] = d_4,\ TT[I_3, a] = I_4$
$I_0AI_3aI_4$	$\$$	$TA[I_4, \$] = r_2,\ r_2 = I \to Aa,\ \lvert Aa\rvert = 2,$ $TT[I_0, I] = I_1$
I_0II_1	$\$$	$TA[I_1, \$] =$ aceita

O reconhecimento foi feito com sucesso.

O algoritmo reconhecimento é então (ver Apêndice B):

```
bool função LR(tabela TA,TT) {
  simb ← ler();
  p ← push(I,vazia);
  repete
     ac ← TA[top(p),simb];
     se (ac=desloca) {
        q ← TT[top(p),simb];
        p ← push(simb,p);
        p ← push(q,p);
        simb ← ler();
     }
     se (ac=A → β) {
        para (i=0;i<2*comprimento(β);i++)
           p ← pop(p);
        q ← TT[top(p),A];
        p ← push(A,p);
        p ← push(q,p);
     }
  até (ac∈{aceita,erro});
  LR ← (ac=aceita)
}
```

5.3.4 Conflitos no Reconhecimento *SLR*

Os reconhecedores *SLR* são bastante poderosos, no entanto existem gramáticas para as quais não é possível decidir qual a acção a executar.

Há duas situações em que tal bloqueio ocorre:

Conflito transição/redução: não é possível decidir entre uma acção de redução e uma acção de deslocação.

Conflito redução/redução: existe mais do que uma hipótese para efectuar uma redução.

Para uma dada gramática podemos verificar se tal acontece, ou não.

Definição 5.2 (Condição SLR(1)) *Uma gramática $G = (T, N, I, P)$ satisfaz a condição SLR(1) se:*

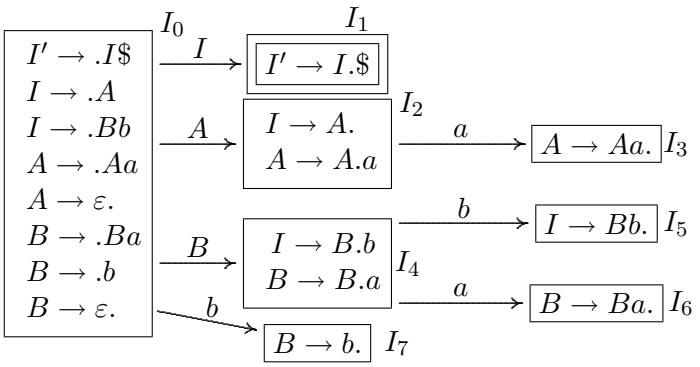
- *para cada produção $A \to \alpha.x\beta \in P$, com $x \in T$, não existe nenhuma produção $B \to \gamma.$ com $x \in \text{Follow}(B)$.*
- *para cada par de produções $A \to \alpha.$ e $B \to \beta.$ em P verifica-se que:* $\text{Follow}(A) \bigcap \text{Follow}(B) = \emptyset$.

A primeira condição refere-se aos conflitos *transição/redução*, a segunda aos conflitos *redução/redução*.

Considere-se a gramática $G = (\{a, b\}, \{I, A, B\}, I, P)$, com P o seguinte conjunto de produções:

$$\begin{array}{rcl} I & \to & A \mid Bb \\ A & \to & Aa \mid \varepsilon \\ B & \to & Ba \mid b \mid \varepsilon \end{array}$$

a partir desta gramática constroí-se o seguinte autómato:



Na construção da tabela de acções segundo o método *SLR* verificam-se situações de conflito, note-se que $\text{Follow}(I) = \{\$\}$, $\text{Follow}(A) = \{a, \$\}$, $\text{Follow}(B) = \{a, b\}$.

		TA			TT						
		T			$T \cup N$						
		a	b	\$	a	b	\$	I'	I	A	B
Q	I_0	$A \to \varepsilon.$ $B \to \varepsilon.$	$B \to \varepsilon.$ d_7	—	—	I_7	—	—	I_1	I_2	I_4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Tem-se um conflito redução/redução no caso $TA[I_0, a] = \text{reduz}_{A\to\varepsilon}$ ou $\text{reduz}_{B\to\varepsilon}$ e tem-se um conflito redução/deslocação no caso de $TA[I_0, b] = \text{reduz}_{B\to\varepsilon}$ ou *desloca (d)*.

Estas situações de conflito podem ser evitadas se se recorrer aos outros métodos referidos.

5.4 *Bison*

O *Bison* (Donnelly and Stallman(1995)) é um programa capaz de criar um reconhecedor LALR(1) a partir de uma especificação em formato próprio.

Os passos necessários para criar um reconhecedor para uma dada gramática G são os seguintes:

1. Escrever uma especificação *Bison* capaz de descrever a gramática G, o ficheiro que contém a especificação tem extensão «.y» (por convenção);
2. Escrever um analisador léxico capaz de transformar sequências de caracteres em sequências de palavras (redex) que serão os objectos de entrada do reconhecedor produzido pelo *Bison*. O nome da rotina segue as convenções do *Flex*, como tal o seu nome tem de ser `yylex`;
3. Escrever uma função que faça a chamada da função de reconhecimento;
4. Escrever uma função que faça o tratamento de erros;

Após estes processo de construção das componentes necessárias à construção de um reconhecedor torna-se necessário processá-las de forma a criar um executável. Para isso temos os seguintes passos:

1. Processar a especificação através do *Bison*:

   `> bison` <*nome*>`.y`

   como resultado deste processamento temos o ficheiro <*nome*>`.tab.c`.
2. Compilar os ficheiros fonte através de um compilador de *C*.
3. Criar o executável por junção dos módulos já compilados com os módulos do sistema («linkar»).

O ficheiro <*nome*>`.tab.c` contém a rotina de reconhecimento `yyparse()` a qual chama a rotina `yylex()` sempre que necessita de um novo redex. Para produzir um reconhecedor é necessário escrever as rotinas `main`, que tem a função habitual de ponto de partida, e no qual se faz a chamada à rotina de reconhecimento, assim como a rotina `yyerror` que faz o tratamento de erros. Em apêndice (apêndice C) é apresentado um pequeno exemplo de utilização conjunta dos programas *Flex* e *Bison*.

5.4.1 Especificações *Bison*

Uma especificação *Bison* descreve uma gramática livre do contexto com um símbolo antecipável, $G = (T, N, I, P)$, sendo a sua forma genérica a seguinte:

```
%{
zona de declarações C
%}
zona de declarações Bison
%%
zona das produções
%%
zona das rotinas em C
```

Zona das Declarações: a zona de declarações vai conter por uma parte declarações em *C*, e por outra parte declarações próprias da especificação da gramática.

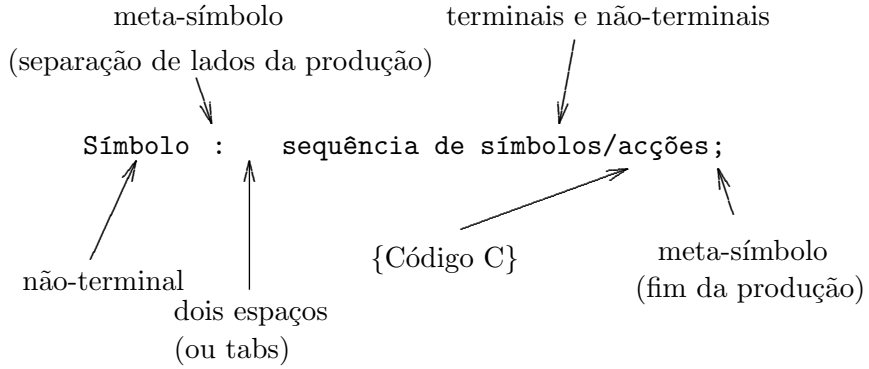
A zona de declarações *C* pode conter declarações para o pré-processador de *C*, assim como declarações de variáveis, tipos, e funções que são depois usados nas acções da gramática. Este código vai ser transcrito para o início do ficheiro que contém o reconhecedor, sendo opcional.

A zona das declarações *Bison* utiliza um conjunto de meta-palavras (Donnelly and Stallman(1995), pags 50-57) para definir as palavras reservadas da linguagem a descrever, assim como as suas características, temos assim:

`%token <nome>`	declara o nome de uma palavra reservada (símbolo terminal).
`%left <nome>`	declara um operador associativo à esquerda.
`%right <nome>`	declara um operador associativo à direita.
`%nonassoc <nome>`	declara um operador não-associativo.
`%union`	declara um conjunto de possíveis tipos para os valores semânticos.
`%type`	declara o tipo de um terminal.
`%start`	declara o símbolo inicial. Por omissão é o símbolo do lado esquerdo da primeira produção.
`%prec`	atribue uma precedência ao operador. Quando omisso a precedência dos operadores é determinada pela ordem das declarações dos diferentes operadores.

As declarações referentes à associatividade dos operadores servem também como forma de declaração de um nome (símbolo terminal), sendo por isso alternativas à declaração `%token`.

Zona das Produções: na zona das produções especificação-se produções da gramática. A sua forma é a seguinte:



ou

```
Símbolo :     sequência de símbolos/acções
              | sequência de símbolos/acções
              ...
              ;
```

meta-símbolo (alternativa)

Temos ainda que:

- da mesma forma que para a escrita das produções da gramática é possível apresentar várias produções alternativas (para um mesmo símbolo não-terminal) através do meta-símbolo «`|`»;

- os nomes dos símbolos podem ter um qualquer comprimento, e podem consistir de letras números ou os símbolos «`.`», ou «`_`», sendo que é necessário que comecem por uma letra;

- maiúsculas e minúsculas são diferentes;

- não-terminais em minúsculas, por convenção;

- terminais em maiúsculas, por convenção. Os terminais podem ser denotados de três formas diferentes (Donnelly and Stallman(1995), pag. 41):

  - através de um identificador previamente declarado através de uma declaração `%token` (ou similar). O valor numérico de identificação é atribuído automaticamente;
  - através de um caracter entre plicas, por exemplo `'+'`. O valor numérico de identificação é dado pela posição do caracter na tabale *ASCII*;
  - através de uma constante do tipo sequência de caracteres, por exemplo `''<=''`. O valor numérico de identificação é guardado na tabela `yytname`, a qual é gerada automáticamente pelo *Bison* sempre que se inclua a directiva `%token_table`.

- se não for possível reconhecer a sequência de palavras dada no canal de entrada, a mensagem «syntax error» é enviada automaticamente, fica a cargo do utilizador escrever a rotina de tratamento de erros `yyerror()`. O símbolo terminal `error` é pré-definido e é reservado pelo *Bison* para a especificação de produções especiais para o tratamento de erros de sintácticos;

- o *Bison* providência um conjunto de pseudo-variáveis «`$n`» as quais nos permitem obter os vários elementos da expressão que foi reconhecida. A pseudo-variável «`$$`» devolve o valor da acção.

Zona das rotinas em *C*: todo o texto que for escrito nesta secção é copiado literalmente para o fim do ficheiro que contém o reconhecedor. Este é o lugar mais conveniente para colocar todo o código *C* que não necessita de vir antes da rotina `yyparse`, nomeadamente é este o lugar indicado para colocar as rotinas `yyerror` e `main`.

5.4.2 Especificação das Produções

As produções da gramática são transcritas para o *Bison* da forma que vimos acima, o lado esquerdo contém um não-terminal, o meta-símbolo «:» faz a separação entre o lado esquerdo e o lado direito, sendo que este último é constituído por uma sequência de símbolos (terminais e não-terminais) e de acções (código *C*) separados entre si por um ou mais espaços. Embora as acções e os símbolos possam aparecer entremeados entre si normalmente só existe uma dada acção a qual é escrita no fim da produção, antes do meta-símbolo «;» que define o fim da produção.

Tipo dos valores semânticos O *Bison* implementa um autómato finito reactivo sendo as acções semânticas implementadas através de código *C*, a ligação entre a componente sintáctica e a componente semântica é feita através das pseudo-variáveis.

Por omissão o tipo `int` é usado para todos os valores semânticos sendo que o valor semântico de um dado símbolo terminal pode ser definido de forma explícita através da atribuição de um dado valor à variável global `yylval`.

Por exemplo:

```
[0-9]+   sscanf(yytext,"%d",&yylval); return(NUM);
```

atribue o valor inteiro do número que acabou de ser reconhecido (pela rotina `yylex`) ao símbolo terminal `NUM`.

```
.|\n     return yytext[0];
```

determina que para todos os outros casos o símbolo terminal é representado directamente pelo caracter *ASCII* reconhecido. Neste caso não se atribuiu nenhum valor semântico.

Podemos redefinir o tipo único dos valores semânticos através da declaração desse valor através do nome `YYSTYPE`, basta incluir:

```
#define YYSTYPE <tipo>
```

na zona de declarações *C* da especificação *Bison*.

É ainda possível definir um conjunto de diferentes valores semânticos para diferentes símbolos. Por exemplo, para uma constante numérica um dado tipo numérico, para uma constante do tipo sequência de caracteres o tipo `char *`, ou outros tipos conforme as necessidades.

Para poder definir mais do que um tipo para os valores semânticos é necessário:

- declarar os diferentes tipos semânticos através da declaração `%union`, muito a exemplo do que se faz em *C*;

- declarar o tipo de cada um dos símbolos da gramática de forma explícita.

A declaração explícita do tipo implica, no caso dos símbolos terminais a inclusão do tipo nas declarações `%token` (ou similares), por exemplo `%token <int> NUM`, no caso dos símbolos não-terminais ter-se-á de explicitar o seu tipo através da declaração `%type`. Por exemplo:

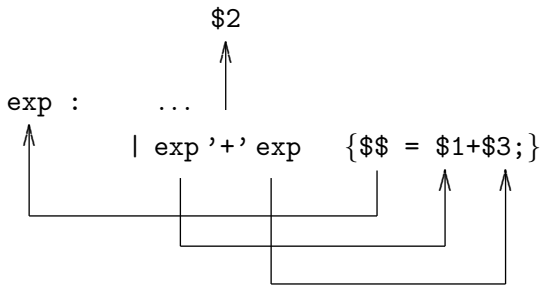
```
%union {
  double val;
  simbolo *tptr;
}

%token <val> NUM
%token <tptr> ID
%type <val> exp
```

Acções Semânticas As acções contém código *C* que é executado sempre que uma instância da produção é encontrada. Não vamos aqui tratar das acções que ocorrem entre os símbolos das produções (Donnelly and Stallman(1995), pags 48-50) considerando-se só o caso da existência de uma só acção escrita logo após o fim da sequência de símbolos.

As acções podem conter o código *C* que se desejar, a forma de comunicação entre as produções, isto é a fase do reconhecimento sintáctico, e as acções, isto é o código *C* é feito através das pseudo-variáveis `$n`, as quais nos permitem aceder ao valor semântico da n-ésima componente do lado esquerdo da produção, e pela pseudo-variável `$$`, a qual nos permite atribuir um dado valor semântico ao resultado (lado direito) da produção.

Por exemplo:



Por omissão o valor de uma dada produção é dado pelo valor semântico da primeira componente, isto é `$$=$1`.

Caso seja necessário pode-se usar a sintaxe `$<tipo>n` para, de forma explícita, definir o tipo de uma dada pseudo-variável.

5.4.3 Os Reconhecedores Léxico e Sintáctico

O reconhecedor produzido pelo *Bison* tem o seu ponto de entrada definido pela rotina `yyparse`, é esta a rotina que se tem de chamar sempre que se pretende usar o reconhecedor.

A rotina `yyparse` espera como entrada uma sequência de redexes e tem como saída 0 caso o reconhecimento tenha sido feito com sucesso, ou 1 caso surjam erros sintácticos. É possível interromper o normal funcionamento do reconhecedor e provocar a saída do mesmo seja com o valor 0 ou com o valor 1, através da utilização das macros `YYACCEPT` e `YYABORT` respectivamente.

O que foi dito atrás acerca da informação de entrada implica que a rotina `yyparse` está à espera que haja uma outra rotina que faça o reconhecimento lexical prévio transformando a sequência de caracteres presente no canal de entrada numa sequência de redexes. A convenção de chamada do *Bison* implica que essa rotina se chame `yylex` a qual é chamada pela rotina `yyparse` sempre que esta necessitar de um novo redex.

A rotina `yylex` pode ser incluída directamente na secção de rotinas *C* do ficheiro da especificação *Bison*, ou pode estar num ficheiro separado. Neste último caso torna-se necessário encontrar um dado mecanismo que permita à rotina `yylex` aceder à informação que diz respeito aos redexes (nome e valores semânticos), isto é feito através da utilização da opção de compilação `-d` do *Bison*, por exemplo:

```
> bison -d <nome>.y
```

a opção `-d` tem como efeito a inclusão de todas as declarações necessárias às duas rotina num ficheiro de cabeçalhos (≪header file≫) separado, isto é os ficheiros produzidos pelo *Bison* passam a ser `<nome>.tab.c` e `<nome>.tab.h`.

Após isso basta fazer a inclusão deste ficheiro no ficheiro que contém a rotina `yylex` através do comando apropriado, isto é:

```
#include <nome>.tab.h
```

Comunicação entre as rotina `yyparse` e `yylex` O valor devolvido pela rotina `yylex` deve ser o código numéricos do redex que acabou de ser reconhecido, ou 0 para assinalar o fim da entrada.

Quando foi definido um nome para o redex reconhecido esse nome pode ser usado na instrução de retorno. Quando o redex reconhecido é constituído por um só caracter esse mesmo caracter pode ser usado na instrução de retorno. Os valores semânticos associados aos redexes são passados através da variável `yylval`:

- se só se usar um tipo para todos os valores semânticos então basta atribuir o valor que se pretende à variável `yylval`;

- se há vários tipos possíveis para os valores semânticos então é necessário usar o nome qualificado apropriado.

Por exemplo:

- Um só tipo para os valores semânticos:

```
yylval = atoi(yytext);
return NUM;
```

- vários tipos possíveis para os valores semânticos:

```
%union {
  int ival;
  double dval;
}

yylval.ival = atoi(yytex);
return NUM;
```

5.4.4 Tratamento de Erros

O reconhecedor detecta uma erro sintáctico sempre que receber um redex que não satisfaça as regras gramaticais que foram especificadas. Quando tal acontece o comportamento do reconhecedor é o seguinte, chama a rotina `yyerror` eventualmente com uma sequência de caracteres como argumento, se este tiver omisso a mensagem de erro gerada será ≪parse error≫. A rotina `yyparse` tem de ser escrita explicitamente pelo programador sendo que para um reconhecedor simples a seguinte definição chega:

```
yyerror(s)
  char *s;
{
  fprintf(stderr,"%s\n",s);
}
```

Após o completar de `yyerror` o controlo é devolvido à rotina `yyparse` sendo tentado de imediato a recuperação do erro (Donnelly and Stallman(1995), pags 79-81), se tal não for possível o fluxo do programa é interrompido sendo 1 o valor de retorno.

Recuperação de erros Para evitar o fim prematuro do processo de reconhecimento sintáctico o programador deve tentar escrever rotinas de recuperação de erros, para tal o *Bison* possuí uma palavra reservada `error` a qual pode ser incluída em qualquer produção como uma outra opção, por exemplo:

```
linha :    '\n'
         | exp '\n'  { printf("\t%g\n",$1); }
         | error '\n' { yyerrok; }
         ;
```

Esta nova opção para a produção referente ao símbolo não-terminal `linha` permite uma recuperação de erro simples.

Na ocorrência de um erro sintáctico a rotina `yyerror` é chamada sendo que após a sua execução, quando o controlo é devolvido à rotina `yyparse` a produção `error` (existindo) é escolhida e a acção correspondente é executada, no exemplo apresentado a macro pré-definida `yyerrok` é executada sendo que o seu efeito é o de permitir o continuar do processo de reconhecimento. O programador pode implementar estratégias de recuperação de erro mais sofisticadas, não vamos no entanto tratar esse assunto aqui, veja-se (Donnelly and Stallman(1995), pags 79-81) para um tratamento mais aprofundado.

5.4.5 Alguns exemplos

Utilização do *Bison* isoladamente. Pretende-se construir um reconhecedor capaz de ler linhas contendo um só dígito. Dado a gramática:

$$G = (\{0,1,2,3,4,5,6,7,8,9,'\backslash n'\}, \{\text{linhas}, \text{linha}, \text{digito}\}, \text{linhas}, P)$$

com P o seguinte conjunto de produções:

$$\begin{array}{rcl} \text{linhas} & \rightarrow & \varepsilon \mid \text{linhas linha} \\ \text{linha} & \rightarrow & \text{digito } '\backslash n' \\ \text{digito} & \rightarrow & 0 \mid 1 \mid \ldots \mid 9 \end{array}$$

O ficheiro *Bison* correspondente vai implementar a gramática, assim como um reconhecedor lexical o qual, por convenção, se chama `yylex()`. Para se poder construir um programa autónomo é necessário construir a rotina `main`, a qual chama a rotina de reconhecimento `yyparse()`, assim como a rotina de tratamento de erros `yyerror`. Após isso, para se obter um executável bastaria fazer:

```
> bison cap5-exemplo1.y
> gcc cap5-exemplo1.tab.c -o cap5-exemplo1
```

```
%{
#include <ctype.h>
%}

%token DIGITO
```

```
%%
linhas :            /* vazia */
       | linhas linha       { printf("%d\n",$2); }
       ;

linha  : DIGITO '\n'   { $$ = $1; }
       ;

%%

int main(void) {
  return yyparse();
}

int yyerror(s)
char *s;
{
  printf("%s\n",s);
}

yylex(){
        int c;
        c=getchar();
        if (isdigit(c)) {
                yylval = c-'0';
                return DIGITO;
        }
        return c;
}
```

Utilização do *Bison* e do *Flex* em conjunto. Pretende-se construir um reconhecedor capaz de ler linhas contendo um natural. Dado a gramática:

$$G = (\{0,1,2,3,4,5,6,7,8,9,'\backslash n'\}, \{\text{linhas}, \text{linha}, \text{natural}\}, \text{linhas}, P)$$

com P o seguinte conjunto de produções:

$$\begin{array}{rcl} \text{linhas} & \rightarrow & \varepsilon \mid \text{linhas linha} \\ \text{linha} & \rightarrow & \text{natural } '\backslash n' \\ \text{natural} & \rightarrow & (0 \mid 1 \mid \ldots \mid 9)+ \end{array}$$

O reconhecedor léxico será construído através da utilização do *Flex*, ficheiro `cap5-exemplo2.l`.

```
%{
#include "cap5-exemplo2.tab.h"
%}

%%
[0-9]+      {
             sscanf(yytext,"%d",&yylval);
             return(NATURAL);
            }
\n          return('\n');
```

Duas notas: a inclusão de um ficheiro (`cap5-exemplo2.tab.h`), o qual ver-se-á de seguida, é gerado pelo *Bison*; a utilização de um identificador que não é definido neste ficheiro, mas que está definido no ficheiro que é incluído.

Vejamos o ficheiro `cap5-exemplo2.tab.h`

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NATURAL 257


extern YYSTYPE yylval;
```

De seguida vejamos o reconhecedor sintáctico, ficheiro `cap5-exemplo2.y`.

```
%{
#include <stdio.h>
%}

%token NATURAL

%%
linhas:
        |       linhas linha { printf("-> %d\n",$2);}
                ;

linha:          NATURAL '\n' {$$ = $1;}
                ;

%%

main()
```

```
{
  yyparse();
}

int yyerror(s)
 char *s;
{
 printf("%s\n",s);
};
```

Para obter o reconhecer ter-se-ia de fazer o seguinte:

```
> bison -d cap5-exemplo2.y
```

obtêm-se o reconhecedor sintáctico (`cap5-exemplo2.tab.c`), assim como (opção `-d`) o ficheiro `cap5-exemplo2.tab.h` com as definições necessárias ao *Flex*.

De seguida far-se-ia:

```
> flex cap5-exemplo2.l
```

para se obter o reconhecedor léxico, ficheiro `lex.yy.c`.

Finalmente compilam-se todos os programas para obter um executável.

```
> gcc lex.yy.c cap5-exemplo2.tab.c -lfl -o cap5-exemplo2
```

a biblioteca `fl`, refere-se ao `Flex` e é necessária para providenciar de forma automática a rotina `yywrap`.