Universidade de Coimbra

Departamento de Engenharia Informática

Compiladores

Resumo da matéria para exame

by **PascalMasters**

Autor:
António Carlos Lima
2011166926

Autor:
Inês Lopes Petronilho
2012137900

19 de Junho de 2015

Conteúdo

1 Advertência Inicial

WARNING

Os autores não se responsabilizam por qualquer infurtuito causado pelos erros que esteja presentes no seguinte documento. O mesmo deve ser lido sempre com pensamento crítico e analisado com cautela.

Para não começarmos esta sessão de estudo com um tom tão sério aqui vai uma piada: *In order to understand recursion, one must first understand recursion.*

Pronto, já deu para ver que não temos jeito para a comédia, por isso, sem mais demoras, de agora em diante só falaremos da matéria de Compiladores.

2 Parsing Descendente - Analisador Predictivo

2.1 Gramáticas e Produções

Uma gramática é um conjunto de produções da forma $X \rightarrow \gamma$ sendo que **X** representa qualquer símbolo não-terminal e γ representa qualquer sequência de símbolos terminais e não terminais. Por sua vez, ϵ representa o conjunto vazio.

2.2 Nullable, First e Follow

Tomemos as seguinte gramática para efeito de exemplo:

$$S \rightarrow Re \mid PQd$$
$$R \rightarrow c$$
$$Q \rightarrow b \mid \epsilon$$
$$P \rightarrow a \mid Q$$

2.2.1 Nullable

- Nullable(γ) significa que γ pode derivar ϵ
- $\neg$Nullable(γ) significa que γ **não** pode derivar ϵ

Por definição, Q e P são Nullable, ao passo que S e R não são.

2.2.2 First

First(X) é o conjunto de todos os primeiros símbolos terminais que X pode gerar imediatamente.

Caso Nullable(X) e $\neg$Nullable(Y) então First(XYZ) = First(X) $\cup$ First(Y)

Assim,
First(R) = {c}
First(Q) = {b}
First(P) = {a} $\cup$ First(Q) = {ab}

First(S) = First(Re) $\cup$ First(PQd), embora R não seja Nullable P e Q são
First(S) = First(R) $\cup$ First(P) $\cup$ First(Q) $\cup$ First(d)
First(S) = {c} $\cup$ {ab} $\cup$ {b} $\cup$ {d}
First(S) = {abcd}

2.2.3 Follow

Follow(X) é o conjunto de todos os símbolos terminais que se podem seguir a X numa derivação.

Por conseguinte, se o o primeiro X for Nullable então temos de seguir o Follow(Y) para Y pertencente ao conjunto de símbolos não-terminais que possam derivar X.

De forma implícita, o símbolo inicial da gramática tem como follow o caracter $ representativo do **End Of File**.

Assim,
Follow(R) = First(e) = {e}

Follow(Q) = First(d) ∪ Follow(P) = {bd}, Follow(P) pois Q é Nullable
Follow(P) = First(Qd) = First(Q) ∪ First(d) = {bd}
Follow(S) = $

2.3 Tabela de Parsing Predictivo

Podemos condensar toda esta informação quanto à gramática numa simples tabela:

Símbolo não-terminal	**Nullable**	**First**	**Follow**
S	não	abcd	$
R	não	c	e
Q	sim	b	bd
P	sim	ab	bd

A tabela anterior torna-se especialmente útil para criarmos então a Tabela de Parsing Predictivo que, embora tenha um nome pomposo, serve para determinar se uma determinada gramática pode ser intrepretada por um parser **LL(1)** (Left to right, Leftmost derivation, 1 symbol lookahead).

Regras de construção:

- Existe uma linha para cada símbolo não-terminal
- Existe uma coluna para cada símbolo terminal
- Se $\neg Nullable(\gamma)$ então escreve-se $X \to \gamma$ na célula (X,T) para cada $T \in First(\gamma)$
- Se $Nullable(\gamma)$ então escreve-se $X \to \gamma$ na célula (X,T) para cada $T \in Follow(X)$

Só para garantir que estamos na mesma página de entendimento que o leitor, imaginemos que estamos a começar a preencher a tabel com a gramática anterior. A produção $S \to PQd$ segundo a notação $X \to \gamma$ indica que **X é S** e que γ **é PQd**. Ora, PQd nunca pode ser Nullable pois, mesmo que P e Q o sejam d, um símbolo terminal, não o será, temos então de efectuar o First(γ), ou seja, First(PQd). Mas imaginemos que PQd era Nullable para efeitos de explicação, assim teriamos seguir o Follow(X), ou seja, Follow(S).

Posto isto, para a gramática que temos vindo a utilizar e com o auxílio da tabela anterior obtemos a seguinte Tabela de Parsing Predictivo:

	a	**b**	**c**	**d**	**e**	**$**
S	$S \to PQd$	$S \to PQd$	$S \to Re$	$S \to PQd$		
R			$R \to c$			
Q		$Q \to \epsilon$ $Q \to b$		$Q \to \epsilon$		
P	$P \to a$	$P \to Q$		$P \to Q$		

Um Parser do tipo LL(1), também conhecido como Parser Descendente Recursivo, começa na raíz da Parse Tree expande-se em direcção às folhas escolhendo as produções que devem ser utilizadas com base nos tokens que são lidos. Ou seja, se for lido um token **a** e estivermos no estado **S** sabemos que devemos utilizar a produção $S \to PQd$. No entanto, podemos verificar que existe uma célula **(Q,b)** para a qual um Parser do tipo LL(1) não consegue determinar qual a acção que deve ser tomada pois existe mais do que uma produção que pode ser escolhida.

2.4 Ambiguidade, Recursividade e Factorização

Caso a gramática tenha falhado o teste da tabela anterior então podemos afirmar com certeza que ela não é uma LL(1), mas pode vir a ser! Para isso necessitamos "apenas"de inspeccionar a mesma e remover quaisquer ambiguidades, recursividades à esquerda e até mesmo factorizar produções que gerem ambiguidade. Como ponto de partida, Devemos ter em atenção as regras e símbolos que geraram os conflitos na tabela de parsing predictivo.

Vamos analisar a gramática de uma simples calculadora:

2.4.1 Ambiguidade

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow E * E$$
$$E \rightarrow E/E$$
$$E \rightarrow num$$
$$E \rightarrow (E)$$

O leitor atento notará que para esta calculadora não existe precedência de operações, ou seja, elas são computadas pela ordem com que forem analisadas. Isto não é benéfico, como se pode imaginar pois no caso da expressão $2 - 3 * 4$ pretendemos que seja interpretada como $2 - (3 * 4)$ por forma a obtermos -10 e não -4 $((2 - 3) * 4)$. No fundo, o que queremos especificar então é que as operações de multiplicação e divisão devem ser colocadas em níveis inferiores da **árvore de parsing** para que sejam calculadas primeiro.

Podemos então reestruturar um pouco a gramática anterior adicionando novos símbolos não terminais.

$$E \rightarrow E + T$$
$$E \rightarrow E - T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow T/F$$
$$T \rightarrow F$$
$$F \rightarrow num$$
$$F \rightarrow (E)$$

2.4.2 Recursividade à Esquerda

Para eliminar recursividades à esquerda temos de procurar os símbolo terminal que interrompe o ciclo recursivo para o podermos reposicionar a fim de criar uma recursividade à direita, utilizando um novo símbolo não-terminal.

Tendo por exemplo as seguintes produções vamos fazer precisamente isto.

$$P \rightarrow Pa$$
$$P \rightarrow b$$

Pode então ser reescrito como:

$$P \rightarrow bP'$$
$$P' \rightarrow aP' \mid \epsilon$$

Outro exemplo.

$$E \rightarrow E + T$$
$$E \rightarrow T$$

Que se pode reescrever como:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$

2.4.3 Factorização

Outro problema semelhante à recursividade à esquerda é o de ter várias produções cujos parcelas são semelhantes à esquerda até determinado ponto, sendo diferentes à direita depois de começarem com o mesmo símbolo. Confuso? Nada que um exemplo não resolva:

$$S \rightarrow if\ E\ then\ S\ else\ S$$
$$S \rightarrow if\ E\ then\ S$$

O que podemos fazer é substituir a parte semelhante destas produções por um novo símbolo terminal, ficando assim com:

$$S \rightarrow if\ E\ then\ S\ X$$
$$X \rightarrow else\ S\ \mid \epsilon$$

2.5 Analisador LL(1)

As gramáticas livres de contexto que obedecem às características atrás descritas designam-se por Gramáticas LL(1): L: left scan (leitura da esquerda para a direita - num passo) L: leftmost derivation (derivação pela esquerda) (1): 1-symbol lookahead (antecipação de 1 símbolo)

Como já podemos comprovar, para verificar se uma gramática é **LL(1)** temos apenas que preencher a **Tabela de Parsing Predicitvo**.

3 Análise Ascendente

A análise ascendente utiliza **Parsing LR** (scans left to right, rightmost-derivation Parsing bottom-up) que é caracterizado pelo recurso a uma **pilha** e a operações de **shift** e **reduce** para reduzir a string lida ao símbolo inicial da gramática.

Coloca-se então a questão: ***Quando devemos reduzir e quando devemos fazer shift?***

Para responder a esta questão temos de criar um **Diagrama de Estados** e por conseguinte uma **Tabela de Parsing** que, para cada estados nos indicam o conteúdo do look-ahead buffer bem como o estado para o qual devemos transitar mediante o token lido.

3.1 Analisador LR(0)

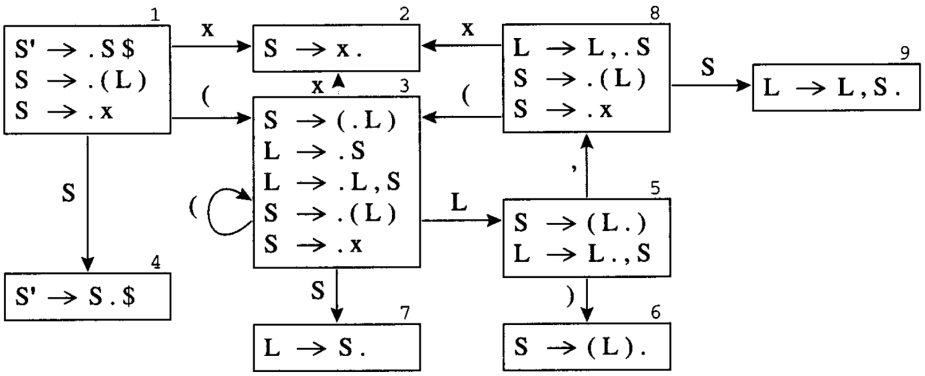
Tomemos então a seguinte gramática para os próximos exemplos:

0. $S' \rightarrow S\$$
1. $S \rightarrow (L)$
2. $S \rightarrow x$
3. $L \rightarrow S$
4. $L \rightarrow L, S$

Começamos por criar o **Diagrama de Estados** para todos as transições possíveis na cabeça de leitura. A cabeça de leitura é denotada por um ponto e para $A \rightarrow \alpha.\beta$ indica que α está no topo da pilha e que β está na cabeça de leitura.

Sempre que na cabeça de leitura se encontra um símbolo não terminal temos de expandir no estado as produções desse mesmo símbolo.

Para a gramática que estamos a estudar obtemos o seguinte diagrama:



Para proceder a construção da **Tabela de Parsing** é preciso estabelecer algumas regras:

- Há uma linha para cada estado do diagrama
- Há uma coluna para cada símbolo (terminais e não-terminais), mas deve ser feita uma separação dos mesmos
- Para acada aresta de transição do estado I para o estado J pelo símbolo X:

- Se **X** é terminal coloca-se shift(J) na posição (I,X)
- Se **X** é não-terminal coloca-se goto(J) na posição (I,X)

- Para o estado I contendo **.$** coloca-se um accept em (I,$)
- Para o estado contendo $A \rightarrow \gamma.$ coloca-se um reduce(n) (n é o número da produção na gramática) em todos os símbolos terminais

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

Uma nota importante, à semelhança do que aconteceu no parsing descendente. Caso haja uma célula da tabela que tenha mais do que uma operação então podemos afirmar que gramática não é LR(0) pois possui conflictos.

3.2 Analisador SLR

Um SLR (Simple LR parser) é idêntico a um LR(0) com uma pequena diferença: aquando da criação da tabela só se efectua **reduce** se para $A \rightarrow \gamma$ o símbolo que está na cabeça de leitura pertencer a Follow(A). E porquê esta alteração? Assim, é muito menos provável haver conflitos que envolvam reduções porque diminuímos drasticamente o número de operações de reduce na tabela, filtrando os que não se podem seguir imediatamente!

Podemos observar isto comparando as Tabelas de Parsing para a gramática que utilizámos na **secção do LR(0)**. Como sempre, dá jeito preencher a tabela auxiliar com os Nullable, First e Follow dos símbolos não-terminais.

	Nullable	First	Follow
S'	Não	"(", "x"	"$"
S	Não	"(", "x"	"$", ",",
L	Não		")", ",",

Agora sim, é clara a diferença que o SLR pode fazer relativamente a eliminar conflictos entre shifts e reduces:

LR(0)

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

VS

SLR

	(	)	x	,	$	S	L
1	s3		s2			g4	
2				r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6					r1		
7				r3	r3		
8	s3		s2			g9	
9		r4		r4			

3.3 Analisador LR(1)

Idêntico ao LR(0) mas recorrendo a um buffer de lookahead, o LR(1) é mais poderoso que o SLR. O diagrama de estados passa a conter o conjunto de símbolos que serão lidos depois do conteúdo na cabeça de leitura ser processado. Assim, $A \rightarrow \alpha.\beta\ ,\ x$ indica que depois de processarmos β será encontrado x.

No entanto, esta não é a única alteração. Uma consequência directa de armazenar os símbolos que esperamos é a de que para a mesma regra podemos esperar diferentes sequências de símbolos, por forma a tratar esses casos temos então de expandir produções sempre que um símbolo não-terminal estiver na cabeça de leitura. Confuso? É normal, se isto fosse trivial teria sido ensinado no ensino primário ;)

Segue-se então um exemplo de como se deve proceder. Para a seguinte gramática

$$S' \rightarrow S\ \$$$
$$S \rightarrow V = E$$
$$S \rightarrow E$$
$$E \rightarrow V$$
$$V \rightarrow x$$
$$V \rightarrow *E$$

o diagrama de estados começará no primeiro estado como

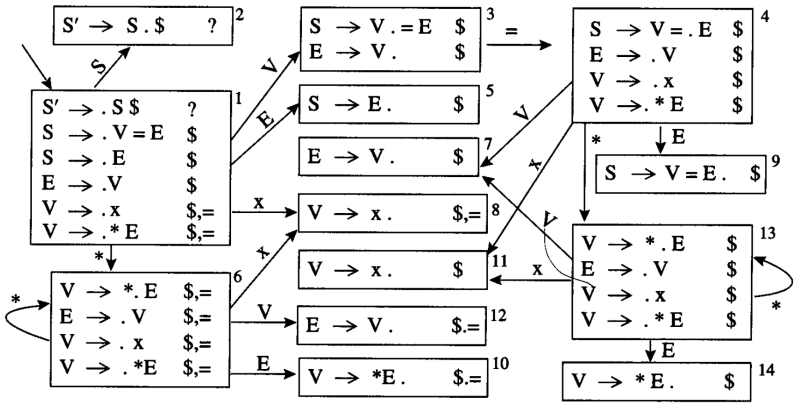
1. $S' \rightarrow .S\ \$\ ,\ ?$
2. $S \rightarrow .V\ =\ E\ ,\ \$$
3. $S \rightarrow .E\ ,\ \$$
4. $V \rightarrow .x\ ,\ =$
5. $V \rightarrow .*\ E\ ,\ =$
6. $E \rightarrow .V\ ,\ \$$
7. $V \rightarrow .x\ ,\ \$$
8. $V \rightarrow .*\ E\ ,\ \$$

O leitor atento notará de imediato que existem agora mais produções que aquelas que resultariam ao aplicar o LR(0). Passamos a explicar, por expansão:

- A regra 1 origina as regras 2 e 3.
- A regra 2 origina as regras 4 e 5.
- A regra 3 origina as regras 6, 7 e 8.

Ora, a única diferença entre as regras (4 e 7) e (5 e 8) deve-se ao lookahead buffer, que acabou por ser diferente devido ao caminho que foi tomado aquando da leitura. Isto não é um problema e podemos então juntar estas regras!

Continuando com o algoritmo chegamos ao seguinte diagrama de estados:
Procedendo do mesmo modo que para o SLR, obtemos a tabela de parsing:



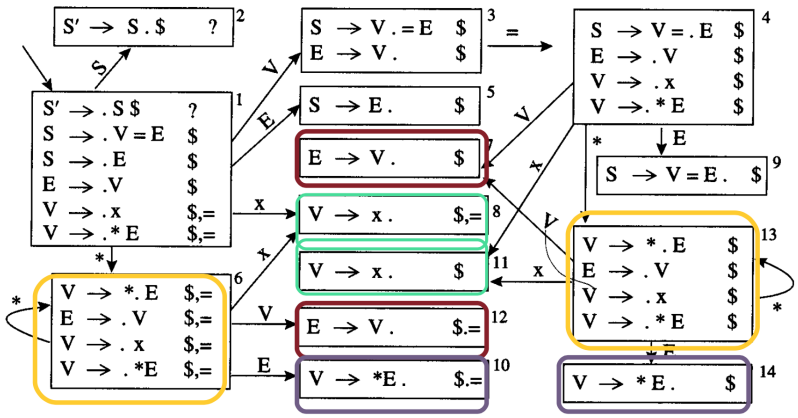
	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

3.4 Analisador LALR(1)

Ora, já sabemos analisar gramáticas segundo 4 algoritmos distintos: LL(1), LR(0), SLR, LR(1). Parece muito? Talvez. Visto que cada um destes algoritmos visa melhorar a capacidade de análise do anterior de modo a serem mais abrangentes. Será que ainda conseguimos melhorar algum aspecto? Sim. O leitor atento terá certamente notado que o diagrama de estados de um LR(1) tende a conter estados que são em tudo idênticos à excepção dos símbolos no lookahead buffer tornando-se **redundante e complexo** desnecessariamente.

Por conseguinte, a tabela de parsing torna-se também desnecessariamente grande e redundante. Assim, o que podemos fazer é fundir esses estados. Mas nem todas as alterações são perfeitas! A fusão de estados pode originar conflitos reduce/reduce que não existiam na tabela de parsing do LR(1).

Nasce assim o **LALR** (Look-Ahead LR parser).



	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

3.5 Ambiguidade e a Análise LR (erros shift/reduce)

Talvez o caso óbvio e comum de ambiguidade nas linguagens de programação seja o de decidir se um **else** deve ser associado ao **if** mais próximo ou ao mais distante.

Esta situação ocorre quando num diagrama de estados temos

$$S \rightarrow if\ E\ then\ S.\ else\ S\ ,\ ?$$

$$S \rightarrow if\ E\ then\ S.\ ,\ else$$

pelo que a primeira produção representará um shift e a segunda um reduce, originando precisamente um conflito de shift/reduce na tabela de parsing.

Como podemos resolver esta situação? Já vimos que talvez seja possível alterar a gramática por meio de factorização, mas podemos também definir um conjunto de regras de precedência e de associatividade para as regras e os tokens.

3.5.1 Precedência e Associatividade

Alguns **conflitos shift/reduce** podem ser resolvidos estabelecendo precedência e associatividade dos tokens.

Precedência:

- Se o símbolo a ler tiver **menos prioridade** que o já lido efectua-se **reduce**. $E * E. + E$
- Se o símbolo a ler tiver **maior prioridade** que o já lido efectua-se **shift**. $E + E. * E$

E em caso de terem a mesma prioridade? Recorremos à associatividade!

Associatividade:

- à **esquerda** é equivalente a efectuar **reduce**. $E + E. + E$
- à **direita** é equivalente a efectuar **shift**. $E = E. = E$

3.6 Erros reduce/reduce

Um erro reduce/reduce ocorre quando podemos optar entre duas produções para o mesmo input, ou seja, podemos ter árvores que representam o mesmo texto mas que são diferentes. Infelizmente, não existe outra solução que não a de alterar a gramática para eliminar os caminhos múltiplos. Segue-se um exemplo desta mesma situação:

- $S \rightarrow id = A$
- $S \rightarrow id = B$
- $B \rightarrow B + B \mid id$
- $A \rightarrow A - A \mid id$

Ora, o problema nesta gramática reside no facto tanto **A** como **B** poderem gerar os símbolos terminais **id** para o input $x = y$. Mas podemos reescrever a gramática!

- $S \rightarrow id = B$
- $B \rightarrow B + B \mid \mathbf{A}$
- $A \rightarrow A - A \mid id$

4 Resumo dos Analisadores

Segue-se um resumo do que é necessário fazer com cada analisador.

- **LL(1):**
  - Preencher a **Tabela Auxiliar**.
  - Preencher a **Tabela de Parsing Predictivo**.
- **LR(0):**
  - Criar o **Diagrama de Estados**.
  - Criar a **Tabela de Parsing** com base nas transições entre estados.
- **SLR:**
  - Preencher a **Tabela Auxiliar**.
  - Criar o **Diagrama de Estados**
  - Criar a **Tabela de Parsing** com base nas transições entre estados, com auxílio dos Follow na Tabela Auxiliar.
- **LR(1):**
  - Preencher a **Tabela Auxiliar**.
  - Criar o **Diagrama de Estados**, preenchendo os símbolos no **lookahead buffer** e expandindo produções sempre que na cabeça de leitura esteja um símbolo não-terminal.
  - Criar a **Tabela de Parsing** com base nas transições entre estados, com auxílio dos símbolos de lookahead.
- **LALR(1):**
  - Preencher a **Tabela Auxiliar**.
  - Criar o **Diagrama de Estados**, preenchendo os símbolos no **lookahead buffer** e expandindo produções sempre que na cabeça de leitura esteja um símbolo não-terminal.
  - **Fundir estados** que sejam idênticos à excepção dos lookahead buffers.
  - Criar a **Tabela de Parsing** com base nas transições entre estados, com auxílio dos símbolos de lookahead.

Na seguinte tabela podemos perceber quais as diferenças que os analisadores ascendentes têm, tornando-se mais abrangentes e simples:

	Diagrama de Estados	**Lookahead buffer**	**Tabela de Parsing**	**Número de Reduces**	**Fusão de Estados**
LR(0)	X		X		
SLR	X		X	menos que LR(0)	
LR(1)	X	X	X	menos que SLR	
LALR(1)	X	X	X	menor ou igual a LR(1)	X

5 Código de 3 endereços (C3E)

5.1 Tempo de Vida

Para as variáveis existentes no código de 3 endereços existe o conceito de "***tempo de vida***"que expressa, para cada instrução do programa quais as variáveis que se encontram em memória, quer por estarem a ser usadas ou por virem a ser necessárias em operações posteriores.

Podemos dividir as variáveis de C3E em dois tipos: temporárias (t_1, t_2, etc.) e do programa (a, b, x, y, etc.).

5.1.1 Variáveis temporárias

As variáveis temporárias são, como o nome indica, usadas para armazenar cálculos intermédios e estão vivas desde o momento em que são criadas até ao momento em que são usadas pela última vez, e têm de estar **todas mortas na última instrução do programa**. Por este motivo é mais simples analisar o tempo de vida das variáveis temporárias **do início para o fim** do programa.

5.1.2 Variáveis do programa

As variáveis do programa têm de estar **todas vivas na última instrução do programa** e estão mortas desde a última vez que são o alvo de uma atribuição para trás. Por este motivo é mais simples analisar o tempo de vida das variáveis do programa **do fim para o início** do programa.

5.1.3 Tabela de tempo de vida

Criamos um tabela com uma linha para cada instrução do programa, pela ordem em que aparecem, e com uma coluna para cada variável (temporárias e do programa, devendo estas estar agrupadas de acordo com a sua classificação). Para cada instrução devemos indicar quais as variáveis que estão mortas e quais as que estão vivas, podendo opcionalmente ser indicado em cada célula das variáveis que se encontram vivas qual a instrução em que são/foram usadas por último.

Um pequeno à parte: caso haja variáveis que não sigam as regras atrás descritas então podemos afirmar que são inúteis **ou** que têm lixo. Por exemplo, uma variável de programa que chegue ao fim morta indica-nos que nunca foi aramazenada informação nela.

5.2 Diagrama de fluxo

Para criar o diagrama de fluxo é necessário definir os conceitos de **instrução leader** e **bloco de instruções**.

5.2.1 Leader

Uma instrução é leader se **pelo menos uma** das seguintes condições for verdadeira:

- é a primeira instrução do programa
- é a instrução destino de um salto condicional ou incondicional (*goto(n)* indica que a instrução em n é um leader)
- é a instrução que se segue a um salto condional ou incondicional (a instrução a seguir a uma instrução que tenha um *goto* é um leader)

5.2.2 Block

Estando definidos as instruções leader, um bloco é o conjunto de todas instruções entre dois leaders, incluíndo o leader que dá início ao bloco e excluíndo o próximo leader.

5.2.3 Ciclos

Os ciclos não são nada mais nem nada menos que os conjuntos de blocos que se podem repetir devido à presença dos saltos condicionais ou incondicionais do programa (goto, if-else, while, etc.). Nada impede um bloco de se chamar a ele próprio num ciclo.

5.2.4 Entry & Exit

Como se não bastasse, é necessário adicionar um ponto de entrada e um de saída ao programa. Assim, criamos um bloco denominado **Entry** que salta para a primeira instrução/bloco do programa e um bloco **Exit** que resulta do salto do bloco final do programa.