Compiladores - Prova 1

André L. Mendes Fakhoury
Gustavo V. V. Silva Soares
Eduardo Dias Pennone
Matheus S. Populim
Thiago Preischadt

2021

1 Considere a gramática

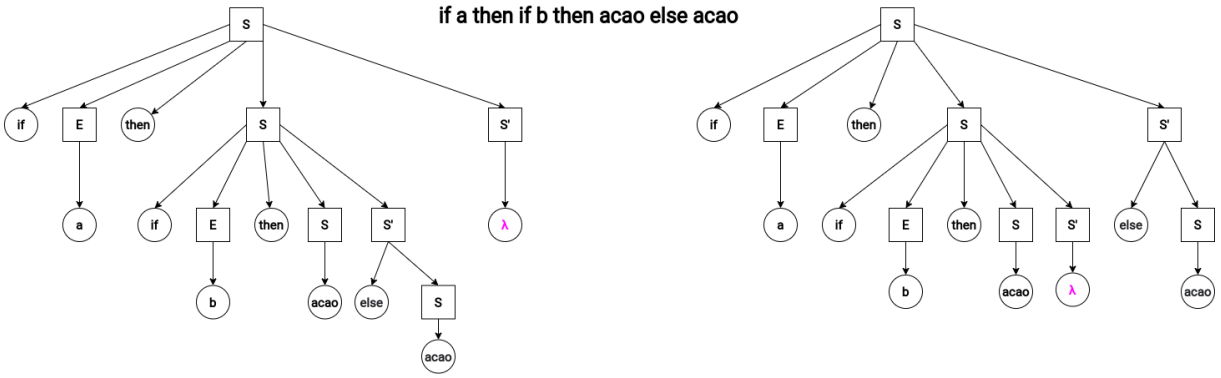
```
<S> ::= if <E> then <S> | if <E> then <S> else <S> | acao
<E> ::= a | b
```

1.1 Transforme em uma gramática LL(1)

```
<S> ::= if <E> then <S> <S'> | acao
<S'> ::= else <S> | λ
<E> ::= a|b
```

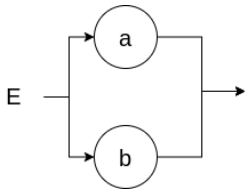
1.2 Mostre duas árvores de derivação para a sentença "if a then if b then ação else ação"

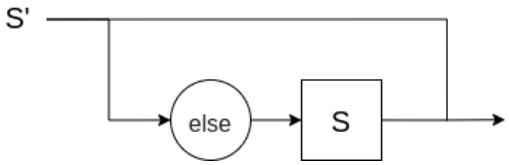
eu to com o diagrama no draw.io, ainda vou exportar de novo pra ficar com os níveis da árvore na mesma altura

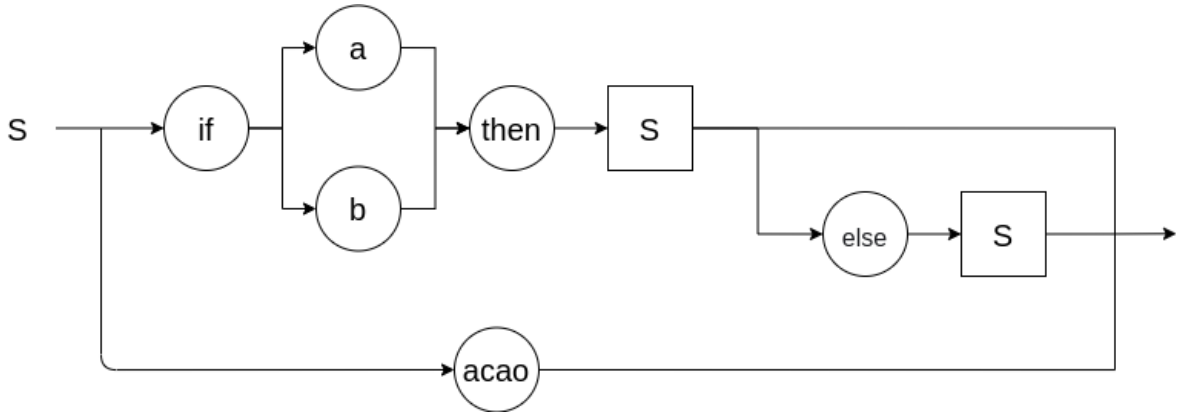


1.3 Considerando a ASD preditiva recursiva: crie os grafos sintáticos e implemente os procedimentos recursivos, incluindo a chamada referente ao símbolo inicial.

Grafos sintáticos de `E,S',S`. O grafo de `S` já está no formato reduzido.








```
procedimento S'
begin
    se (simbolo = 'else')
        obter simbolo;
        S;
end

procedimento S
begin
    enquanto (simbolo <> 'acao') faca
        se (simbolo = 'if')
            obter_simbolo;
            se (simbolo = 'a') ou (simbolo = 'b')
                obter_simbolo;
                se (simbolo = 'then')
                    obter_simbolo;
                senao erro;
            senao erro;
        senao erro;
        obter_simbolo;
```

```
        S';
end

procedimento ASD
begin
    obter simbolo;
    S;
    se terminou_cadeia entao
        sucesso
    senao
        erro
end
```

1.4 Considerando a ASD preditiva não recursiva: crie a tabela sintática. Mostre também uma inconsistência da tabela. Considerando as linguagens de programação mais atuais, mostre como essa inconsistência pode ser resolvida.

	$	**if**	**then**	**acao**	**else**	**a**	**b**
S		S ::= if E then S S'		S ::= acao			
S'	S' ::= λ				S' ::= else S S' ::= λ		
E						E ::= a	E ::= b

A inconsistência ocorre pelo fato das seguintes regras serem ambíguas.

```
S' ::= else S
S' ::= λ
```

Afinal, sabendo que

```
else S = λelse S
```

Vemos a ambiguidade em:

```
S' ::= λelse S
S' ::= λ
```

Para corrigir isso, podemos aplicar uma regra para diferentes tipos de `S`: um que seja `S1` e `S2`. `S1` é um bloco condicional com `else`, e `S2` é um bloco condicional sem `else`.

```
<S> ::= <S1> | <S2>
<S1> ::= if <E> then <S1> else <S1> | acao
<S2> ::= if <E> then <S>
<S2> ::= if <E> then <S1> else <S2>
<E> ::= a|b
```

A lógica por trás dessa abordagem é garantir que o termo `S2` nunca seja chamado entre um `then` e um `else`.

2 Considerando a gramática abaixo, encontre primeiro e seguidor dos símbolos não terminais:

```
<E>::=<T><E'>
<E'>::=+<T><E'>|λ
<T>::=<F><T'>
<T'>::=*<F><T'>|λ
<F>::=(<E>)|id|λ
```

Neste exercício, o conjunto primeiro de um símbolo T será denotado por $P(T)$, e o conjunto seguidor será denotado por $S(T)$.
Começando pelo conjunto primeiro, é fácil de ver que $P(E') = \{+, \lambda\}$, $P(T') = \{*, \lambda\}$ e que $P(F) = \{(, id, \lambda\}$.
Calculando agora $P(T)$. Como $\lambda \in P(F)$ e $\lambda \in P(T')$:

$$P(T) = P(F) \cup P(T') \cup \lambda = \{(, id, *, \lambda\}$$

e, por fim, $P(E)$. Temos que $\lambda \in P(T)$ e $\lambda \in P(E')$, logo:

$$P(E) = P(T) \cup P(E') \cup \lambda = \{+, *, (, id, \lambda\}$$

Portanto, temos os conjuntos primeiros:

$$\begin{aligned} P(E) &= \{+, *, (, id, \lambda\} \\ P(E') &= \{+, \lambda\} \\ P(T) &= \{(, id, *, \lambda\} \\ P(T') &= \{*, \lambda\} \\ P(F) &= \{(, id, \lambda\} \end{aligned}$$

Calculando agora o conjunto seguidor, podemos analisar cada regra de transição.
Começando pela símbolo inicial, temos que $S(E)$ terá λ (por ser inicial), e também) por conta da transição $F \to (E)$. Assim, $S(E) = \{), \lambda\}$.
Pela primeira regra, temos que $S(E') = S(E)$, logo $S(E') = \{), \lambda\}$.
Calculando agora $S(T)$, ele será a união de $P(E')$, por conta de $E \to TE'$, $S(E)$, por conta de $E \to TE'$ e $S(E')$, por conta de $E' \to +TE'$, e $\lambda \in P(E')$. Assim, $S(T) = \{+,), \lambda\}$.
O $S(T')$ será igual ao seguidor de T (pois $T \to FT'$), logo $S(T') = \{+,), \lambda\}$.
E, por fim, temos que analisar todas as regras de transição que possuem F para calcular $S(F)$. O símbolo λ está presente em todos os conjuntos primeiro (em específico, no de T' também), e, a partir das transições $T \to FT', T' \to *FT'$, temos que $S(F) = P(T') \cup S(T) \cup S(T') = \{*, +,), \lambda\}$.
Portanto, temos:

$$\begin{aligned} S(E) &= \{), \lambda\} \\ S(E') &= \{), \lambda\} \\ S(T) &= \{+,), \lambda\} \\ S(T') &= \{+,), \lambda\} \\ S(F) &= \{+, *,), \lambda\} \end{aligned}$$

3 Considere as seguintes questões:

3.1 Explique a razão pela qual as linguagens de programação são implementadas como linguagens livres de contexto, embora as informações de contexto sejam importantes, por exemplo, para identificar se uma variável já foi declarada.

A maioria das implementações da parte sintática de linguagens de programação são implementadas com gramáticas livres de contexto, portanto a análise sintática é feita através de gramáticas livres de contexto. Mas há um problema: o contexto é importante dentro da linguagem de programação. Quanto mais complexa a linguagem mais difícil tratá-la, isto é, mais difícil desenvolver um método de parser para reconhecer aquela linguagem. No entanto é interessante que a capacidade da gramática não seja muito limitada. A linguagem livre de contexto é suficientemente simples e capaz de gerar toda a complexidade necessária para linguagens de programação, sendo mais complexa do que a linguagem regular, e mais fácil de tratar do que uma linguagem sensível ao contexto.

Para realizar a análise semântica seria necessária uma gramática sensível ao contexto, pois tudo o que depende do contexto está relacionado à semântica. No entanto a parte semântica não é tratada através de gramáticas, sendo tratada de maneira informal através de tabelas de símbolos e com anotações semânticas que são feitas na fase de análise sintática. A partir da gramática livre de contexto e feito um parser para que a análise sintática seja realizada, e a ausência de informação de contexto da análise sintática é tratada através da adição de uma nova fase: a fase de análise semântica. Em resumo: Na análise sintática é definida a linguagem de programação através de uma gramática livre de contexto e o tratamento semântico é feito posteriormente, na fase de análise semântica, de maneira informal, com tabela de símbolos e com anotações nas estruturas obtidas na fase de análise sintática. As etapas de análise serão explicadas com mais detalhes na próxima questão.

3.2 Desenhe a estrutura tradicional de um compilador, incluindo todas as suas etapas e estruturas de dados auxiliares. Explique a função de cada etapa e estrutura referente à parte de análise.

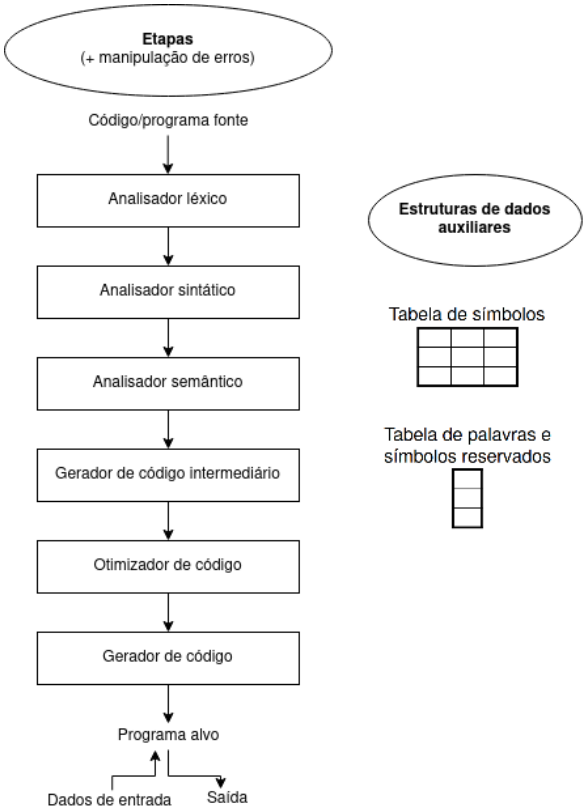


Figura 1: Estrutura de um compilador. Fonte: baseado nos materiais de aula do Prof. Amancio

Análise Léxica

A análise léxica é a primeira fase do processo de compilação e tem como objetivo identificar unidades léxicas ou lexemas que compõem o programa. O analisador léxico lê todos os caracteres do programa fonte, verifica se eles pertencem ao alfabeto da linguagem e, caso um caractere não pertença, é gerado um erro léxico.

Os comentários e espaços em branco são ignorados e removidos e nesse processo um conjunto de *tokens* - que são formados por palavras reservadas, identificadores, delimitadores etc - é formado. Nessa etapa é criada a tabela de símbolos.

Em resumo: a análise léxica quebra o texto do programa fonte em lexemas, verifica a categoria ao qual eles pertencem e produz uma sequência de símbolos léxicos chamados de *tokens*.

O analisador léxico deve permitir identificar na linguagem repetições de subconjuntos permitindo que seja possível identificar e classificar esses subconjuntos, por exemplo o subconjunto de palavras reservadas.

Nessa fase o processamento de uma linguagem pode ser feito por gramáticas regulares podendo ser formalmente descrito por expressões regulares. As rotinas que processam essa linguagem modelam algoritmos construídos a partir de autômatos finitos.

Análise Sintática

A etapa de análise sintática tem como objetivo validar a gramática do programa ao reconhecer se a estrutura gramatical do código fonte está de acordo com as regras sintáticas da linguagem.

Nessa etapa é produzida uma estrutura de dados em formato de árvore conhecida como árvore sintática. Elá é formada após feita uma varredura na sequência de *tokens* recebidas do analisador léxico. A árvore sintática representa a hierarquia do programa fonte. Caso uma construção seja reconhecida com inválida um erro sintético deve ser gerado.

Mesmo com uma boa técnica de detecção de erros o analisador sintático deve ser capaz de recuperá-los e continuar o processo de compilação identificando o maior número possível.

A sintaxe da maioria das linguagens de programação é especificada usando gramáticas livres de contexto que permitem realizar substituições impostas por regras produção e assim validar a estrutura do programa.

Análise Semântica

Essa etapa tem como objetivo verificar se a semântica do programa fonte tem consistência. Para isso é utilizada a árvore sintática e as informações contidas na tabela de símbolos.

A verificação de tipo em uma operação de soma onde cada operando é verificado com cada operador, por exemplo.

Pode ser necessário que nessa fase alguns tipos de dados sejam convertidos para outros tipos, essa operação é conhecida como coerção.

Alguns tipos de erros semânticos:
- Tipos de operandos incompatíveis com operadores;
- Identificadores, variáveis e procedimento, não declarados;
- Chamada de funções ou métodos com número incorreto de operadores;
- Comandos fora de contextos, um comando continue fora de um laço;
- Operações de conversão de tipos.

A tabela de símbolos é muito importante nessa etapa pois é através dela que é possível recuperar informações sobre os identificadores que são utilizadas para avaliar as regras semânticas.

Tabela de Símbolos

É utilizada principalmente dentro da fase de análise semântica. Essa tabela é responsável por armazenar os atributos semânticos relacionados às variáveis ou outros elementos importantes dentro do programa. Por exemplo podemos ter uma variável, com a informação de seu valor, com a informação de qual escopo ela foi declarada e assim por diante.

Os seguintes atributos costumam ser gravados na tabela de símbolos:
- Classe dos identificadores: variável, função, etc.
- Tipo de dado: Inteiro, String, etc.
- Tipos de retornos: no caso de métodos • Variáveis: tipo, endereço no texto, posição e tamanho.
- Parâmetros formais: tipo do mecanismo de passagem, por valor ou referência.

• Procedimentos/sub-rotinas: número de parâmetros.

A tabela de símbolos é utilizado durante todo o processo de compilação a fim de inserir e extrair informações de forma rápida e eficiente.

Podemos armazenar na tabela de símbolos também informações sobre a linha e coluna que o *token* foi examinado para em caso de erro o compilador passa informar a posição da falha.

Tabela de Palavras e Símbolos Reservados

A tabela de palavras e símbolos reservados é importante para que possamos diferenciar aquilo que faz parte da linguagem de programação daquilo que não faz.

Nessa tabela serão armazenadas as palavras chaves que serão necessárias para diferenciarmos o que seria um identificador de uma variável, um identificador de um nome de programa, um identificador de um nome de procedimento, de uma palavra-chave. Essa identificação é feita em forma de tabela porque, em geral, muitas das palavras chaves possuem formatos semelhantes ao formato de identificadores.

Árvore Sintática

A árvore sintática representa a sequência hierárquica da linguagem de programação de maneira estruturada em um grafo (no caso, como o nome sugere, uma árvore). Essa estrutura permite representar cada elemento do programa, e os demais passos do compilador consistem em visitar os nós dessa estrutura em uma determinada ordem.

Esta representação gráfica é resultante das derivações de sentenças - cada nó representa uma unidade sintática, formada por um simbolo terminal ou simbolo não terminal. Na fase de análise semântica, informações são inseridas nos nós da árvore sintática a fim de dar informações semânticas das variáveis relacionadas ou do conjunto representado em cada um dos nós. Posteriormente, em fases futuras, essa estrutura não linear será transformada em uma estrutura linear (que seriam as fases de síntese).

Referências complementares utilizadas:

O que é uma linguagem livre de contexto? *Stack overflow*. Disponível em: <https://pt.stackoverflow.com/questions/180927/o-que-%C3%A9-uma-linguagem-livre-de-contexto> Acesso em 27 maio 2020.

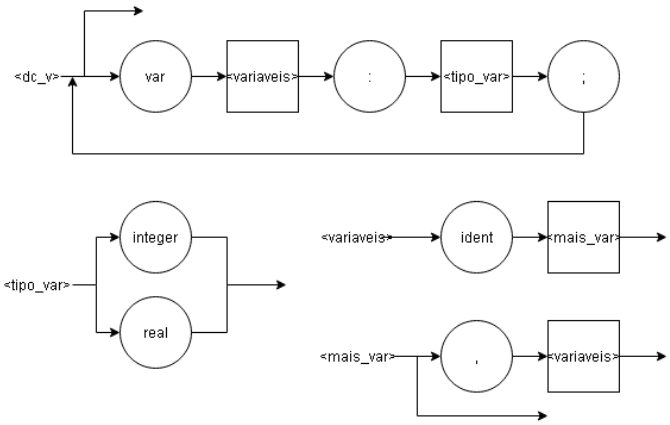
Estrutura de um Compilador. MARANGON, Johni. Disponível em: <https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/structure-of-a-compiler.html> Acesso em 27 maio 2020.

4 Considere as seguintes questões com relação à gramática da LALG (disponível no TIDIA)

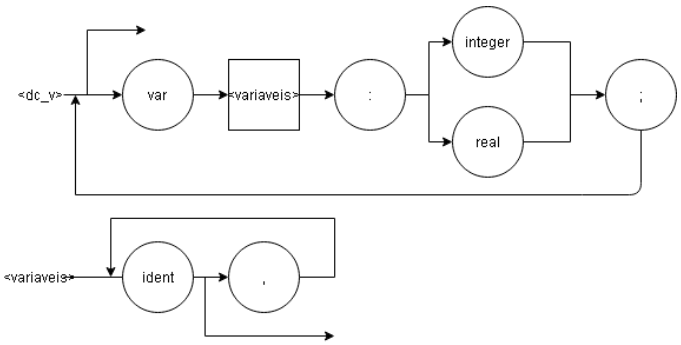
4.1 Identifique as 4 regras responsáveis pela declaração de variáveis.

```
<dc_v> ::= var <variaveis> : <tipo_var>; <dc_v> | λ
<tipo_var> ::= real | integer
<variaveis> ::= ident <mais_var>
<mais_var> ::= , <variaveis> | λ
```

4.2 Construa os grafo(s) sintático(s) em número reduzido (tente substituir grafos dentro de grafos)



Podemos substituir o grafo <tipo_var> em <dc_v>, e o grafo <mais_var> em <variaveis>. Com isso reduzimos de quatro grafos para dois.



4.3 Construa o(s) procedimento(s) recursivo(s) para o analisador sintático destas 4 regras. Crie também o procedimento principal que inicializa a análise sintática.

```
procedimento <dc_v>
begin
    se simbolo = var entao
        obter_simbolo
        variaveis
        se simbolo = : entao
            se simbolo = real ou simbolo = integer entao
```

```
                obter_simbolo
                se simbolo = ; entao
                    obter_simbolo
                    dc_v
                senao
                    erro
            senao
                erro
        senao
            erro
end

procedimento <variaveis>
begin
    se simbolo = ident entao
        obter_simbolo
        se simbolo = , entao
            variaveis
    senao
        erro
end

procedimento ASD
begin
    obter_simbolo
    dc_v
    se terminou_cadeia entao
        sucesso
    senao
        erro
end
```

5 Considere a gramática abaixo com as seguintes 3 regras. Considerando uma análise preditiva não recursiva, monte a tabela sintática completa, com tratamento de erros.

```
<S> ::= 0<X>1 | cc<C>
<X> ::= x<X> | λ
<C> ::= c
```

Primeiramente vamos definir os conjuntos Primeiro e Seguidor de cada símbolo não terminal:

$P(S) = \{0, c\}$
$P(X) = \{x, \lambda\}$
$P(C) = \{c\}$

$S(S) = \{\lambda\}$
$S(X) = \{1\}$
$S(C) = \{\lambda\}$

Agora criaremos a tabela sintática utilizando as regras da gramática:

	0	1	c	x	λ
S	$S \to 0X1$		$S \to ccC$		
X				$X \to xX$	$X \to \lambda$
C			$C \to c$		

Note que o símbolo X pode se apagar (utilizando a segunda regra da gramática). Dessa forma, precisamos utilizar os elementos do conjunto seguidor $S(X)$ para indicar na tabela como obter o 1:

	0	1	c	x	λ
S	$S \to 0X1$		$S \to ccC$		
X		$X \to \lambda$		$X \to xX$	$X \to \lambda$
C			$C \to c$		

Para os demais cenários, será necessário tratamento de erro.

Para reagir a uma entrada que se encontra no conjunto seguidor de um símbolo não-terminal, a tratativa de erro será de desempilhá-lo.

	0	1	c	x	λ
S	$S \to 0X1$		$S \to ccC$		Desempilhar
X		$X \to \lambda$		$X \to xX$	$X \to \lambda$
C			$C \to c$		Desempilhar

Para os outros, precisaremos usar a função de erro de forma a encontrar um ponto de sincronização.

O primeiro grau de sincronização será utilizando o conjunto primeiro de um símbolo não-terminal: São consumidos caracteres de entrada até que encontrado algum símbolo terminal (excluindo-se o λ) que consta no conjunto.

Caso não sejam encontrados, o segundo grau de sincronização é a procura dos símbolos terminais (excluindo-se o λ) do conjunto seguidor deste símbolo não-terminal.

Dessa forma, a tabela ficará da forma:

	0	1	c	x	λ
S	$S \to 0X1$	Erro($\{0, c\}$)	$S \to ccC$	Erro($\{0, c\}$)	Desempilhar
X	Erro($\{x\} + \{1\}$)	$X \to \lambda$	Erro($\{x\} + \{1\}$)	$X \to xX$	$X \to \lambda$
C	Erro($\{c\}$)	Erro($\{c\}$)	$C \to c$	Erro($\{c\}$)	Desempilhar