



Universidade Estadual de Santa Cruz
Departamento de Ciências Exatas e Tecnológicas
Curso de Ciência da Computação
Disciplina: Compiladores – 2017.2

Especificação do 1º trabalho

Analizador Léxico para Portugol

Professor

Paulo Costa

Ilhéus, BA
25 de setembro de 2017

Sumário

1. Introdução	1
2. Instruções gerais	1
2.1. Realização	1
2.2. Entrega	1
2.3. Aceitação	3
2.4. Avaliação	3
2.5. Segunda chamada	3
3. Requisitos do trabalho	4
3.1. Modularização	4
3.2. Autômato	4
3.3. Tabela de símbolos	5
3.4. Geração das saídas	5
4. Especificação léxica da linguagem Portugal	6
4.1. Tipos de dados	6
4.2. Constantes e variáveis	6
4.3. Identificadores	6
4.4. Palavras reservadas	6
4.5. Operadores	7
4.6. Gramática	7
4.7. Expressões regulares	7
4.8. Delimitadores	8
4.9. Comentários	9
5. Tratamento de erros	10
6. Impressão dos resultados para o relatório	11
6.1. Arquivo de entrada e erros léxicos	11
6.2. Lista de tokens reconhecidos	12
6.3. Tabela de símbolos	13
7. Realização	14
7.1. Plano de ação	14
7.2. Cronograma	16
7.3. Desenvolvimento	16
8. Avaliação	17
8.1. Defesa	17
8.2. Discrepâncias	17
8.3. Compilação	17
8.4. Testes	18
8.5. Modelo	18
8.6. Critérios de avaliação	19
8.7. Pontuação	19
8.8. Casos omissos	19
8.9. Desempenho na defesa	19
9. Formulário de pré-avaliação	23

9.1.	Resumo	23
9.2.	Campos do formulário	23
9.3.	Referências cruzadas.....	24
9.4.	Seção A – Nome do arquivo de entrada	25
9.5.	Seção B – Tratamento de maiúsculas, minúsculas e acentos	25
9.6.	Seção C – Detalhes de implementação.....	25
9.7.	Seção D – Estruturas de dados de tamanho arbitrário.....	26
9.8.	Seção E – Tokens.....	26
9.9.	Seção F – Tabela de transições	27
9.10.	Seção G – Tabela de símbolos.....	27

Lista de Tabelas

Tabela 1 – Exemplo de plano de ação.	15
Tabela 2 – Exemplo de cronograma de trabalho.	16
Tabela 3 – Critérios de avaliação e descontos na nota.	20

Lista de Figuras

Figura 1 – Exemplo de programa Portugol incorreto.....	11
Figura 2 – Exemplo de avaliação: Erros cometidos.	21
Figura 3 – Exemplo de avaliação: Descontos nas notas.	22
Figura 4 – Exemplo de avaliação: Notas.....	22
Figura 5 – Estrutura do formulário de avaliação.....	24
Figura 6 – Exemplo de referência cruzada no formulário de avaliação.....	24
Figura 7 – Exemplo de referência cruzada no código impresso.....	24

1. Introdução

O objetivo do trabalho é projetar, implementar e testar um analisador léxico para a linguagem de programação fictícia Portugol. Dada a complexidade do trabalho, muitas dúvidas podem surgir durante sua realização. Este documento visa responder todas elas. Leia-o pelo menos duas vezes. Uma forma didática de fazer a leitura em três etapas: primeiro, leia apenas a seção 4; depois, as seções 3 a 5; por fim, todo o documento.

2. Instruções gerais

2.1. Realização

O trabalho deve ser realizado em duplas. Se o tamanho da turma for ímpar, o aluno que não se encaixar em nenhum grupo deve procurar o professor da disciplina pelo menos 15 dias antes da data da entrega. Em nenhuma outra circunstância será aceito trabalho que não em dupla.

O código deve ser implementado em C, com qualquer compilador ou ambiente de desenvolvimento, desde que seja compilável sem modificações com GCC em Linux, pois esse será o ambiente usado para a avaliação de todos os trabalhos (veja seção 8.3, pág. 17). Além disso, os programas serão executados em um processo de testes automatizado que requer a observância de diversas convenções (veja seção 8.4, pág. 18).

2.2. Entrega

O trabalho deve ser entregue em dois formatos:

- Eletrônico: por email, para psscosta@uesc.br até as 23h59m da véspera da primeira aula reservada para as defesas.
- Impresso: um relatório, descrito a seguir, entregue nos 10 primeiros minutos da primeira aula reservada para as defesas. **Importante:** o preparo do relatório é muito trabalhoso, não deixe para o último dia.

Entrega e defesa antes do prazo: negocie com o professor com pelo menos uma semana de antecedência.

2.2.1. Relatório – conteúdo

O relatório deve conter os seguintes itens e nada mais, nesta ordem:

- Capa contendo ao menos os nomes dos autores e a data da entrega.
- Sumário indicando títulos das seções e respectivos números de página.
- Código fonte integral do programa, com linhas numeradas.
- Representações do autômato utilizado:
 - Diagrama de estados e transições, claro e legível, criado com qualquer programa de computador.
 - Descrição das ações associadas aos vários estados, especialmente os de aceitação.
 - Tabela de transições de estados propriamente dita, usada pelo programa.
- Resultados do processamento dos três arquivos de teste. Para cada teste, três arquivos separados:
 - Conteúdo do arquivo de teste e os erros léxicos encontrados (seção 6.1, pág. 11).
 - Lista de *tokens* reconhecidos no arquivo de entrada (seção 6.2, pág. 12).
 - Conteúdo da tabela de símbolos após processar cada entrada (seção 6.3, pág. 13).
- *Formulário de Pré-Avaliação* preenchido a mão e de lápis (seção 9, pág. 23).

2.2.2. Relatório impresso – preparação recomendada

Como método recomendado, use o documento Word **Analisador Léxico – Relatório.doc** fornecido pelo professor; esse modelo pré-formatado simplificará bastante a produção do relatório. Adicione seus conteúdos **sem mudar** nenhuma formatação do documento (fonte, espaços entre linhas, margens, etc).

Copie seu código fonte, resultados dos testes etc. e cole no relatório pré-formatado, preservando a formatação do documento (*Editar | Colar especial | Texto não formatado*). Não use a opção *Colar* simples, pois isso manterá a formatação do original, arruinando a formatação pré-definida.

No cabeçalho de qualquer página, digite os nomes dos autores e clique com o botão direito do mouse sobre a data para atualizá-la (*Atualizar campo*).

Ao final, clique com o botão esquerdo do lado esquerdo do sumário e pressione F9 para atualizá-lo.

Não preencha o *Formulário de Pré-Avaliação* no Word; imprima o documento e só então preencha-o a mão, seguindo as instruções na seção 9 (pág. 23).

2.2.3. Relatório impresso – preparação alternativa

Use Microsoft Word ou qualquer processador de texto. Inclua tudo que está descrito na seção 2.2.1 (pág. 1), exatamente na ordem listada, e nada mais. A capa e o sumário podem ter formatação livre. O código fonte e os resultados dos testes devem sem formatados com

- fonte de largura fixa, como Lucida Console 10pt,
- espaço simples entre as linhas, e
- margens de 2cm nos quatro lados.

Em hipótese alguma use fonte proporcional como Times New Roman ou Arial. O código fonte deve ser impresso com as linhas numeradas. Certifique-se que a impressão não contém linhas truncadas ou quebras de linha excessivas. Isso pode ser evitado usando

- fonte menor que 10pt (mas não a ponto de tornar o texto ilegível) e
- indentação moderada (p.ex. 2 espaços em branco por nível de indentação).

Inclua em cada página um cabeçalho contendo uma descrição do seu conteúdo. Exemplos: “*Código fonte*” e “*Testes, prog3.ptg, Tokens reconhecidos*”.

Inclua o *Formulário de Pré-Avaliação*, em branco, a ser preenchido a mão depois de impresso. Ele é parte do documento **Analisador Léxico – Relatório.doc** (ver seção 2.2.2, pág. 1).

Numere todas as páginas em uma única sequência, posicionando os números no rodapé da página, à direita. Não numere cada parte do documento a partir da página 1.

Imprima o relatório, depois encaderne ou grampeie tudo em um único volume.

2.2.4. Submissão eletrônica – conteúdo

O que entregar em formato eletrônico:

- Código fonte completo do programa.
- Arquivos de teste e os resultados de seu processamento.
- Arquivos usados na elaboração do relatório impresso.

2.2.5. Submissão eletrônica – preparação

Crie uma pasta com os nomes dos autores, p.ex. **JoseCarlos-AnaMaria**. As iniciais de cada nome devem ser maiúsculas; as demais letras, minúsculas. Não use números, letras acentuadas, espaços em branco, sublinhados, caracteres especiais, etc. Crie nessa pasta principal três sub-pastas: **Código**, **Testes** e **Relatório**.

Coloque na pasta **Código** o código fonte completo do programa. Inclua nessa pasta apenas arquivos **.c** e **.h**. Não inclua o *makefile*, os arquivos compilados (**.o**) ou o executável (**Portugol**). Não use nenhuma biblioteca estática ou dinâmica que não faça parte da biblioteca padrão de GCC em Linux.

Certifique-se que

- essa pasta contém tudo que é necessário à compilação de seu código em outra máquina com Linux;
- quando seu código for compilado pelo professor (seção 8.3, pág. 17), todos os arquivos lidos e gravados pelo compilador residem na mesma pasta;
- quando seu código for executado pelo professor (seção 8.4, pág. 18), todos os arquivos lidos e gravados pelo executável residem na mesma pasta.

Mova para a pasta **Testes** os três arquivos de testes fornecidos e as saídas do programa. São esperados 12 arquivos nesta pasta:

teste-01. ptg	teste-02. ptg	teste-03. ptg
teste-01. ptg. err	teste-02. ptg. err	teste-03. ptg. err
teste-01. ptg. tok	teste-02. ptg. tok	teste-03. ptg. tok
teste-01. ptg. tbl	teste-02. ptg. tbl	teste-03. ptg. tbl

Coloque na pasta **Relatório** o relatório da versão impressa (ver seção 2.2.1, pág. 1) apenas em formato *pdf*. Certifique-se que no *pdf* não há porções truncadas do documento original, especialmente nos cabeçalhos, rodapés e margens laterais. Não inclua o arquivo salvo pelo editor de texto usado (Word ou equivalente). Coloque também nessa pasta as representações do autômato utilizado. Não inclua o arquivo salvo pelo programa usado para criar o diagrama (p.ex. **automato.vsd** ou **automato.svg**), mas é necessário incluir a imagem usada no relatório, no formato *jpeg* ou *png* (p.ex. **automato.jpg** ou **automato.png**), com resolução mínima de 300dpi.

Não serão corrigidos diagramas desenhados a mão ou, por qualquer motivo, ilegíveis, integral ou parcialmente. Certifique-se que o diagrama do autômato incluído no relatório impresso é claro e totalmente legível, incluindo todos os símbolos de transição entre estados.

Compacte a pasta principal criando um arquivo **zip**, p.ex. **JoseCarlos-AnaMaria.zip** Use qualquer ferramenta de compactação, mas certifique-se que o arquivo compactado é do tipo **zip** e não qualquer outro como **rar**, **tar**, **7z**, **tgz**, etc.

Envie o trabalho para o email e no prazo descritos na seção 2.2 (pág. 1).

- O assunto do email deve ser *Compiladores Portugol*.
- O corpo do email deve conter os números de matrícula e os nomes completos dos autores, p.ex.
201000123 Jose Carlos Pereira
201000456 Ana Maria Braga
- Anexe ao email o arquivo zipado para avaliação.

Se um trabalho for enviado mais de uma vez, apenas a versão enviada por último será corrigida; todas as anteriores serão descartadas. Em hipótese alguma serão admitidas correções, substituições ou ajustes de qualquer natureza após o prazo de entrega.

2.3. Aceitação

Só serão aceitos trabalhos entregues: a) dentro do prazo; b) pelos meios especificados; c) com as versões eletrônicas e impressas; d) com o *Formulário de Pré-Avaliação* devidamente preenchido.

2.4. Avaliação

O trabalho será defendido por cada grupo na data marcada, em uma espécie de avaliação oral (seção 8.1, pág. 17). O código fonte será então recompilado e executado pelo professor, e os resultados serão utilizados na correção do trabalho (seção 8, pág. 17). Trabalhos incompletos ou com erros receberão notas parciais, após os descontos apropriados. Não deixe de entregar o trabalho apenas porque está incompleto.

2.5. Segunda chamada

Quem faltar à defesa por justa causa pode solicitar segunda chamada no protocolo da UESC, anexando a devida justificativa (p.ex. atestado médico).

Se a 2ª chamada não for solicitada no protocolo ou se for indeferida pelo departamento, o aluno ou a dupla ausente à defesa terá nota zero no trabalho, sem direito a 2ª chamada.

Se os dois membros da dupla faltarem à defesa e seus pedidos de 2ª chamada forem deferidos, o trabalho não será avaliado, mesmo que tenha sido entregue no prazo. Ambos farão uma prova teórica, individual e sem consulta, no dia e hora das provas de 2ª chamada.

Se apenas um membro da dupla faltar à defesa e seu pedido de 2ª chamada for deferido, o trabalho só será avaliado se estiver dentro das condições descritas na seção 2.3 (pág. 3). Nesse caso, o aluno presente à defesa não será penalizado na nota pela ausência do parceiro; o ausente fará uma prova teórica, individual e sem consulta, no dia e hora das provas de 2ª chamada.

3. Requisitos do trabalho

O trabalho consiste na implementação de um analisador léxico para a linguagem de programação fictícia Portugol. O programa deve ser invocado na linha de comando conforme exemplificado abaixo:

Portugol prog01.ptg

onde **prog01.ptg** é um arquivo texto contendo código na linguagem Portugol. Nesse caso, seu programa deve gerar os seguintes arquivos de saída, todos do tipo texto:

prog01.ptg.err com o conteúdo do arquivo de entrada e os erros léxicos devidamente marcados,
prog01.ptg.tok com a lista de *tokens* reconhecidos no arquivo de entrada e
prog01.ptg.tbl com o conteúdo da tabela de símbolos após processar a entrada.

Seu analisador léxico deve:

- Reconhecer as palavras reservadas, operadores e delimitadores e utilizar códigos únicos para cada um.
- Criar e atualizar uma tabela de símbolos para os *tokens* que possuem lexemas.
- Ser implementado como uma sub-rotina (função ou método) que, a cada chamada:
 - Processa a entrada caractere por caractere, ainda que o arquivo seja lido todo de uma vez.
 - Identifica um *token* na entrada – exatamente um, nem mais, nem menos.
 - Retorna o código do *token* e, se for o caso, uma referência à sua entrada na tabela de símbolos.

Em um compilador real, o analisador sintático chama o léxico sempre que precisa de um *token*. Durante a análise sintática o compilador constrói com esses *tokens* uma representação do código fonte, geralmente uma árvore sintática. Implemente seu analisador léxico como uma função a ser chamada pelo analisador sintático sempre que este necessitar de um *token*. A função **main** deve simular o analisador sintático, chamando sucessivamente o léxico e obtendo um *token* de cada vez, até que não haja mais *tokens*. Não implemente no analisador léxico um laço que consome todos os *tokens* de uma vez.

Utilize o *token* **tkEOF** para sinalizar o fim do arquivo. Apesar deste trabalho não lidar com análise sintática, os *tokens* não serão descartados. Veja na seção 6.2 (pág. 12) a lista de *tokens* que o programa deve imprimir.

3.1. Modularização

O analisador léxico possuirá funcionalidades bem definidas e independentes umas das outras, como leitura do arquivo de entrada, reconhecimento de palavras reservadas, manutenção da tabela de símbolos, impressão de mensagens de erro e geração das saídas. Implemente essas funcionalidades distintas através de funções separadas e projete as passagens de parâmetros a fim de minimizar o uso de variáveis globais.

3.2. Autômato

O analisador léxico deve ser baseado em um AFD guiado por uma tabela de transições. Implemente-a como uma matriz contendo estados de transição, indexada pelo estado atual e pelo próximo símbolo na entrada. A seção 9.9 (pág. 27) contém código que ilustra esse princípio.

Palavras reservadas podem ser reconhecidas com: 1) um autômato simples que reconhece um identificador e depois decide se é palavra reservada, ou 2) um autômato complexo, com um estado para cada letra de cada palavra reservada, que reconhece palavras reservadas separadamente dos identificadores. Nos dois casos o autômato deve reconhecer todos os *tokens* da linguagem, e não apenas palavras reservadas e identificadores. Use o 1º método: não tente reconhecer palavras reservadas desde o início, procure apenas por cadeias que satisfaçam a definição de identificador. Ao encontrar uma, verifique se é palavra reservada com uma função que retorna o código da palavra reservada ou **-1** se o identificador não for palavra reservada. Tal função deve conhecer as palavras reservadas da linguagem. Alguns autores as colocam na tabela de símbolos, mas uma tabela própria e separada é mais apropriada.

O diagrama do autômato deve ser completo e legível. Estados de aceitação devem ser acompanhados de uma lista de ações associadas (p.ex. devolver caractere para a entrada, instalar *token* na tabela de símbolos, retornar *token* **tk_virg**). Os estados devem ser numerados e corresponder fielmente aos da tabela de transições.

Sugestão: Comece o trabalho projetando o autômato; isso facilitará muito a construção do código.

3.3. Tabela de símbolos

A tabela de símbolos deve conter informações apenas sobre *tokens* que possuem lexemas. Para cada *token*, armazene seu código numérico, seu lexema e suas ocorrências no arquivo de entrada (linha e coluna).

Uma combinação *token*–lexema deve ser incluída uma única vez na tabela de símbolos, ou seja, sem duplicação. A impressão da tabela de símbolos está descrita na seção 6.3 (pág. 13).

A tabela de símbolos deve ser implementada através de uma **tabela hash** indexada pela cadeia do lexema. Escolha um mecanismo de tratamento de colisões (endereçamento aberto ou encadeamento separado) e fundamente sua escolha na defesa do trabalho.

Sugestão: peça ao professor material didático para ajudá-lo na compreensão de tabelas *hash* em geral e na implementação da sua tabela *hash* em particular.

3.4. Geração das saídas

Em um compilador real, o analisador sintático não tem, em princípio, acesso direto ao programa de entrada, apenas aos *tokens* retornados pelo analisador léxico e a seus respectivos atributos, armazenados na tabela de símbolos. Neste trabalho, o analisador sintático é apenas minimamente simulado pela função **main** do programa principal, mas isso não muda as seguintes observações:

1. Algumas informações sobre o código fonte, no arquivo de entrada, não são acessíveis ao analisador sintático. Exemplo: natureza e posição dos erros léxicos.
2. Algumas informações sobre o código fonte, independentemente de serem ou não acessíveis ao analisador sintático, estão disponíveis apenas após o término da análise léxica, quando todo o arquivo de entrada tiver sido processado. Exemplo: quantidade e posição de lexemas duplicados no código fonte.

Essas observações têm implicações importantes para a geração das saídas descritas nas seções 6.1 a 6.3 (págs. 11 a 13). Assim, é necessário refletir sobre quando e como cada uma dessas saídas será produzida.

Considere a observação 1 acima. Segundo ela, o analisador léxico estaria em melhor posição que o sintático para produzir as três saídas especificadas. Mesmo que essas saídas sejam produzidas pelo analisador léxico, há pelo menos duas opções sobre o momento de fazê-lo:

1. Gerar as saídas simultaneamente, durante a análise léxica, à medida que o arquivo de entrada for lido e os *tokens* forem reconhecidos.
2. Gerar as saídas uma de cada vez, na última chamada ao analisador léxico, imediatamente após o reconhecimento do *token* de fim de arquivo.

É importante notar as implicações de cada uma dessas opções. Considere a listagem do arquivo de entrada e dos erros léxicos, descrita na seção 6.1 (pág. 11). Se as mensagens de erro forem impressas ao final da análise léxica (opção 2 acima), será necessário guardar uma lista de erros encontrados, com tipo e posição na entrada, em uma estrutura de dados adicional. Por outro lado, se as mensagens de erro forem impressas no instante em que os erros forem detectados (opção 1 acima), tal estrutura de dados adicional será desnecessária.

O mesmo se aplica à lista de *tokens* encontrados na entrada, descrita na seção 6.2 (pág. 12). Essa lista pode ser criada durante a análise léxica, armazenando os *tokens* em uma estrutura auxiliar, para ser impressa ao final da análise (opção 2 acima), mas ela também pode ser criada em paralelo à análise léxica, imprimindo *tokens* à medida que forem reconhecidos (opção 1 acima).

A impressão da tabela de símbolos, descrita na seção 6.3 (pág. 13), é mais problemática. Cada *token* inserido na tabela deve ser impresso uma única vez, acompanhado de uma lista de posições no arquivo de entrada onde ele ocorre. Isso significa que a impressão da tabela de símbolos não pode ocorrer em paralelo à análise léxica, à medida que os *tokens* forem reconhecidos (opção 1 acima).

Independente de como as saídas serão produzidas, é preciso garantir que as informações necessárias estarão disponíveis no local e momento certos. Isso pode exigir a alocação e gerenciamento de estruturas de dados adicionais, aumentando o consumo de memória e o tempo de execução do programa. Maior consumo de memória e tempo de execução podem ser o preço justo por um processamento mais adequado da entrada.

4. Especificação léxica da linguagem Portugol

4.1. Tipos de dados

Há três tipos de dados em Portugol: *inteiro*, *decimal* e *cadeia*, equivalentes, respectivamente, ao *int*, *float* e *string* de C. Não existe o tipo *char* nem modificadores de tipo como *short*, *long* e *unsigned* de C.

Um *inteiro* consiste de um ou mais dígitos, mas sem sinal, ponto decimal ou expoente. Assim, **-15** deve ser reconhecido como `tk_menos` e `tk_INT`; **+001** deve ser reconhecido como `tk_mais` e `tk_INT`.

Um *decimal* consiste de ao menos um dígito e exatamente um ponto decimal, sem sinal ou expoente. Assim, **-.15** deve ser reconhecido como `tk_menos` e `tk_DEC`; **+0.** deve ser reconhecido como `tk_mais` e `tk_DEC`. Note que **+.** deve ser reconhecido como `tk_mais` e, a seguir, produzir erro léxico no ponto: em Portugol, o ponto só pode ocorrer como parte de uma constante decimal com ao menos um dígito.

Uma *cadeia* é uma sequência de caracteres delimitada por aspas duplas e totalmente contida em uma única linha. Assim, **"-.15"** deve ser reconhecido como o *token* `tk_CADEIA`, bem como **"\n"** e **" "**.

Uma *cadeia* deve ser fechada na mesma linha em que foi aberta. Caso contrário, seu programa deve: 1) gerar o erro léxico “Cadeia não fechada”, e 2) retomar a análise no início da linha seguinte àquela onde a cadeia foi aberta, efetivamente transformando-a em uma cadeia válida, apesar de truncada.

4.2. Constantes e variáveis

Constantes podem ser de qualquer um dos três tipos acima. Os termos *constante* e *literal* são equivalentes, no sentido que constantes são valores numéricos ou *strings* que aparecem literalmente no código fonte.

Não há variáveis do tipo *cadeia* em Portugol, apenas do tipo *inteiro*, declaradas através da palavra reservada **int**, e do tipo *decimal*, declaradas através da palavra reservada **dec**. Uma declaração **int** ou **dec** pode conter múltiplas variáveis separadas por vírgulas.

Qualquer caractere é válido em uma cadeia, exceto aspas duplas. As aspas delimitam a cadeia, mas não fazem parte dela. Fica a seu critério armazená-las ou não na tabela de símbolos.

No exemplo ao lado há duas constantes: **2** (do tipo *inteiro*) e **Média=** (do tipo *cadeia*); seu analisador léxico deve reconhecê-las como tais.

```
media <- soma/2;  
imprima ("Média=");  
imprima (media);
```

4.3. Identificadores

Nomes de variáveis devem começar com uma letra e conter apenas letras, dígitos e sublinhado (veja seção 4.7, pág. 7). Letras maiúsculas e minúsculas são permitidas e distinguíveis: **media**, **Media**, **MeDiA** e **MEDIA** devem ser tratados como identificadores diferentes. Rejeite acentos no reconhecimento de identificadores: **media** deve ser aceito como identificador, mas **média** deve causar erro léxico e a mensagem de erro deve apontar o **é** como caractere inválido. Não há limite para o tamanho de identificadores; seu programa deve reconhecer identificadores de qualquer comprimento.

4.4. Palavras reservadas

Todos os comandos de Portugol são **palavras reservadas**. Maiúsculas e minúsculas são permitidas e indistinguíveis: **ate**, **aTe**, **AtE** e **ATE** são formas válidas da mesma palavra reservada. Rejeite acentos: **até** deve causar erro léxico e a mensagem de erro deve apontar o **é** como caractere inválido.

A sintaxe e semântica das palavras reservadas são irrelevantes para o analisador léxico; a descrição a seguir visa apenas ajudá-lo a compreender os arquivos de teste.

O comando **imprima** aceita variáveis ou constantes como argumentos; **leia** aceita apenas variáveis. Contudo, cada comando **leia** ou **imprima** aceita apenas um argumento. Assim, a impressão de texto e valores numéricos misturados deve ser feita com vários comandos **imprima**. Todo comando deve ser finalizado por ponto e vírgula, exceto os comandos **se** e **para**.

4.5. Operadores

	Unários				Binários					
Aritméticos e lógicos	-	nao	++	--	+	-	*	/	e	ou
Relacionais					=	<>	>	>=	<	<=
Atribuição					<-					

4.6. Gramática

Programa → **inicio** *Declarações Comandos* **fim**
Declarações → *Declaração Declarações*
 / ϵ
Declaração → **Tipo** : *Identificadores* ;
Tipo → **int**
 / **dec**
Identificadores → *Identificador , Identificadores*
 / *Identificador*
Comandos → *Comando Comandos*
 / ϵ
Comando → *CMD-Se / CMD-Para / CMD-Obtenha / CMD-Mostre / CMD-Atrib*
CMD-Se → **se** *Expr* **entao** *Comandos* **senao** *Comandos* **fim_se**
 / **se** *Expr* **entao** *Comandos* **fim_se**
CMD-Para → **para** *Identificador de Expr ate Expr passo Expr Comandos* **fim_para**
 / **para** *Identificador de Expr ate Expr Comandos* **fim_para**
CMD-Obtenha → **leia** (*Identificador*) ;
CMD-Mostre → **imprima** (*Identificador*) ;
 / **imprima** (*Número*) ;
 / **imprima** (*Cadeia*) ;
CMD-Atrib → *Identificador* <- *Expr* ;
Expr → *Expr OpBinario Expr*
 / *OpUnario Expr*
 / *Número*
OpBinario → + / - / * / / / e / ou / = / <> / > / >= / < / <=
OpUnario → - / ++ / -- / nao
Número → *Inteiro*
 / *Decimal*

4.7. Expressões regulares

Inteiro: *Dígito +*
Decimal: *[[Dígito * . Dígito +] | [Dígito + . Dígito *]]*
Identificador: *Letra [Letra | Dígito | _] **
Cadeia: *" [^"] "* (qualquer caractere entre aspas duplas, inclusive letras acentuadas, @, # etc.)
Letra: *[a-zA-Z]* (maiúscula ou minúscula sem acentos)
Dígito: *[0-9]*

4.8. Delimitadores

Pode-se dizer que um **delimitador** é um “marcador de fim”, mas o significado exato do termo varia, dependendo do contexto em que é utilizado e também do autor que trata do tema.

Do ponto de vista sintático, um delimitador é um *token* que marca o fim de uma construção sintática ou um bloco de comandos; geralmente ocorre no fim de uma regra gramatical. Em Portugal, os seguintes *tokens* são delimitadores sintáticos: `tk_fim`, `tk_fim_se`, `tk_fim_para`, `tk_virg` e `tk_pt_virg`.

Do ponto de vista léxico, um delimitador marca o fim de um *token*. Mais precisamente, um delimitador é um caractere válido da linguagem que não pode fazer parte do *token* que o antecede. A partir de agora usaremos o termo **delimitador** somente com este significado; mais adiante adotaremos uma definição mais estrita.

Branco é um termo que se aplica ao espaço (' '), ao *tab* (`\t`) e à quebra de linha (`\n`). Brancos são delimitadores, assim como comentários. Uma sequência de espaços, *tabs*, quebras de linha e comentários, em qualquer combinação, é equivalente a um único branco.

O único efeito de um branco é delimitar o *token* que o antecede. Uma vez que esse efeito é obtido, o branco deve ser descartado. Note, contudo, que o *token* imediatamente após o branco deverá ter sua posição (linha e coluna) registrada corretamente. Para isso, é necessário

- incrementar o contador de colunas ao encontrar um espaço
- zerar o contador de colunas e incrementar o contador de linhas ao encontrar uma quebra de linha.

O *tab* é mais problemático. Seu significado (avanço de múltiplas colunas) é determinado pelo editor de texto usado quando ele foi inserido no programa Portugal. Ao encontrar um *tab*, o analisador léxico não tem meios de determinar quantas colunas ele avançou originalmente; qualquer interpretação será arbitrária. Assim, para todos os efeitos, um *tab* deve ser tratado como um único espaço.

Considere a seguinte linha de um programa Portugal:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
p	a	r	a	\t		d	e	\t	1	a	t	e		x	p	a	s	s	o	1		s	(*	*	+	+	*)	e			2	<	3	\n

O analisador léxico deve reconhecer nessa linha os seguintes *tokens* e lexemas:

Token	Lexema	Coluna	Erro
tk_para	para	1	
tk_de	de	7	
tk_INT	1	10	Delimitador esperado na coluna 11
tk_ate	ate	11	
tk_IDEN	xpassol	15	
tk_IDEN	s	23	
tk_IDEN	e	31	
tk_INT	2	34	
tk_menor	<	35	
tk_INT	3	36	

Os *tokens* de Portugal obedecem a **regra da cadeia mais longa**: se uma cadeia de caracteres puder representar um único *token* mais longo e também vários *tokens* mais curtos consecutivos, então o *token* mais longo deverá ser escolhido. Essa regra é útil porque elimina ambiguidades na interpretação da entrada.

Considere a seguinte linha de um programa Portugal:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
p	a	r	a	i	d	e	1		a	t	e	9	p	a	s	s	o		1	.	2	.	3	.	<	-	-	-	-	(*	*	*))	\n

O analisador léxico deve reconhecer nessa linha os seguintes *tokens* e lexemas:

Token	Lexema	Coluna	Erro
tk_IDEN	paraide1	1	
tk_IDEN	ate9passo	10	
tk_DEC	1.2	20	
tk_DEC	.3	23	
tk_atrib	<-	26	Ponto isolado na coluna 25
tk_decr	--	28	
tk_menos	-	30	
tk_fecha_par)	36	

Note que, em Portugal, um delimitador pode:

- não corresponder a *token* algum (brancos e comentários),
- ser um *token* por si próprio (p.ex. / em tk_dividido, e em tk_e), ou
- ser o primeiro de vários caracteres de um *token* (p.ex. < em tk_menor_igual, n em tk_nao).

A partir de agora usaremos o termo **delimitador** com o significado mais estrito de **branco** ou **comentário**. Assim, um delimitador não corresponde a *token* algum; seu único efeito é terminar *tokens*.

O uso de delimitadores em Portugal só é necessário quando, em sua ausência, uma porção do código fonte

- tiver interpretação ambígua,
- resultar em erro léxico, ou
- resultar em erro sintático.

No exemplo anterior, a ausência de delimitadores nas cadeias **paraide1** e **ate9passo** resultaria em erro sintático. No mesmo exemplo, a ausência de delimitadores na cadeia **1.2.3.** resultaria em erro sintático (os números decimais 1.2 e 0.3 em sequência) e também em erro léxico (um ponto isolado após o decimal 0.3).

Considere a seguinte linha de um programa Portugal:

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
p a r a i d e 1 a t e 9 . 0 . p a s s o 1 $ j \n

```

O analisador léxico deve reconhecer nessa linha os seguintes *tokens* e lexemas:

Token	Lexema	Coluna	Erro
tk_para	para	1	
tk_IDEN	i	6	
tk_de	de	8	
tk_INT	1	11	Delimitador esperado na coluna 12
tk_ate	ate	12	
tk_DEC	9.0	16	
tk_passo	passo	21	Ponto isolado na coluna 19
tk_INT	1	27	Caractere inválido na coluna 28
tk_IDEN	j	29	

Note que o analisador léxico deve gerar o erro “Delimitador esperado” na coluna 12. Caso contrário, o analisador sintático receberá os *tokens* tk_INT e tk_ate e não perceberá nenhum erro nesta parte do código.

4.9. Comentários

Comentários devem ser ignorados pelo analisador léxico apenas para efeito de reconhecimento de *tokens*, não para efeitos de impressão. Ou seja, todos os comentários devem aparecer integralmente na saída contendo a listagem da entrada e os erros léxicos. Assim, ao encontrar o início de comentário, o analisador deve consumir e guardar caracteres da entrada até encontrar o final do comentário, imprimindo-o na saída apropriada, e depois deve retomar a busca do próximo *token*.

Existem dois tipos de comentários: de linha e de bloco. Comentários de linha são iniciados pelos caracteres `/*` e terminam no final da linha, por exemplo:

```
inteiro: x, y, z;      /* Tres valores a serem ordenados
inteiro: maior, menor; /* O maior e o menor dos tres
```

Comentários de bloco são demarcados pelas sequências `(*` e `*)`, por exemplo:

```
(* Este programa le tres numeros *
 * e os coloca em ordem crescente *)
inteiro: x, y, z, maior, menor;
```

Comentários de bloco não podem ser aninhados. Por exemplo, no código a seguir

```
(* Comentario de fora
   (* Comentario de dentro *)
*)
```

o analisador deve descartar as duas primeiras linhas e reconhecer dois tokens: `tk_vezes` e `tk_fecha_par`.

Um comentário de bloco não fechado deve gerar o erro léxico “Comentário de bloco não fechado”. A análise deve ser retomada no início da linha seguinte àquela onde o comentário foi aberto, efetivamente transformando-o em um comentário de linha válido, apesar de truncado.

5. Tratamento de erros

O tratamento de erros léxicos varia consideravelmente de uma linguagem para outra. Uma estratégia de tratamento de erros pode ser vista como três tarefas interligadas e realizadas em sequência:

1. **Identificação** – consiste em determinar a natureza e localização do erro (linha e coluna) no texto do código fonte.
2. **Processamento** – consiste em:
 - decidir qual porção da cadeia lida aceitar como parte do *token* reconhecido, qual porção descartar, e qual porção preparar para ser lida novamente na próxima chamada
 - processar normalmente o *token* (instalar na tabela de símbolos, emitir mensagens várias, etc)
 - emitir mensagem diagnóstica precisa e detalhada sobre o erro identificado.
3. **Recuperação** – consiste em consumir e descartar apenas as porções adequadas da cadeia lida, possivelmente devolvendo à entrada caracteres lidos até o momento da detecção do erro. O objetivo da recuperação de erro é permitir que a análise léxica prossiga, ao invés de abortar a compilação. Em geral, a melhor estratégia de tratamento de erros é aquela que descarta o mínimo da entrada, de forma a construir um *token* com lexema o mais longo possível.

Como a recuperação de erro se dá depende do tipo do erro e de quando foi detectado. Se detectado *antes* de um possível *token* válido ter sido encontrado (p.ex. `$` de início, ou comentário de bloco não fechado), então

- Descarte a porção inválida da entrada.
- Emita mensagem de erro adequada, p.ex. “Caractere inválido” ou “Comentário de bloco não fechado”.
- Retome o reconhecimento no próximo caractere da entrada, retornando ao estado inicial do autômato.

Se o erro foi detectado *depois* de um possível *token* válido ter sido encontrado (p.ex. `1$` ou `1a` ou `1.2. mas não 1+` ou `1/+` pois nesses dois exemplos não há erro), então

- Termine o lexema no caractere que antecede aquele que causou o erro.
- Decida se o processamento do erro
 - deve ser feito na chamada atual (p.ex. `1a`),
 - pode ser feito na chamada atual ou na seguinte (p.ex. `1$`), ou
 - deve ser feito na chamada seguinte (p.ex. `1.2.`)
- Para processar o erro na chamada atual, emita a mensagem de erro apropriada.
- Devolva à entrada os caracteres que causaram o erro.
- Instale o *token* na tabela de símbolos.
- Retorne o *token* encontrado.

Ao reportar um erro léxico o analisador deve fornecer, logo abaixo da linha de código correspondente, uma descrição clara e completa da posição e da natureza do erro. Um exemplo é fornecido na seção 6.1 (pág. 11).

Não modifique os arquivos de teste; use-os exatamente como estão, mesmo com caracteres ou comandos inexistentes na linguagem ou comentários com sintaxe incorreta. Não procure nos arquivos de teste nenhuma indicação sobre a ortografia ou a sintaxe correta de Portugol: os testes podem conter erros deliberadamente.

6. Impressão dos resultados para o relatório

A seção 3 (pág. 4) lista três saídas que devem ser geradas para cada arquivo de teste. A seguir, descrições detalhadas dessas saídas, ilustradas com exemplos produzidos a partir do programa Portugol na Figura 1.

Figura 1 – Exemplo de programa Portugol incorreto.

```
(* Ex-00-incorreto.ptg *)  
inicio  
  int: @número, r;  
  imprima ("Digite um nro:");  
  leia (numero);  
  r <- raiz (número);  
  imprima(r);  
fim
```

6.1. Arquivo de entrada e erros léxicos

A primeira saída consiste do conteúdo original da entrada (arquivo de teste completo) com todas as linhas numeradas e os erros léxicos marcados apropriadamente, imediatamente após as linhas que contêm os erros. Essa marcação deve consistir de:

- uma localização visual da posição do erro,
- as coordenadas da posição do erro (números da linha e da coluna onde o erro se encontra), e
- uma descrição clara e precisa da natureza do erro.

Após sinalizar um erro léxico o analisador deve continuar a processar o arquivo de entrada até encontrar o próximo *token*. O analisador não deve retornar *token* de erro. Todos os comentários de linha e de bloco devem ser incluídos na saída (veja seção 4.9, pág. 9).

Ao processar o programa na Figura 1, seu analisador léxico deve gerar a seguinte saída:

```
LISTA DE ERROS LEXICOS EM "Ex-01-incorreto.ptg"  
  
[ 1] (* Ex-00-incorreto.ptg *)  
[ 2]  
[ 3] inicio  
[ 4]   int: @número, r;  
      -----^  
      Erro lexico na linha 4 coluna 8: Caracter invalido '@'  
[ 4]   int: @número, r;  
      -----^  
      Erro lexico na linha 4 coluna 10: Caracter invalido 'ú'  
[ 5]     imprima ("Digite um nro:");  
[ 6]     leia (numero);  
[ 7]     r <- raiz (número);  
      -----^  
      Erro lexico na linha 7 coluna 15: Caracter invalido 'ú'  
[ 8]     imprima(r);  
[ 9] fim  
  
TOTAL DE ERROS: 3
```

6.2. Lista de tokens reconhecidos

A segunda saída consiste de uma lista de *tokens* reconhecidos em cada linha da entrada, com as linhas numeradas. Para cada *token*, imprima seu nome, seu código numérico e, quando for o caso, a posição do respectivo lexema na tabela de símbolos. Após a lista de *tokens* reconhecidos, seu analisador léxico deve gerar um resumo mostrando o número total de ocorrências de cada *token*.

Ao processar o programa na Figura 1, seu analisador léxico deve gerar a seguinte saída (ou algo parecido, dependendo da estratégia de correção de erros que você empregar):

LISTA DE TOKENS RECONHECIDOS EM "Ex-01-incorreto.ptg"

LIN	COL	COD	TOKEN	LEXEMA	POS	TAB	SIMB
3	1	6	tk_inicio				
4	3	8	tk_int				
	6	26	tk_dois_pts				
	9	2	tk_IDEN	n			1
	11	2	tk_IDEN	mero			2
	15	24	tk_virg				
	17	2	tk_IDEN	r			3
	18	25	tk_pt_virg				
5	3	11	tk_imprima				
	11	27	tk_abre_par				
	12	5	tk_CADEIA	"Digite um nro:"			4
	28	28	tk_fecha_par				
	29	25	tk_pt_virg				
6	3	10	tk_leia				
	8	27	tk_abre_par				
	9	2	tk_IDEN	numero			5
	15	28	tk_fecha_par				
	16	25	tk_pt_virg				
7	3	2	tk_IDEN	r			3
	5	37	tk_atrib				
	8	2	tk_IDEN	raiz			6
	13	27	tk_abre_par				
	14	2	tk_IDEN	n			1
	16	2	tk_IDEN	mero			2
	20	28	tk_fecha_par				
	21	25	tk_pt_virg				
8	3	11	tk_imprima				
	10	27	tk_abre_par				
	11	2	tk_IDEN	r			3
	12	28	tk_fecha_par				
	13	25	tk_pt_virg				
9	1	7	tk_fim				
10	0	1	tk_EOF				

RESUMO

COD	TOKEN	USOS
1	tk_EOF	1
2	tk_IDEN	9
3	tk_INTEIRO	0
4	tk_DECIMAL	0
5	tk_CADEIA	1
6	tk_inicio	1
7	tk_fim	1
8	tk_int	1
9	tk_dec	0
10	tk_leia	1
11	tk_imprima	2
12	tk_para	0

Note como todas as colunas possuem a **largura exata** para seu conteúdo: mais estreita, e algum conteúdo seria truncado; mais larga, e sobriariam espaços ao redor do conteúdo mais largo.

Certifique-se que todas as saídas em formato de tabela possuem colunas com a largura exata, conforme o exemplo.

OBSERVAÇÃO:

O programa modelo fornecido pelo professor (veja seção 8.5, pág. 18) pode determinar a largura das colunas de maneira diferente. Se isso acontecer, ignore o tratamento dado pelo programa modelo e implemente seu programa exatamente como descrito acima.

13	tk_de	0
14	tk_ate	0
15	tk_passo	0
16	tk_fim_para	0
17	tk_se	0
18	tk_entao	0
19	tk_senao	0
20	tk_fim_se	0
21	tk_e	0
22	tk_ou	0
23	tk_nao	0
24	tk_virg	1
25	tk_pt_virg	5
26	tk_dois_pts	1
27	tk_abre_par	4
28	tk_fecha_par	4
29	tk_menor	0
30	tk_menor_igual	0
31	tk_maior	0
32	tk_maior_igual	0
33	tk_diferente	0
34	tk_igual	0
35	tk_incr	0
36	tk_decr	0
37	tk_atrib	1
38	tk_mais	0
39	tk_menos	0
40	tk_vezes	0
41	tk_dividido	0
+-----+-----+-----+		
0	TOTAL	33
+-----+-----+-----+		

TOTAL DE ERROS: 3

6.3. Tabela de símbolos

A terceira saída consiste do conteúdo da tabela de símbolos ao final do processamento do arquivo de entrada. Ao processar o programa na Figura 1, seu analisador léxico deve gerar a seguinte saída (ou algo parecido, dependendo da estratégia de correção de erros que você empregar):

TABELA DE SIMBOLOS - "Ex-01-incorreto.ptg"

POS	TOKEN	LEXEMA	POS NA ENTRADA (linha,coluna)
1	tk_IDEN	n	(4, 9) (7, 14)
2	tk_IDEN	mero	(4, 11) (7, 16)
3	tk_IDEN	r	(4, 17) (7, 3) (8, 11)
4	tk_CADEIA	"Digite um nro:"	(5, 12)
5	tk_IDEN	numero	(6, 9)
6	tk_IDEN	raiz	(7, 8)

Note que os pares *token*-lexema devem ser listados na ordem em que ocorreram na entrada. Certamente eles ocuparão posições aleatórias na tabela *hash* que contém a tabela de símbolos, portanto é necessário criar um mecanismo que conecte, para cada par *token*-lexema, a *ordem* em que ele ocorre na entrada e sua *posição* na tabela de símbolos.

Note, ainda, que cada par *token*-lexema deve ocupar apenas uma linha na tabela impressa. Isso significa que a largura da tabela dependerá do número de ocorrências do lexema mais frequente na entrada.

7. Realização

Esta seção oferece sugestões para facilitar a realização do trabalho. Não é necessário segui-las, mas é muito importante **planejar antes de realizar**. Durante o planejamento, seja objetivo, específico, metódico e realista. Acima de tudo, resista à tentação de começar a escrever código antes de completar o plano. O trabalho exige muito mais que apenas escrever código, tal como desenhar o autômato, produzir as saídas dos testes, montar o relatório e preencher o formulário de pré-avaliação

7.1. Plano de ação

Comece elaborando um plano de ação dividido em fases. Particione o trabalho em tarefas ou fases conceitualmente bem delineadas, listando-as em alguma ordem lógica e cronológica. Por exemplo:

1. Projeto do autômato
2. Construção do autômato
3. Projeto da tabela de transições
4. Construção da tabela de transições
5. Projeto da tabela de palavras reservadas
6. Construção da tabela de palavras reservadas
7. Projeto da tabela de símbolos
8. Construção da tabela de símbolos
9. Projeto da impressão da entrada Portugal
10. Implementação da impressão da entrada Portugal
11. Projeto da impressão da lista de tokens reconhecidos
12. Implementação da impressão da lista de tokens reconhecidos
13. Projeto da impressão da tabela de símbolos
14. Implementação da impressão da tabela de símbolos
15. Integração de todos os módulos
16. Teste integrado do programa
17. Construção do relatório
18. Preenchimento do formulário de pré-avaliação

Note que essas fases não serão necessariamente concluídas nessa exata ordem, pois

- fases mutuamente dependentes devem ser realizadas na ordem correta (p.ex. fase 1 antes da fase 2);
- fases independentes mas relacionadas podem ser melhor realizadas em uma ordem específica (p.ex. fases 5 e 14 antes da fase 6, de modo que, ao terminar de implementar a tabela de símbolos, será possível testá-la imediatamente com as funções de impressão já implementadas);
- fases independentes e não relacionadas podem ser realizadas em qualquer ordem, inclusive simultaneamente (p.ex. fases 3 e 5).

A Tabela 1 (pág. 15) mostra um exemplo, apenas hipotético e incompleto, de plano de ação. Use esse exemplo como ponto de partida para seu próprio plano de ação.

Tabela 1 – Exemplo de plano de ação.

FASE 1 – Projeto do autômato	
Ações <ul style="list-style-type: none"> Definir lista de <i>tokens</i> a serem reconhecidos Definir expressões regulares dos respectivos lexemas Desenhar a mão o diagrama do autômato, com os estados e transições necessários para reconhecer cada <i>token</i> Definir ações necessárias à construção dos lexemas e ao reconhecimento dos <i>tokens</i> <ul style="list-style-type: none"> devolver caractere para a entrada emitir mensagem de erro instalar lexema na tabela de símbolos retornar <i>token</i> (<i>tk_IDEN</i>, <i>tk_INT</i>, <i>tk_EOF</i>, etc) etc. Definir protótipos das funções que realizarão as ações acima Identificar ações que devem ser realizadas em cada estado Identificar os estados finais do autômato 	Produtos <ul style="list-style-type: none"> Diagrama do autômato <ul style="list-style-type: none"> legível e completo com expressões regulares dos caracteres de transição com ações realizadas em cada estado com legenda
FASE 2 – Construção do autômato	
Ações <ul style="list-style-type: none"> Desenhar diagrama do autômato com algum programa de desenho vetorial (Inkscape, Visio, PowerPoint, etc) Salvar em dois formatos: bitmap e vetorial 	Produtos <ul style="list-style-type: none"> Diagrama do autômato <ul style="list-style-type: none"> salvo em formato bitmap (<i>jpg</i> ou <i>png</i>) com resolução mínima de 300dpi salvo também em formato vetorial (<i>svg</i> ou <i>emf</i>)
...	
FASE 7 – Projeto da tabela de símbolos (TS)	
Ações <ul style="list-style-type: none"> Definir as informações que devem ser armazenadas na TS para cada lexema encontrado na entrada, p.ex. <ul style="list-style-type: none"> cadeia do lexema posição na entrada onde ocorreu Definir estrutura de dados p/ armazenar um lexema na TS, p.ex. <i>struct { ... }</i> Selecionar um mecanismo de tratamento de colisões (endereçamento aberto ou encadeamento separado) Definir os parâmetros da tabela <i>hash</i> <ul style="list-style-type: none"> tamanho do <i>array</i> código <i>hash</i> função de compressão Definir as funções de acesso à TS, p.ex. <ul style="list-style-type: none"> buscar lexema inserir novo lexema inserir nova ocorrência de lexema existente Definir protótipos das funções acima 	Produtos <ul style="list-style-type: none"> Estruturas de dados que conterão a TS Protótipos das funções de acesso
FASE 8 – Construção da tabela de símbolos	
Ações <ul style="list-style-type: none"> Implementar as funções de acesso à TS Testar as funções de acesso à TS 	Produtos <ul style="list-style-type: none"> Funções de acesso à TS, implementadas e testadas

7.2. Cronograma

O próximo passo é montar um cronograma. Há várias formas de fazê-lo. Por exemplo, liste as fases do plano na ordem em que serão realizadas, depois conte o número de dias ou semanas até a data da entrega e distribua o tempo disponível entre todas as fases. Crie algum mecanismo para registrar períodos de trabalho individual e períodos de trabalho conjunto. Deixe uma margem de segurança na distribuição do tempo, pois é provável que algumas tarefas demorem mais do que planejado.

A Tabela 2 (pág. 16) mostra um exemplo, apenas hipotético e incompleto, de cronograma. Use esse exemplo como ponto de partida para seu próprio cronograma de trabalho.

Tabela 2 – Exemplo de cronograma de trabalho.

A = Ana Maria J = José Carlos	Semanas e datas					
	1	2	3	4	5	6
Início	12/08	19/08	26/08	02/09	09/09	16/09
Fim	18/08	25/08	01/09	08/09	15/09	22/09
Fases						
1 Autômato – Projeto	A J					
2 Autômato – Construção	A					
3 Tabela de transições – Projeto	A J					
4 Tabela de transições – Construção	J					
5 Tabela de palavras reservadas – Projeto	A J					
6 Tabela de palavras reservadas – Construção	J					
7 Tabela de símbolos – Projeto		A J				
8 Tabela de símbolos – Construção		A				
9 Impressão da entrada Portugol – Projeto		A J				
10 Impressão da entrada Portugol – Implementação		J				
11 Impressão da lista de tokens reconhecidos – Projeto			A J			
12 Impressão da lista de tokens reconhecidos – Implementação			J			
13 Impressão da tabela de símbolos – Projeto			A J			
14 Impressão da tabela de símbolos – Implementação			A			
15 Integração de todos os módulos				A J		
16 Teste integrado do programa				A J	A J	A J
17 Construção do relatório						A J
18 Preenchimento do formulário de pré-avaliação						A J

7.3. Desenvolvimento

O paradigma de programação orientada a objetos presta-se muito bem a este trabalho. Apesar da linguagem C não ser orientada a objetos, é possível aplicar princípios básicos do paradigma, como

- separação entre estruturas de dados e funções de acesso (encapsulamento de dados),
- separação entre objetos e métodos públicos (interface) e privados (implementação),
- modularização do código (diversos arquivos `.c` e `.h` contendo declarações e definições correlatas).

Conceba o problema (realizar a análise léxica de um programa Portugol) e sua solução em termos de

- objetos,
- mensagens trocadas por objetos, e
- seus respectivos métodos de acesso.

Projete seu código a partir dessa perspectiva, sempre na direção do abstrato (alto nível) ao concreto (baixo nível). Conclua todo o projeto e só então comece a escrever código propriamente dito. Isso facilitará imensamente o desenvolvimento e teste do seu programa.

8. Avaliação

8.1. Defesa

Cada trabalho será defendido oralmente pelos autores através de questionamentos individuais sobre:

- A qualidade do código (legibilidade, uso de comentários, nomes de identificadores, etc.).
- Os algoritmos implementados e sua lógica.
- As estruturas de dados utilizadas e suas justificativas.
- Os resultados dos testes.
- As escolhas feitas para lidar com situações inesperadas ou não descritas neste documento.

Um dos objetivos da defesa é demonstrar a autoria do trabalho. Não é necessário que os dois autores implementem juntos todo o código; cada um pode implementar uma parte, mas ambos devem conhecer o conjunto em detalhes. Não saber ou não gostar de programar em C *não é* um argumento aceitável num curso de graduação em Ciência da Computação.

Um trabalho será bem defendido se cada autor demonstrar claramente que conhece todo o código e for capaz de justificar todas as decisões de implementação adotadas. Um trabalho mal defendido sofrerá um desconto subjetivo na nota *após* a aplicação dos critérios objetivos de avaliação descritos a seguir.

8.2. Discrepâncias

Cada trabalho deve ser acompanhado do código fonte, resultados de testes e relatório impresso. O código fonte será compilado pelo professor (veja seção 8.3, pág. 17) e usado no processamento dos três arquivos de testes. Os resultados serão comparados com os resultados dos testes entregues com o trabalho.

Discrepâncias resultarão em descontos na nota, conforme a Tabela 3 (pág. 20). Especificamente, serão verificadas discrepâncias nas comparações entre:

- Código fonte impresso e código fonte entregue em formato eletrônico.
- Resultados dos testes impressos, dos testes entregues em formato eletrônico, e dos testes gerados pelo executável recompilado.

No evento de qualquer discrepância, o professor decidirá o que será corrigido e o que será descartado. Em hipótese alguma serão admitidas correções, substituições ou ajustes de qualquer natureza após a entrega.

8.3. Compilação

Seu código fonte será recompilado em Linux com GCC versão 4.8.5 ou superior, com as mesmas *flags* de compilação usadas no *makefile* fornecido para o desenvolvimento do trabalho:

```
-ansi -pedantic -std=c99 -Wall -Wstrict-prototypes -Wmissing-prototypes -Wnested-externs -O2
```

O principal objetivo dessas *flags* é instruir GCC a identificar o maior número possível de construções inválidas ou suspeitas no código.

Warnings indicam inconsistências ou omissões suspeitas no código fonte, provavelmente resultantes de erros de programação, com o potencial de gerar código semanticamente incorreto. *Warnings* não impedem a geração de código executável; tampouco significam que o código executável necessariamente produzirá erros de execução ou resultados incorretos. **Erros de compilação** resultam de violações da sintaxe ou semântica da linguagem de tal severidade que impedem a geração de código executável. **Erros de execução** são caracterizados pelo término anormal ou prematuro do programa, geralmente como consequência de uma violação de acesso a memória. Essa definição não diz respeito à qualidade ou correção dos resultados produzidos pelo programa, mas à normalidade de sua execução e término. Assim, um programa que produz resultados incorretos será corrigido, desde que seu término seja normal, ou seja, ocorra no tempo certo.

Se, ao compilar seu código, GCC gerar erros de compilação, será impossível testar seu programa. Nesse caso, o trabalho não será corrigido e a nota será zero. Se o compilador não gerar erros mas gerar *warnings*, o trabalho será corrigido normalmente, mas haverá desconto na nota.

8.4. Testes

A avaliação do trabalho baseia-se, em grande parte, em testes do programa. Além dos três arquivos de testes fornecidos, serão usados programas em Portugol contendo uma variedade de erros léxicos. Todos os testes serão automatizados, com um mínimo de intervenção manual, num processo envolvendo:

- Compilação do código fonte com *makefile* semelhante ao fornecido com esta especificação.
- Invocação do executável passando os arquivos de teste como parâmetros (veja seção 9.4, pág. 25).
- Redirecionamento das saídas de todas as execuções para pastas específicas.
- Comparação textual das saídas assim geradas com as saídas esperadas.
- Classificação e contagem das diferenças encontradas.

Naturalmente a compilação e execução de seu código deverão se ajustar ao procedimento de testes automatizados. Assim, o trabalho não será corrigido e a nota será zero em qualquer dos casos abaixo:

- Se o código fonte entregue em formato eletrônico
 - der erro de compilação;
 - não gerar um executável com o nome esperado.
- Se o código executável compilado a partir do código fonte entregue em formato eletrônico
 - der erro de execução;
 - não aceitar o nome do arquivo de entrada pela linha de comando junto à invocação do programa;
 - não gerar arquivos de saída na pasta ou com os nomes esperados.

A fim de evitar transtornos desnecessários, certifique-se do seguinte:

- A versão final do código fonte foi usada para efetuar seus testes.
- As versões impressa e eletrônica do código fonte e dos testes são idênticas.
- O nome do executável gerado pelo compilador é **Portugol**.
- Os arquivos de saída são criados na pasta onde residem o executável e os arquivos de entrada.
- Os arquivos de saída são criados com os nomes esperados, conforme especificado na pág. 4.
- As saídas não contêm nada a mais ou a menos que o especificado neste documento.
- As saídas são formatadas exatamente como especificado neste documento, não só para os arquivos de teste fornecidos, mas para qualquer entrada. Crie seus próprios testes, com lexemas em quantidade e de comprimento variáveis, e observe se as saídas preservam a formatação correta. Verifique se nas tabelas de *tokens* reconhecidos e de símbolos as colunas têm larguras compatíveis com os conteúdos.

O mais indicado é escrever o código desde o início em Linux, em qualquer ambiente de desenvolvimento baseado no GCC versão 4.4.5 ou superior, ou apenas um editor de texto e nada mais. Não use nenhuma biblioteca estática ou dinâmica que não faça parte da biblioteca padrão de GCC em Linux. Use o *makefile* fornecido para compilar o código e eliminar erros e *warnings*. Use o mesmo *makefile* para realizar os testes.

8.5. Modelo

Observe atentamente os resultados dos testes fornecidos como modelo. Use também como referência o executável fornecido pelo professor, invocado numa console de Linux ou linha de comando de Windows com

```
./Portugol-Modelo -d Portugol.mel -p prog-qualquer.ptg
```

Execute-o com diversos arquivos de entrada e observe o conteúdo e formatação de tudo que é gravado nos arquivos de saída. Seu programa deverá reproduzir fielmente o funcionamento do programa modelo. Construa suas saídas impressas de forma idêntica, nos mínimos detalhes, até as larguras das tabelas e os espaços em branco. Qualquer desvio ou inconsistência, por menor que seja, poderá resultar em desconto na nota; a única exceção está descrita na “Observação” na pág. 12.

É possível que este documento contenha inconsistências, ambiguidades, omissões ou erros que causem dúvidas sobre o comportamento esperado do programa. Nesse caso, consulte o professor o mais rápido possível pelo email coletivo da disciplina, de modo que todos se beneficiem da discussão. *Não tome o programa modelo como referência para resolver dúvidas ou inconsistências; esclareça-as explicitamente com o professor.*

8.6. Critérios de avaliação

O trabalho será avaliado segundo 31 critérios divididos em 3 grupos e 10 sub-grupos:

Grupos			
Autômato (imagem)		Implementação	Saídas
Sub-grupos	– Diagrama	– Tokens	– Listagem da entrada e erros
	– Tabela de transições	– Palavras reservadas	– Lista de tokens reconhecidos
		– Transições entre estados	– Tabela de símbolos
		– Tabela de símbolos	
		– Comentários	

8.7. Pontuação

Os critérios de avaliação estão detalhados na Tabela 3 (pág. 20). A coluna **Peso** contém os pesos relativos de cada um dos 31 critérios. A coluna **Ocorrência** lista os erros e tudo o mais que será penalizado, seguidos dos pontos deduzidos em cada critério (coluna **Desc**). Após os descontos, a pontuação obtida em cada sub-grupo e em cada grupo de critérios se limitará à faixa de 0 a 10.

Um erro ou omissão pode gerar diversos descontos. Considere, a título de exemplo, a avaliação de trabalhos entregues em semestres anteriores. A Figura 2 (pág. 21) mostra os erros cometidos por três duplas em seus trabalhos. A Figura 3 (pág. 22) mostra os descontos efetuados nas notas de todos os trabalhos da turma. A Figura 4 (pág. 22) mostra as notas obtidas.

8.8. Casos omissos

A avaliação do trabalho não será limitada aos critérios aqui descritos. Poderá haver descontos na nota em consequência de situações que, embora não previstas neste documento, demonstrem uma compreensão ou execução incorreta ou imperfeita do trabalho. Nesses casos, os descontos serão devidamente justificados e lançados na coluna **Desconto posterior** (Figura 4, pág. 22).

8.9. Desempenho na defesa

Um dos objetivos da defesa é relevar até que ponto cada indivíduo se empenhou na realização do trabalho. Não é necessário que os dois membros da dupla escrevam todo o código juntos, mas é preciso que cada um conheça intimamente todo o conjunto, incluindo as partes que não tenha criado pessoalmente.

Uma vez calculada a nota do trabalho, ela será particionada entre os dois autores na proporção dos esforços depreendidos por cada um. Se, durante a defesa, ficar claro que ambos se empenharam igualmente, a nota será particionada igualmente: ambos receberão a nota obtida pelo trabalho. Caso contrário, uma porção da nota será deduzida de um autor e atribuída ao outro. Esse particionamento da nota do trabalho será lançado na coluna **Divisão da nota (após defesa)** (Figura 4, pág. 22).

Tabela 3 – Critérios de avaliação e descontos na nota.

	Peso	Critério	Cód	Ocorrência	Desc
Autômato	2	Diagrama de transições (imagem)	01	Diagrama faltando	10.0
			02	Diagrama ilegível	10.0
			03	Diagrama incompleto ou incorreto	5.0
			04	Diagr. desordenado, mal formatado ou fora da especificação	3.0
	2	Tabela de transições (imagem)	05	Tabela faltando	10.0
			06	Tabela ilegível	10.0
			07	Tabelas diferentes para cada teste	10.0
			08	Tabela incompleta ou incorreta	5.0
			09	Tabela desordenada, mal formatada ou fora da especificação	3.0
Implementação	4	Tokens	10	Scanner retorna tokens incorretos ou mais de um por chamada	5.0
			11	Tokens declarados com #define ou const int	2.0
			12	Tokens diferentes de #define, const int ou enum	4.0
	3	Palavras reservadas	13	Conjunto incorreto de palavras reservadas	8.0
			14	Tratamento incorreto de maiúsculas, minúsculas ou acentos	4.0
	4	Transições entre estados	15	Transições não implementadas na forma de uma tabela	10.0
			16	Tabela sub-utilizada (transições guiadas a mão)	8.0
			17	Tabela incongruente com diagrama de estados	5.0
	4	Tabela de símbolos	18	Tabela não implementada	10.0
			19	Registros ou campos a mais, a menos, duplicados ou errados	5.0
			20	Lexemas numéricos armazenados como string	3.0
	3	Comentários	21	Sintaxe implementada incorretamente	8.0
			22	Não ignorados como devido (ex: token retornado, msg de erro)	10.0
Saídas	3	Listagem da entrada e erros	23	Listagem não é gerada	10.0
			24	Listagem com conteúdo incompleto ou incorreto	5.0
			25	Listagem desordenada, mal formatada ou fora da especificação	3.0
	3	Lista de tokens reconhecidos	26	Lista não é gerada	10.0
			27	Lista com conteúdo incompleto ou incorreto	5.0
			28	Lista desordenada, mal formatada ou fora da especificação	3.0
	3	Tabela de símbolos	29	Tabela não é gerada	10.0
			30	Tabela com conteúdo incompleto ou incorreto	5.0
			31	Tabela desordenada, mal formatada ou fora da especificação	3.0

Figura 2 – Exemplo de avaliação: Erros cometidos.

Dupla		Defesa (dia, início, fim)						Ordem	
Aluno 1 e Aluno 2		01/04/2013		14:30	15:20		1		
Descrição do desconto	Local no relatório		Justificativa				Pontos	Quant	
	Pág	Lin	Objetos de avaliação	Especific. Item	Pág	Cód erro			
Todos os testes em Linux:		Trabalho desenvolvido em Dev-C, não executa em Linux. Será aplicado um desconto de 30% na nota.							
Erro de execução; apenas um teste gera saída (truncada e incompleta)									
Teste01.ptg em Windows:									
Ao encontrar um '=' o programa emite mensagem de erro com branco no lugar do '='									
Teste02.ptg em Windows:									
Programa entra em loop infinito; nenhuma saída é gerada									
Teste03.ptg em Windows:									
Erro de execução; saída incompleta									
Outros:									
Programa não aceita o nome do arquivo no momento da invocação									
Lexemas numéricos armazenados como string							20	3,0 1	

Dupla		Defesa (dia, início, fim)				Ordem			
Aluno 3 e Aluno 4		01/04/2013 15:20 16:40				2			
Descrição do desconto	Local no relatório		Justificativa			Pontos	Quant		
	Pág	Lin	Objetos de avaliação	Especific. Item	Cód Pág erro				
Erros de compilação e de execução:									
Código não compila em Linux (#includes mutuamente recursivos)		Trabalho desenvolvido em Dev-C, não executa em Linux. Será aplicado um desconto de 40% na nota							
Código não compila em Dev-C++ ou CodeBlocks no meu computador (mesmo motivo)									
Código modificado dá erro de execução em Linux após gerar apenas o arquivo .tok									
Código modificado dá erro de execução em Windows com Teste-03.ptg									
Código modificado gera em Windows saídas diferentes das impressas no relatório									
Na listagem da entrada com erros léxicos:									
A última linha listada sempre termina com um ' ÿ '						24	5,0	2	
Se há vários erros na mesma linha da entrada, apenas um é detectado									
Erro "Caracter inválido" não é detectado (p.ex. @ # ú)									
Erro "Cadeia aberta mas não fechada" não é tratado corretamente									
Na lista de tokens reconhecidos:									
Palavras reservadas possuem lexema (p.ex. tk_programa)						28	3,0	2	
As linhas são indevidamente separadas por traços									
Palavras reservadas reconhecidas como identificadores (p.ex. real, para, desde, ate)						27	5,0	1	
Palavras reservadas com maiúsculas reconhecidas como identificadores (p.ex. Senao)									
O total de ocorrências de tk_dividido sempre é errado									
Teste01.ptg									
Palavras reservadas contendo letras maiúsculas (p.ex. E, Imprima, IMPRIMA) são reconhecidas como identificadores					3.5	6	10	5,0	1
							14	4,0	1
							27	5,0	1
							30	5,0	1
Outros:									
Tabela de símbolos mal formatada						31	3,0	1	
Lexemas numéricos armazenados como string						20	3,0	1	

Dupla		Defesa (dia, início, fim)					Ordem	
Aluno 5 e Aluno 6			01/04/2013	16:40	17:30	3		
Descrição do desconto	Local no relatório		Justificativa			Pontos	Quant	
	Pág	Lin	Objetos de avaliação	Especific. Item	Pág			Cód erro
Teste01.ptg								
Palavras reservadas contendo letras maiúsculas (p.ex. E, Imprima, IMPRIMA) são reconhecidas como identificadores				3.5	6	10	5,0	1
						14	4,0	1
						27	5,0	1
						30	5,0	1
Identificadores com quantidade incorreta de ocorrências: N1, MAIOR e IMPRIMA						30	5,0	1
Outros:								
Lexemas numéricos armazenados como string						20	3,0	1

Figura 3 – Exemplo de avaliação: Descontos nas notas.

Erros cometidos, por categoria		Autômato									Implementação										Saídas																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																			
		Diagrama de transições (imagem)				Tabela de transições (imagem)					Tokens			Palavras reservadas		Transições entre estados			Tabela de símbolos		Comentários		Listagem da entrada e erros			Lista de tokens reconhecidos			Tabela de símbolos																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																											
Dupla	Nome	10,0	10,0	5,0	3,0	10,0	10,0	10,0	5,0	3,0	9	5,0	10	2,0	11	4,0	12	8,0	13	4,0	14	10,0	15	8,0	16	5,0	17	10,0	18	5,0	19	3,0	20	8,0	21	10,0	22	10,0	23	5,0	24	3,0	25	10,0	26	5,0	27	3,0	28	10,0	29	5,0	30	3,0	31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																	
1	Aluno 1																														1																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																									

Figura 4 – Exemplo de avaliação: Notas.

Dupla Nome		Critérios de avaliação e pesos										31	Nota do trabalho	Desconto posterior	Nota do trabalho	Divisão da nota (após defesa)	Nota individual
		Autôm.		Implementação					Saídas								
		2	2	4	3	4	4	3	3	3	3						
		Diagrama (imagem)	Tabela (imagem)	Tokens	Palavras reservadas	Tabela de transições	Tabela de símbolos	Comentários	Listagem entrada e erros	Lista de tokens	Tabela de símbolos						
1	Aluno 1	10,0	10,0	10,0	10,0	10,0	7,0	10,0	10,0	10,0	10,0	9,6	30%	6,7	50%	6,7	
1	Aluno 2	10,0	10,0	10,0	10,0	10,0	7,0	10,0	10,0	10,0	10,0	9,6	30%	6,7	50%	6,7	
2	Aluno 3	10,0	10,0	5,0	6,0	10,0	7,0	10,0	0,0	0,0	2,0	5,9	40%	3,5	50%	3,5	
2	Aluno 4	10,0	10,0	5,0	6,0	10,0	7,0	10,0	0,0	0,0	2,0	5,9	40%	3,5	50%	3,5	
1	Aluno 5	10,0	10,0	5,0	6,0	10,0	7,0	10,0	10,0	5,0	5,0	7,6		7,6	40%	6,9	
3	Aluno 6	10,0	10,0	5,0	6,0	10,0	7,0	10,0	10,0	5,0	5,0	7,6		7,6	60%	8,4	
Mínimo		10,0	10,0	5,0	6,0	10,0	7,0	10,0	0,0	0,0	2,0	5,9		3,5		3,5	
Média		10,0	10,0	6,7	7,3	10,0	7,0	10,0	6,7	5,0	5,7	7,7		6,0		6,0	
Máximo		10,0	10,0	10,0	10,0	10,0	7,0	10,0	10,0	10,0	10,0	9,6		7,6		8,4	

9. Formulário de pré-avaliação

A última parte do relatório é o formulário de pré-avaliação, preenchido pelos próprios alunos. Objetivos:

1. Padronizar a correção dos trabalhos, garantindo não apenas a objetividade dos critérios de avaliação como também sua aplicação uniforme.
2. Simplificar a correção dos trabalhos através de referências cruzadas conectando *objetos de avaliação* ao código que os implementa.

Note a terminologia empregada:

- **Crítérios de avaliação** – Estão listados na página anterior. Aplicados a um trabalho, resultam na nota.
- **Objetos de avaliação** – Estão listados no formulário de pré-avaliação. Consistem de características da implementação e funcionalidades do código às quais serão aplicados os critérios de avaliação propriamente ditos.

O formulário de pré-avaliação deve ser impresso em branco e preenchido *antes da defesa, a mão, de lápis* (a fim de evitar rasuras desnecessárias) e *com letra legível*. A seguir, instruções para o preenchimento das várias seções do formulário.

9.1. Resumo

O formulário inicia com a seção **Resumo**, onde devem ser fornecidas informações gerais sobre o trabalho como sistema operacional, linguagem e compilador utilizados. As informações solicitadas são auto-explicativas, mas atente aos seguintes detalhes:

- No campo **Auto-avaliação** indique a nota que você imagina que receberá quando o trabalho for avaliado de acordo com os critérios aqui descritos. Note que *esforço* ou *boa vontade* não são critérios de avaliação. A informação fornecida nesse campo não será utilizada no cômputo da nota.
- No campo **Pontos fortes** descreva as maiores qualidades do seu trabalho, como aspectos particularmente bem implementados ou recursos adicionais além dos especificados. Seja breve e específico. Ex.: “tabela de símbolos” não é suficientemente específico; “tabela de símbolos eficiente e compacta” sim.
- No campo **Pontos fracos** descreva as maiores fraquezas do seu trabalho, como recursos ausentes ou aspectos cuja implementação é ineficiente, incompleta ou incorreta. Siga a orientação acima.

9.2. Campos do formulário

O restante do formulário consiste de:

- **Rótulos** – Todos os objetos de avaliação foram rotulados com códigos de três caracteres (uma letra e dois dígitos) entre colchetes e em letras vermelhas, p.ex. [A01]. A letra indica a categoria onde o respectivo objeto de avaliação foi agrupado; no exemplo anterior, A indica a categoria “Nome do arquivo de entrada (programa Portugal)”. O número de dois dígitos é único em cada categoria.
- **Objetos de avaliação** – Já foram explicados.
- **Posições no código fonte** (sob a coluna Evidência) – Locais, no código fonte, onde os respectivos objetos de avaliação estão implementados. Uma posição é indicada pelo número da página no trabalho impresso e pelo número da linha de código. Múltiplas páginas ou linhas podem ser indicadas com 23—28 ou 23, 25, 28.
- **Comentários** – Espaço reservado para notas explicativas que os autores julgarem indispensáveis à correção do trabalho. Use apenas se realmente necessário, com notas muito breves.

A Figura 5 (pág. 24) mostra a estrutura do formulário de avaliação com seus respectivos campos:

- **rótulos** (circulados em laranja)
- **objetos de avaliação** (em verde)
- **posições no código fonte** (em rosa)
- **comentários** (em azul)

Figura 5 – Estrutura do formulário de avaliação.

Objetos de avaliação		Evidência		Comentários <i>(apenas se necessário)</i>
		Página	Linha	
Nome do arquivo de entrada				
[A01]	Fixo no código			
[A02]	Fornecido via teclado (linha de comando)	9	151-157	
[A03]	Fornecido via interface gráfica			
[A04]	Outro:			
Maiúsculas, minúsculas e acentos				
[B01]	() Diferenciadas em palavras reservadas (ate ≠ Ate)			
[B02]	✕ Diferenciadas em identificadores (media ≠ Média)	23	491-521	
[B03]	✕ Acentos rejeitados em palavras reservadas (até)	26	732-741	
[B04]	✕ Acentos rejeitados em identificadores (média)	26	732-741	

9.3. Referências cruzadas

O restante do formulário utiliza o sistema de referências cruzadas já mencionado. Essa é uma maneira simples de evidenciar, no código fonte, até que ponto os critérios de avaliação foram atendidos. A idéia consiste em conectar cada objeto de avaliação com o respectivo trecho de código através de duas anotações:

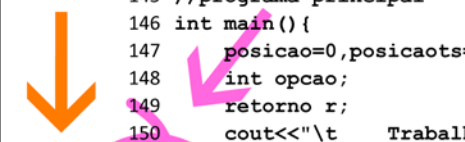
- **No formulário**, ao lado de cada objeto de avaliação: o local, no código fonte, onde está implementado. O local é indicado por dois números: da página no trabalho impresso e da linha de código. Múltiplas páginas ou linhas podem ser indicadas por 23—28 ou 23, 25, 28. Exemplo:

Figura 6 – Exemplo de referência cruzada no formulário de avaliação.

Objetos de avaliação		Evidência		Comentários (apenas se necessário)
		Página	Linha	
Nome do arquivo de entrada				
[A01] ()	Fixo no código			
[A02] (X)	Fornecido via teclado (linha de comando)	9	151-157	
[A03] ()	Fornecido via interface gráfica			
[A04] ()	Outro:			
Maiúsculas, minúsculas e acentos				
[B01] ()	Diferenciadas em palavras reservadas (ate ≠ Ate)			
[B02] (X)	Diferenciadas em identificadores (media ≠ Média)	23	491-521	
[B03] (X)	Acentos rejeitados em palavras reservadas (até)	26	732-741	
[B04] (X)	Acentos rejeitados em identificadores (média)	26	732-741	

- **No código impresso**, à esquerda dos números de linha: os rótulos dos objetos de avaliação que aquele trecho de código implementa. Exemplo:

Figura 7 – Exemplo de referência cruzada no código impresso.



```
145 //programa principal
146 int main(){
147     posicao=0,posicaots=0,linha=1,iniciolinha=0;
148     int opcao;
149     retorno r;
150     cout<<"\t    Trabalho de Compiladores\n\n";
151     //pergunta qual versao vc deseja usar
152     cout<<"Escolha a versao do Analisador Lexico\nVersao 1 ou 2? ";
153     cin>>opcao;
154     //pergunta qual o arquivo, no nosso caso: testel, teste2 e teste3
155     cout<<"\nDigite o nome do arquivo a ser analisado: ";
156     cout<<"\nArquivo: ";
157     cin>>arquivo;
158     ifstream leitura(arquivo.c_str(),ios::in);//ler o arquivo
```

9.4. Seção A – Nome do arquivo de entrada

Indique se o nome do arquivo de entrada (programa em Portugal)

- Está fixo no código, p.ex. `FILE *arquivo = fopen ("r", "testel.ptg")`.
- É fornecido antes da execução, via teclado, na própria linha de comando que invoca o programa, p.ex.
`C:\> Portugal testel.ptg RETURN`
- É fornecido em tempo de execução, via teclado, logo após a invocação do programa, p.ex.
`C:\> Portugal RETURN`
`Digite o nome do arquivo de entrada: testel.ptg RETURN`
- É fornecido em tempo de execução, via mouse ou teclado, através de uma interface gráfica, ou
- É fornecido por algum outro meio; nesse caso, descreva o meio.

Importante: seu programa deve implementar a segunda opção acima (nome do arquivo de entrada fornecido antes da execução, via teclado, na própria linha de comando que invoca o programa). Caso contrário, os testes automáticos falharão e o trabalho não poderá ser avaliado.

9.5. Seção B – Tratamento de maiúsculas, minúsculas e acentos

Indique se:

- Letras maiúsculas e minúsculas são distinguidas em palavras reservadas.
 - Se forem, então `ate` será reconhecido como o token `tkAte`, mas `Ate` e `ATE` serão reconhecidos como identificadores.
 - Se não forem, então `ate`, `Ate` e `ATE` serão reconhecidos como o mesmo token `tkAte`.
- Letras maiúsculas e minúsculas são distinguidas em nomes de identificadores.
 - Se forem, então `media`, `Media` e `MEDIA` serão reconhecidos como identificadores distintos, cada um com seu lexema próprio na tabela de símbolos.
 - Se não forem, então `media`, `Media` e `MEDIA` serão reconhecidos como o mesmo identificador, dando origem a um único lexema na tabela de símbolos.
- Letras acentuadas são rejeitadas em palavras reservadas.
 - Se forem, `ate` será reconhecido como o token `tkAte`, mas `até` e `ATÉ` causarão erros léxicos.
 - Se não forem, então `ate`, `até` e `âté` serão reconhecidos como o mesmo token `tkAte`.
- Letras acentuadas são rejeitadas em nomes de identificadores.
 - Se forem, `media` será reconhecido como identificador, mas `média` e `médiã` causarão erros léxicos.
 - Se não forem, então
 - `media`, `média` e `médiã` serão reconhecidos como identificadores distintos, cada um com seu lexema próprio na tabela de símbolos, ou
 - `media`, `média` e `médiã` serão reconhecidos como o mesmo identificador, dando origem a um único lexema na tabela de símbolos.

9.6. Seção C – Detalhes de implementação

Indique se:

- Comentários de única linha são tratados corretamente.
- Comentários de múltiplas linhas são tratados corretamente.
- O programa retorna `token` para sinalizar um comentário.
- O programa retorna `token` para sinalizar um erro léxico.
- O programa retorna `token` para sinalizar fim de arquivo (p.ex. `tkEOF`).

9.7. Seção D – Estruturas de dados de tamanho arbitrário

Indique se há alguma estrutura de dados (p.ex. vetor, *array*, *string*, tabela) de tamanho arbitrário. Por tamanho arbitrário entende-se, por exemplo:

- Um *string* de tamanho fixo para armazenar linhas do arquivo de entrada.
- Um *array* de tamanho fixo para armazenar as ocorrências de um identificador no arquivo de entrada.
- Uma tabela de símbolos com tamanho máximo fixo.

Em todos esses casos, o tamanho fixo é dado por uma constante, independente de seu valor ou do mecanismo de programação usado (constante numérica, macro, expressão aritmética equivalente a um valor constante, etc). A arbitrariedade consiste em fixar, em tempo de compilação, o tamanho de uma estrutura de dados cuja real necessidade de espaço depende do conteúdo arquivo de entrada, portanto só pode ser conhecida em tempo de execução. Assim, uma tabela de símbolos com capacidade para armazenar 500 milhões de identificadores tem tamanho arbitrário, pois não se sabe, em tempo de compilação, quantos identificadores serão encontrados em determinado arquivo de entrada. Por outro lado, um *array* de tamanho fixo para armazenar palavras reservadas não tem tamanho arbitrário, pois o conjunto de palavras reservadas da linguagem é fixo e independe dos arquivos de entrada, de modo que o espaço necessário para seu armazenamento já é conhecido em tempo de compilação.

Repetindo: o que caracteriza uma estrutura de dados de tamanho arbitrário não é simplesmente o tamanho constante, mas o tamanho constante quando não se sabe, *a priori*, quanto espaço será realmente necessário. A maneira correta de lidar com essa questão é alocar a estrutura de dados dinamicamente, ainda que com um valor inicial arbitrário, e, se necessário, redimensioná-la durante a execução do programa.

9.8. Seção E – Tokens

Indique:

- Como foram declarados os *tokens* fornecidos ao *parser*. Este item **não** se refere à forma como *tokens* são armazenados na tabela de símbolos ou impressos nas saídas. Marque uma das opções abaixo:
 - Através de uma enumeração (p.ex. `enum` em C).
 - Através de `#define` em C.
 - Como variáveis do tipo inteiro (p.ex. `int` em C).
 - Como constantes do tipo inteiro (p.ex. `const int` em C).
 - Como cadeias de caracteres (p.ex. `char[]` em C).
 - De alguma outra maneira; nesse caso, forneça uma descrição.
- O que o *scanner* fornece ao *parser*. Marque todas as opções que se aplicarem:
 - O código numérico do *token* (p.ex. 12).
 - A cadeia do nome do *token* (p.ex. "tkIden").
 - A posição do lexema na tabela de símbolos (p.ex. 25).
 - A cadeia do lexema (p.ex. "media").
 - Outras informações; nesse caso, forneça uma descrição.
- Quando e como o *scanner* fornece *tokens* ao *parser*. Marque todas as opções que se aplicarem:
 - Um *token* por chamada, através de um comando `return`.
 - Um *token* por chamada, através de uma variável global (p.ex. `int` ou `array`).
 - Todos os *tokens* de uma vez, ao final da análise léxica.
 - Outro; forneça uma descrição.

9.9. Seção F – Tabela de transições

Indique (marque todas as opções que se aplicarem):

- Se as transições de estados são guiadas por comandos `if` ou `switch`. Exemplo:

```
if (estado == 13) {  
    ...  
    estado = 5;  
}
```

- Se as transições de estados são guiadas por uma tabela de transições. Exemplo:

```
switch (estado) {  
    ...  
    case 13: { ... }  
    case 14: { ... }  
    ...  
}  
prox_simb = leia_Proximo_Caractere();  
estado = tabela_Transicoes[estado][prox_simb];
```

- Se a tabela de transições foi implementada, descreva a estrutura de dados utilizada.
- Se a tabela é realmente consultada para fazer as transições (como no exemplo acima), ou se foi implementada mas nunca é usada.
- Se a tabela é consultada uma única vez no laço mais interno do analisador léxico.
- Se a tabela é consultada várias vezes, ou seja, se há vários comandos de consulta no código.
- Se a tabela foi impressa e incluída no relatório junto com o diagrama do autômato.
- Se a tabela é congruente com o diagrama do autômato, ou seja, se o diagrama e a tabela contêm exatamente as mesmas informações.
- Se a tabela é única para todos os testes, ou se há uma tabela para cada teste realizado.

9.10. Seção G – Tabela de símbolos

Indique (marque todas as opções que se aplicarem):

- Sobre a tabela de símbolos:
 - Se não foi implementada em nenhuma estrutura de dados.
 - Se foi implementada em uma estrutura de dados compartilhada com outras informações, p.ex. um único *array* para a tabela de símbolos e para a lista de palavras reservadas.
 - Se foi implementada em uma estrutura de dados própria e exclusiva, ou seja, não contém nada além de informações relativas à tabela de símbolos.
 - Se foi implementada, qual a estrutura de dados usada (p.ex. lista encadeada, *array*, tabela *hash*).
 - Se permite duplicação de lexemas, p.ex. o lexema de um identificador aparece em mais de uma posição da tabela de símbolos.
- Sobre as palavras reservadas, operadores, delimitadores, identificadores e constantes inteiras, decimais e *string*:
 - Se os *tokens* não são armazenadas de forma alguma na tabela de símbolos.
 - Se os códigos numéricos dos *tokens* são armazenados – como?
 - Se os nomes dos *tokens* são armazenados – como?
 - Se os *tokens* são armazenados de outra forma – como?
 - Se os lexemas não são armazenadas de forma alguma.
 - Se os lexemas são armazenados – como?
- Se as informações armazenadas na tabela de símbolos são acompanhadas das posições (linha e coluna) onde ocorreram no arquivo de entrada.
- Descreva qualquer outra informação armazenada na tabela de símbolos.