Universidade Católica do Salvador
Cursos de Informática
Bacharelado em Engenharia de Software
Disciplina Compiladores
Professor Osvaldo Requião Melo
Primeiro Semestre de 2024

Documento de especificação do projeto de implementação de Compiladores

2024-1

Versão 1.0
22/05/2024

Salvador (BA) / Maio 2023

Objetivo

Construção de um static Checker sobre a linguagem **ORMPlus2024-1**

O projeto de implementação da disciplina Compiladores será a construção de um Static Checker (executando as tarefas de análise léxica e parte da análise sintática) sobre a linguagem **ORMPlus2024-1**, criada pelo professor da disciplina e padrão para todos os projetos deste semestre. A linguagem **ORMPlus2024-1** é baseada em parte da especificação da linguagem de programação C.

Observação: você não precisa entender a especificação completa da linguagem C, nem da linguagem C para implementar o funcionamento deste trabalho. Você precisa entender a especificação da linguagem **ORMPlus2024-1**, que será detalhada a seguir.

Projeto

A linguagem **ORMPlus2024-1** é definida inicialmente:

a) pela gramática formal especificada abaixo (escrita na notação de produções gramaticais) e composta de duas partes (regras sintáticas e padrões léxicos) e
b) pelas regras de funcionamento descritas neste documento de especificação.

A gramática formal de **ORMPlus2024-1** define:

a) tanto a sintaxe da linguagem e dos comandos da linguagem (que deverão ser usados para a construção do analisador sintático), como
b) os padrões léxicos de formação dos átomos (que deverão ser usados para a construção do analisador léxico).

As regras de funcionamento descritas neste documento vão direcionar como construir o Static Checker:

- como devem funcionar as entradas,
- como são as saídas esperadas,
- as informações base para realizar a análise léxica da linguagem,
- as informações para realizar a análise sintática da linguagem,
- o funcionamento básico dos recursos adicionais fornecidos como recursos ao programador da linguagem,
- como armazenar as informações na tabela de símbolos,
- entre outros.

Cada equipe formada desenvolverá o Static Checker, o qual depois de pronto estará apto a validar **qualquer texto fonte** escrito por usuários nesta linguagem (os textos escritos pelos usuários da linguagem podem estar corretos ou não). A cada execução do programa Static Checker será fornecido um único texto fonte como parâmetro de entrada. Daí em diante o Static Checker verifica o texto de entrada segundo as regras de validação e definição da linguagem **ORMPlus2024-1** e gera os relatórios especificados.

Questões Omissas

As questões que porventura estejam omissas neste documento de especificação que poderão ser decididas:

- nas aulas, ou
- durante a explicação de dúvidas das equipes,
- pelo esclarecimento de dúvidas mandadas no privado por WhatsApp ou e-mail e socializadas com a turma,
- nas mensagens no grupo de WhatsApp ou
- nas respostas de comentários no Google Classroom.

Estas definições passam a valer como especificação do projeto, ou seja, as decisões tomadas nestes momentos passarão a valer também como especificação do projeto e devem ser obrigatoriamente seguidas por todas as equipes.

Plágio

Qualquer implementação de trabalhos de outras fontes (códigos escritos por outros alunos do curso, de outros cursos, de outras fontes da internet, etc...) ou não produzidos pelos alunos que venha a ser identificada na entrega de qualquer parte deste projeto fará com que o projeto seja considerado plágio ou não produzido pela equipe e irá significar o **zeramento completo** da nota do projeto **para todas as etapas do processo** e **para todos os componentes da equipe**.

Equipes

O projeto será realizado por uma equipe formada de alunos de compiladores em um número máximo de quatro componentes. Cada equipe tem um código único que a identifica. A indicação da formação da equipe será feita através do preenchimento de planilha compartilhada no Google Classroom da disciplina. Em nenhuma hipótese será permitida a formação de equipe com número superior ao máximo. Caso haja desistência de alunos e exista um desbalanceamento do tamanho das equipes o professor pode alterar a composição das equipes de forma a reestabelecer o equilíbrio da quantidade de componentes de cada equipe.

A linguagem de programação a ser utilizada pela equipe para a construção do projeto de compiladores é livre, desde que combinada previamente com o professor e será indicada pela equipe na mesma planilha compartilhado do Google Classroom. Adicionalmente ao nome da linguagem deve ser fornecido também qual o ambiente (IDE) e compilador (versão da linguagem) sugeridos pela equipe para o desenvolvimento do projeto. Os prazos para o fornecimento destas informações serão indicados em tarefa no Google Classroom.

O professor vai indicar na própria planilha a indicação de aprovação ou ajuste da linguagem sugerida pela equipe. Após o retorno do professor, a relação de linguagens e ambientes aprovados estará fechada e não mais será aceita a inclusão ou troca para outra linguagem.

Entradas

O Static Checker a ser construído deve aceitar como entrada qualquer texto fonte escrito em ASCII que deverá ser analisado de acordo com as regras da linguagem definidas nesta documentação.

Este texto fonte deverá ter obrigatoriamente a extensão .241.

Não deverá ser solicitada a extensão do texto fonte na chamada de execução do Static Checker. Por exemplo: para que seja analisado o arquivo **MeuTeste.241**, deve ser passado como parâmetro apenas o nome MeuTeste e o programa Static Checker construído deve procurar no disco a existência do texto fonte de nome **MeuTeste.241** para começar os trabalhos.

Caso seja fornecido apenas o nome do texto fonte, este deve ser procurado no diretório corrente onde o Static Checker está sendo executado. Caso seja fornecido o caminho completo mais o nome do texto fonte como parâmetro de entrada, o arquivo deve ser procurado neste caminho indicado na entrada.

Saídas

Quando o Static Checker estiver completo, para cada texto fonte que seja passado como parâmetro para o programa executável do projeto, deverão ser gerados obrigatoriamente dois arquivos de saída em separado na mesma pasta onde o texto fonte parâmetro se encontra:

- um arquivo com o relatório da análise léxica e
- um outro arquivo com o relatório da tabela de símbolos

Os arquivos gerados devem ter o mesmo nome base do texto fonte (modificando apenas a extensão do arquivo). Por exemplo: caso o texto fonte a ser analisado seja o **MeuTeste.241** devem ser gerados na saída os arquivos **MeuTeste.LEX** e **MeuTeste.TAB** (respectivamente os relatórios da análise léxica e da tabela de símbolos correspondentes ao arquivo de entrada).

Estes arquivos (juntamente com o conteúdo que cada um destes relatórios deve conter) serão detalhados abaixo.

Arquivo .LEX (Relatório da análise léxica)
O relatório da análise léxica (contido no arquivo de extensão .LEX) deve mostrar a relação dos símbolos da linguagem que foram encontrados no texto fonte analisado, na ordem em que estes aparecerem e tantas vezes quantas tenham aparecido.

Este relatório deve indicar no cabeçalho: o código identificador da equipe, os nomes, e-mails e telefones de todos os componentes da equipe que participaram da elaboração desta etapa do projeto, além do Título de RELATÓRIO DA ANÁLISE LÉXICA e o nome do texto fonte analisado.

Para cada linha detalhe do relatório de análise léxica, devem ser exibidas no mínimo as informações: o elemento léxico formado (chamado de lexeme), o código do átomo correspondente a este elemento léxico, o índice deste símbolo na tabela de símbolos (quando for um símbolo que seja armazenado nesta estrutura de dados) e a linha onde ele foi encontrado. Caso a equipe julgue interessante podem ser incluídas outras informações no relatório da análise léxica.

Segue um exemplo de um relatório .LEX da análise léxica gerado pela avaliação de um texto fonte exemplo correspondente.

Considere um texto fonte chamado de Teste.241 com o conteúdo abaixo:

```
var1 var2 var3
var1 var1 var1
var2 var2 var1
```

O relatório Teste.LEX gerado após a avaliação do texto fonte Teste.241 acima deverá ser algo conforme o conteúdo exibido abaixo:

```
Código da Equipe: 99
Componentes:
      Fulano da Silva; fulano.silva@ucsal.edu.br; (71)99999-9999
      Beltrano da Silva; beltrano.silva@ucsal.edu.br; (71)99999-9999
      Sicrano da Silva; sicrano.silva@ucsal.edu.br; (71)99999-9999

RELATÓRIO DA ANÁLISE LÉXICA. Texto fonte analisado: Teste.231
----------------------------------------------------------------------
Lexeme: VAR1, Código: 513, ÍndiceTabSimb: 1, Linha: 1.
----------------------------------------------------------------------
Lexeme: VAR2, Código: 513, ÍndiceTabSimb: 2, Linha: 1.
----------------------------------------------------------------------
Lexeme: VAR3, Código: 513, ÍndiceTabSimb: 3, Linha: 1.
----------------------------------------------------------------------
Lexeme: VAR1, Código: 513, ÍndiceTabSimb: 1, Linha: 2.
----------------------------------------------------------------------
Lexeme: VAR1, Código: 513, ÍndiceTabSimb: 1, Linha: 2.
----------------------------------------------------------------------
Lexeme: VAR1, Código: 513, ÍndiceTabSimb: 1, Linha: 2.
----------------------------------------------------------------------
Lexeme: VAR2, Código: 513, ÍndiceTabSimb: 2, Linha: 3.
----------------------------------------------------------------------
Lexeme: VAR2, Código: 513, ÍndiceTabSimb: 2, Linha: 3.
----------------------------------------------------------------------
Lexeme: VAR1, Código: 513, ÍndiceTabSimb: 1, Linha: 3.
----------------------------------------------------------------------
```

Arquivo .TAB (Relatório da tabela de símbolos)
O relatório da tabela de símbolos (contido no arquivo de extensão .TAB) deve mostrar todos os símbolos do tipo identificadores que foram armazenados na tabela de símbolos durante o processo de avaliação do texto fonte.

O relatório deve refletir a última situação da tabela de símbolos após o término do processo de análise realizado.

Para cada símbolo armazenado na tabela de símbolos devem ser relacionados **todos os atributos** dele com todos os valores que foram preenchidos durante o funcionamento do Static Checker. Mais detalhes sobre quais os atributos que devem ser controlados e exibidos na tabela e no relatório .TAB correspondente podem ser encontrados na seção Tabela de Símbolos a seguir nesta mesma especificação.

Segue um exemplo de um relatório .TAB da Tabela de Símbolos gerado pela avaliação de um texto fonte exemplo correspondente.

Considere um texto fonte chamado de Teste.241 com o conteúdo abaixo:

```
var1 var2 var3
var1 var1 var1
var2 var2 var1
```

O relatório Teste.TAB gerado após a avaliação do texto fonte Teste.241 acima deverá ser algo conforme o conteúdo exibido abaixo:

```
Codigo da Equipe: 99
Componentes:
      Fulano da Silva; fulano.silva@ucsal.edu.br; (71)99999-9999
      Beltrano da Silva; beltrano.silva@ucsal.edu.br; (71)99999-9999
      Sicrano da Silva; sicrano.silva@ucsal.edu.br; (71)99999-9999

RELATÓRIO DA TABELA DE SÍMBOLOS. Texto fonte analisado: Teste.231
Entrada: 1, Codigo: 513, Lexeme: VAR1,
QtdCharAntesTrunc: 4, QtdCharDepoisTrunc: 4,
TipoSimb: -, Linhas: {1, 2, 2, 2, 3}.
---------------------------------------------------------------------
Entrada: 1, Codigo: 513, Lexeme: VAR2,
QtdCharAntesTrunc: 4, QtdCharDepoisTrunc: 4,
TipoSimb: -, Linhas: {1, 3, 3}.
---------------------------------------------------------------------
Entrada: 1, Codigo: 513, Lexeme: VAR3,
QtdCharAntesTrunc: 4, QtdCharDepoisTrunc: 4,
TipoSimb: -, Linhas: {1}.
```

Entregas do projeto

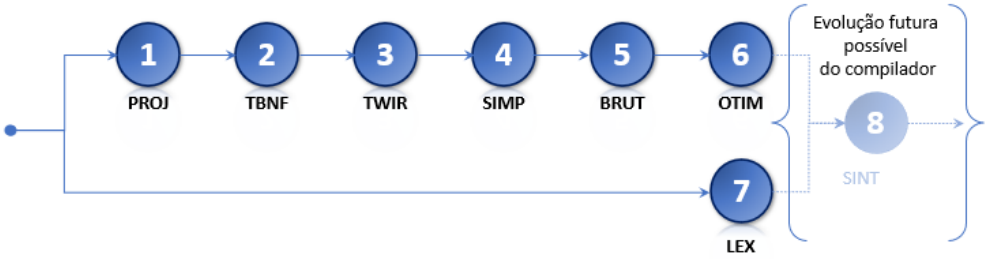
A construção do projeto de compiladores será composta de etapas de entrega, discriminadas abaixo. Cada etapa tem um código próprio de três ou quatro letras que deve ser usado na nomeação do arquivo de entrega correspondente.

- **Etapa 1 (PROJ):** Projeto conceitual da implementação do compilador;
- **Etapa 2 (TBNF):** Transformação da gramática do projeto para a notação de BNF;
- **Etapa 3 (TWIR):** Transformação da gramática do projeto para a notação de Wirth (com eliminação de recursão);
- **Etapa 4 (SIMP):** Simplificação e Enxugamento da gramática da linguagem (primeira etapa de transformação no aut bruto);
- **Etapa 5 (BRUT):** Documentação do processo de obtenção do autômato bruto equivalente à linguagem do projeto;
- **Etapa 6 (OTIM):** documentação do processo de obtenção do autômato ótimo equivalente à linguagem do projeto;
- **Etapa 7 (LEX):** entrega da implementação completa do Static Checker funcionando com programa principal, a análise léxica e tabela de símbolos (fontes e executável).

As etapas de acima terão datas de entrega escalonadas e divulgadas através do Google Classroom da disciplina. Mas a ordem de entrega não limita a ordem de elaboração das etapas que podem e devem ser desenvolvidas em frentes paralelas pelas equipes, não sendo necessário que uma etapa tenha terminado para que seja

iniciada a próxima.

Por exemplo, a etapa LEX de codificação do Static Checker já pode ser iniciada de imediato, embora seja a última entrega do trabalho.



As equipes podem se dividir na execução direta das tarefas, mas todos os componentes devem conhecer tudo o que está sendo feito e podem ser cobrados das explicações de qualquer etapa do desenvolvimento do projeto pelas suas equipes.

Cada uma das entregas do projeto de compiladores acontece com o envio de um único arquivo compactado em formato ZIP livre de contaminação anexado à entrega da tarefa específica determinada para esta etapa no Google Classroom da disciplina (nunca para o grupo de WhatsApp) ou de forma pública para as outras equipes. Neste arquivo compactado devem estar todos os itens solicitados para esta etapa de entrega do projeto. O nome do arquivo compactado deve conter o código identificador da equipe seguido do código da entrega (Exemplo: E01WIRT.ZIP será a entrega da etapa WIRT da equipe E01 do projeto de compiladores).

Na entrega incluir arquivo texto chamado ENTREGA contendo obrigatoriamente as informações: código da equipe, nome de todos os componentes da equipe que participaram daquela etapa, acompanhados de telefones de contato e endereços de e-mail e a relação dos itens que compõem aquela entrega (o formato do arquivo ENTREGA será determinado no Google Classromm em cada atividade).

Em cada etapa a entrega deve ser feita **apenas** do arquivo compactado anexado e do arquivo ENTREGA preenchido.

Roteiro sugerido

Segue roteiro sugerido do que deve ser feito pelas equipes para o projeto da disciplina. As atividades sugeridas neste roteiro estão na ordem de execução. Algumas atividades vão sendo complementadas com os assuntos vistos nas aulas:

1. Ler com cuidado o documento de especificação do projeto (este documento) e listar todas as dúvidas para esclarecimentos que serão feitos em sala de aula nas próximas aulas ou pelo Google Classroom;
2. Tentar entender a lógica da linguagem **ORMPlus2024-1** (tentar compreender o que pode ou não pode ser gerado pela linguagem, tentar escrever alguns textos fontes na linguagem e comparar com a especificação);
3. Projetar a implementação necessária para fazer o Static Checker funcionar e iniciar a codificação da entrega;
4. Planejar as rotinas para funcionamento isolado do analisador léxico (pensar

nas rotinas, variáveis, lógica, estruturas, interface);

5. Planejar as rotinas para funcionamento isolado da tabela de símbolos (pensar nas rotinas, variáveis, lógica, estruturas, interface);
6. Planejar as rotinas gerais para que o analisador léxico e tabela de símbolos funcione (pensar nas rotinas, variáveis, lógica, estruturas, interface);
7. Documentar o projeto da implementação indicando qual a estrutura dos principais módulos, rotinas e informações que será a primeira entrega do trabalho;
8. Iniciar a implementação do código das rotinas gerais de inicialização e abertura do texto fonte (no programa principal que vai se tornar o sintático no futuro). o código da análise léxico, o código da tabela de símbolos, e as rotinas gerais de controle de escopo (no programa principal)
9. Transformar a gramática fornecida para a notação BNF;
10. Transformar a gramática BNF produzida no passo 8 acima para a notação mais adequada transformando a mesma para a notação de Wirth com eliminação de recursão (de BNF para Wirth);
11. Simplificar a linguagem transformada em Wirth reduzindo a quantidade de símbolos não terminais, deixando apenas os não terminais auto recursivos centrais essenciais;
12. Transformar a gramática fornecida em um conjunto de autômatos brutos equivalentes conforme processo a ser visto em sala de aula;
13. Gerar arquivo com a entrega desta etapa contendo: a situação final da gramática depois de toda a simplificação realizada, cada um dos autômatos brutos transformados na notação de tabelas de transição (identificação dos estados, das transições, inicia e final e a tabela com as informações preenchidas dos autômatos brutos);
14. Transformar o conjunto de autômatos brutos em um outro conjunto de autômatos ótimos equivalentes (para a implementação de parte do analisador sintático da linguagem), conforme processo a ser visto em sala de aula;
15. Gerar arquivo com a entrega desta etapa contendo: cada um dos autômatos ótimos transformados na notação de tabela de transição, cada um dos autômatos ótimos transformados na notação de diagrama de estados e o planejamento atualizado das rotinas do compilador para a realização da análise léxica, tabela de símbolos e gerais necessárias para estas etapas (incluindo inicializações e validação de escopo) contendo as principais rotinas a serem implementadas, parâmetros, variáveis, estruturas, interface e funcionalidades que serão realizadas.
16. Complementar a implementação do código das rotinas gerais de inicialização e abertura do texto fonte (no programa principal que vai se tornar o sintático no futuro). o código da análise léxico, o código da tabela de símbolos, e as rotinas gerais de controle de escopo (no programa principal)
17. Testar o código implementado
18. Gerar arquivo da quinta entrega com: **o código fonte** do compilador com parte da análise léxica, as rotinas de tabela de símbolos e o programa principal fazendo controle de escopo e as inicializações necessárias e **o executável** correspondente a esta parte funcionando e produzindo como saídas o relatório da análise léxica e o relatório da tabela de símbolos (com o conteúdo que pode ser identificado até este ponto) e pequena documentação do projeto.

Analisador léxico

No analisador léxico devem ser lidos todos os caracteres do texto fonte, um a um, e baseado nesta leitura devem ser montados os átomos que se encontram neste texto fonte de acordo com os padrões de formação de cada um deles e com a situação que ocorre em cada texto fonte. Cada chamada ao analisador léxico tem por função formar apenas um átomo do texto fonte. A cada chamada, o analisador léxico é informado da posição corrente no texto fonte e deve ser capaz de formar o próximo átomo que existe após esta posição retornando esta informação para o programa chamador do analisador léxico. Esta leitura poderá ser implementada usando a técnica de bufferização de entrada com buffer de duas metades, embora não seja exigido.

A relação dos átomos possíveis para a linguagem **ORMPlus2024-1**será fornecida logo abaixo no **apêndice A** desta especificação.

A implementação do seu projeto não deve considerar diferenças entre letras maiúsculas e minúsculas. O texto fonte fornecido como parâmetros pode conter letras maiúsculas e minúsculas, mas devem ser consideradas como se fossem todas maiúsculas. Os identificadores com diferença de caixa não serão considerados como símbolos distintos (caixa alta ou caixa baixa é o mesmo que maiúsculo ou minúsculo).

O texto fonte pode ser formado de qualquer sequência de caracteres. Alguns caracteres são caracteres válidos para a linguagem, outros caracteres são inválidos para a linguagem.

Os caracteres válidos são aqueles usados em alguma construção da linguagem (no padrão de formação de algum átomo da linguagem ou em construções auxiliares como comentários e arrumação do texto). Os espaços em branco, os comentários, marcas de tabulação (ASCII 09), caractere de fim de linha (ASCII 10), o de quebra de linha (ASCII 13), além de todos os caracteres usados na montagem de um átomo válido da linguagem, são considerados caracteres válidos da linguagem.

Os caracteres inválidos devem ser filtrados no processo de formação dos átomos. Este filtro é chamado de filtro de primeiro nível e não deve significar erro na execução do analisador léxico. Neste caso os caracteres inválidos são simplesmente desconsiderados do texto fonte sem funcionar como um delimitador permitindo que a formação do átomo continue.

Os espaços em branco existentes nos textos fontes normalmente funcionam como delimitadores na formação dos átomos para a maioria das linguagens. No caso da linguagem **ORMPlus2024-1**, eles se encaixam na regra geral, ou seja, todos os caracteres válidos que não fizerem parte do padrão de formação do átomo que estiver sendo formado correntemente devem ser considerados como delimitadores para este processo de montagem de átomo. Nos casos onde o espaço em branco não faça parte do padrão de formação do átomo sendo formado ele vai funcionar como um delimitador. Nos casos onde o espaço em branco faça parte do padrão léxico de formação do átomo eles poderão ser considerados como parte do elemento léxico que está sendo formado. Os espaços em branco adicionais, repetidos, devem ser filtrados do texto fonte.

Os comentários em **ORMPlus2024-1** são fornecidos como recurso adicional da linguagem e não estão especificados na definição formal dada pela gramática. Os comentários existentes nos textos fontes normalmente funcionam como delimitadores na formação dos átomos e neste caso deverão ser filtrados do texto para efeito das etapas posteriores do Static Checker. Os comentários podem acontecer de duas formas nos textos fontes: ou como comentários de bloco ou como comentários de linha. No caso dos comentários de bloco eles devem ser iniciados com a cadeia " /* " (abre comentário) e finalizados com a cadeia " */ " (fecha comentário). Neste caso, se não existir a segunda cadeia " */ " fechando o comentário, todo o restante do programa fonte até o final de arquivo deverá ser considerado comentário. Os comentários de linha podem ser iniciados com a cadeia " // " (inicia comentário de linha), devendo neste caso valer até o final da linha corrente. Se não existir o final de linha fechando o comentário, todo o restante do programa fonte até o final de arquivo deverá ser considerado comentário.

Na leitura do primeiro caractere válido, logo após a chamada do analisador léxico, é seguido um dos padrões léxicos existentes para a linguagem. Este padrão vai ser seguido até que se encontre algum caractere válido para a linguagem que não seja válido para o padrão de formação do átomo que está sendo montado. Ou seja, tudo o que não puder fazer parte deste átomo será considerado como delimitador, usando o critério do máximo comprimento possível, ou seja, enquanto o caractere lido puder fazer parte do átomo ele será considerado como parte do átomo que está sendo montado.

Para a linguagem **ORMPlus2024-1**, os átomos possuem um limite máximo de 30 caracteres válidos de tamanho. Todas as seqüências de caracteres válidos para a linguagem que respeitem a um determinado padrão de formação de átomo devem ser consideradas apenas até o limite máximo de 30 caracteres e após este limite devem ser desconsiderados para o átomo que está sendo formado no momento. Por exemplo, um nome de variável que tenha 100 caracteres válidos de tamanho será reconhecido pelo analisador léxico como sendo um átomo do tipo nome de variável com as 30 primeiras posições e o restante dos caracteres (os outros 65 caracteres) que poderiam fazer parte deste átomo nome de variável, pelo padrão léxico de formação, deverão ser desconsiderados do texto fonte. Apenas os caracteres não filtrados são contados para o limite de 30 caracteres de um átomo. A leitura deve continuar mesmo depois dos 30 primeiros caracteres até encontrar o ponto onde vai existir algum caractere delimitador para aquele átomo.

No limite de 30 caracteres para a formação do átomo, o analisador léxico deve estar atento para formar apenas átomos válidos para o padrão definido para aquele átomo. Algumas situações especiais devem ser tratadas como, por exemplo, forçar um final de cadeia na posição número 30, ou garantir que os números após truncar os elementos após a posição 30 irão formar construções coerentes e válidas para o padrão que foi usado. Os abre e fecha aspas são considerados como caractere válido na contabilização do tamanho do átomo. Para os comentários não valem as regras de tamanho máximo de um átomo, já que neste caso não se trata de átomo da linguagem.

A cada chamada da rotina do analisador léxico serão utilizadas três informações: 1) a posição corrente do texto fonte que deve ser analisada no momento, 2) o código do átomo formado (parâmetro de retorno) e 3) o índice da tabela de símbolos onde o átomo foi gravado (apenas para os identificadores, também parâmetro de retorno). O átomo formado pode ser verificado, após a sua montagem, com a tabela de palavras e símbolos reservados que irá conter todas as palavras definidas da linguagem e com a tabela de símbolos, contendo todos os identificadores já reconhecidos. A análise léxica trabalhará sobre o arquivo texto com extensão .241 e irá gerar o relatório .LEX.

Analisador sintático

O analisador sintático da linguagem funciona como programa principal do compilador usando o esquema de implementação do *sintax driven* conforme visto no assunto de conceitos básicos.

Em função do tempo para a realização do projeto não será exigida a realização da análise sintática propriamente dita, mas parte do analisador sintático precisa ser implementada, principalmente as atividades de controlador, inicializador e emissão de relatórios. Além disto as bases para a verificação da ordem dos átomos (pelos autômatos ótimos) e o controle de escopo para o correto funcionamento do analisador léxico também precisa ser implementado.

Como atividades do analisador léxico temos:

- Solicitação ao usuário do arquivo fonte a ser compilado
- Abertura do arquivo fonte
- Inicialização das estruturas de dados da tabela de símbolos e da tabela de palavras e símbolos reservados
- Chamadas ao analisador léxico
- Controle de escopo para a formação dos átomos
- Emissão dos relatórios .LEX e .TAB

Tabela de Símbolos

Apenas os átomos identificadores serão armazenados na tabela de símbolos. As palavras e símbolos reservados da linguagem **ORMPlus2024-1** deverão constar numa tabela especial separada da tabela de símbolos, que será chamada de tabela de palavras e símbolos reservados e que deverá estar previamente carregada antes do início da primeira análise. A tabela de palavras e símbolos reservados será fixa para todos os programas analisados. A tabela de símbolos irá variar a depender dos átomos existentes no texto fonte que estiver sendo analisado. A tabela de símbolos do projeto irá conter os seguintes atributos: número da entrada da tabela de símbolos, código do átomo, lexeme, quantidade de caracteres antes da truncagem, quantidade de caracteres depois da truncagem, tipo do símbolo e as cinco primeiras linhas onde o símbolo aparece.

Os números das entradas indicarão os índices de armazenamento daqueles símbolos na tabela e são usados em todo o processo de análise do texto fonte. Cada símbolo (lexeme com o mesmo significado) na tabela de símbolos deverá ter um endereço único, ou seja, não existirão duas entradas na tabela de símbolos para o mesmo símbolo. Esta informação não deve ser modificada durante o processo de

análise.

Os códigos dos átomos, correspondentes aos tipos dos símbolos encontrados no texto fonte, seguirão a relação de códigos de átomos fornecida na especificação da linguagem e constante no item 10 desta documentação (apenas para os átomos identificadores, que são os símbolos guardados na tabela de símbolos). Esta informação não deve ser modificada durante a análise.

Os lexemes devem ser guardados com apenas os 30 primeiros caracteres válidos da linguagem que aparecem no texto fonte. Esta informação não deve ser modificada durante a análise.

Os tipos dos símbolos deverão ser preenchidos apenas para os identificadores em que fizer sentido (Identifier, Function, Integer-Number, Float-Number, Constant-String, Character). Os tipos podem um dos seguintes: PFO (ponto flutuante) INT (inteiro), STR (string), CHC (character), BOO (booleano), VOI (void), APF,(array de ponto flutuante) AIN (array de inteiro), AST (array de string), ACH (array de character), ABO (array de booleano). Esta informação não deve ser modificada durante a análise.

As quantidades de caracteres do lexeme levando em conta apenas os 30 primeiros caracteres válidos irão armazenar para cada símbolo a quantidade de caracteres válidos antes do processo de truncagem acontecer. Não devem ser levados em consideração os caracteres filtrados na montagem do átomo (caracteres inválidos da linguagem). Para as cadeias considerar as aspas simples ou duplas como parte do tamanho do átomo. Esta informação pode ser modificada durante a análise. As quantidades de caracteres do lexeme sem levar em conta apenas os 30 primeiros caracteres válidos irão armazenar para cada símbolo a quantidade de caracteres independente do processo de truncagem que tenha acontecido. Não devem ser levados em consideração os caracteres filtrados na montagem do átomo (caracteres inválidos da linguagem). Para as cadeias considerar as aspas simples ou duplas como parte do tamanho do átomo. Esta informação pode ser modificada durante a análise.

Os números das linhas onde o símbolo aparece pelas primeiras cinco vezes serão guardados com base no controle de linhas a ser efetuado sobre o texto fonte. Para este controle de linhas não devem ser descartadas as linhas de comentários existentes no texto. Deve ser considerada a linha onde o símbolo começa para os casos em que um símbolo possa ser definido em mais de uma linha. Esta informação pode ser modificada durante a análise.

Autômatos

As equipes vão transformar a linguagem fornecida nesta especificação em seus autômatos ótimos equivalentes (IMPORTANTE: usando apenas o nível sintático de definição sintática da linguagem, ou seja, até o item 11 desta especificação). Estes autômatos ótimos servirão para a implementação do analisador sintático. A linguagem deverá ter sido transformada em N regras de gramática que produzirá N autômatos equivalentes. Devemos ter ao final de todo o processo N autômatos distintos que serão simplificados através das técnicas de obtenção de autômatos

ótimos. O conjunto destes N autômatos é o reconhecedor da linguagem e a sua implementação seria o analisador sintático da linguagem **ORMPlus2024-1**.

No processo de simplificação considere todos os átomos da linguagem (veja relação de átomos logo abaixo) como sendo símbolos terminais do ponto de vista do analisador sintático, mesmo que estes sejam símbolos não terminais na especificação da linguagem. Isto acontece com os símbolos identificadores, já que eles são detalhados ainda na linguagem para determinação dos padrões léxicos de formação dos átomos.

Nas regras de gramática resultantes e nos autômatos brutos e ótimos encontrados existirão referências aos outros autômatos e regras da linguagem (um dos outros N), que deverão ser encarados neste ponto como um outro átomo qualquer da linguagem. Os códigos dos autômatos serão iniciados em 90, até o código 90+N (dependendo da quantidade de autômatos resultantes do processo de obtenção do autômato ótimo).

Apêndices

Complementam a especificação do projeto de especificação do compilador três apêndices que são fornecidos a seguir

A. Definição dos átomos da linguagem
B. Definição da Gramática da linguagem (sintaxe dos comandos)
C. Definição da Gramática da linguagem (padrões léxicos de formação)

APÊNDICE A) Definição dos Átomos da Linguagem **ORMPlus2024-1**

Átomo	Cód	Átomo	Cód	Átomo	Cód	Átomo	Cód	Átomo	Cód	Átomo	Cód
cadeia	A01	inteiro	A14	%	B01	-	B13	consCadeia	C01	subMáquina1	D01
caracter	A02	logico	A15	(	B02	*	B14	consCaracter	C02	subMáquina2	D02
declaracoes	A03	pausa	A16	)	B03	/	B15	consInteiro	C03	subMáquina3	D03
enquanto	A04	programa	A17	,	B04	+	B16	consReal	C04	...	
false	A05	real	A18	:	B05	!=	B17	nomFuncao	C05	subMáquinan	D0n
fimDeclaracoes	A06	retorna	A19	:=	B06	#	B17	nomPrograma	C06		
fimEnquanto	A07	se	A20	;	B07	<	B18	variavel	C07		
fimFunc	A08	senao	A21	?	B08	<=	B19				
fimFuncoes	A09	tipoFunc	A22	[	B09	==	B20				
fimPrograma	A10	tipoParam	A23	]	B10	>	B21				
fimSe	A11	tipoVar	A24	{	B11	>=	B22				
funcoes	A12	true	A25	}	B12						
imprime	A13	vazio	A26								

APÊNDICE B)
Definição da Gramática da Linguagem ORMPlus2024-1 – Sintaxe dos comandos

Todos os símbolos marcados em azul no texto abaixo são símbolos do alfabeto da linguagem. Todos os não terminais marcados em vermelho são átomos da linguagem e do ponto de vista **do sintático** devem ser considerados como se fossem símbolos terminais. Eles estão detalhados na segunda parte da especificação da linguagem no **APÊNDICE C**.

Gramática para a sintaxe dos comandos:

G = (V, Σ, P, FileProgram)

V = {
FileProgram, DeclarationList, DeclarationVar, TypeSpecification, VariableListVetor, VariableList, FunctionList, DeclarationFunc, Parameters, ParamTypeList, ParamType, ParamList, Param, Command, BlockCommand, ListCommand, IfCommand, WhileCommand, ReturnCommand, PauseCommand, PrintCommand, AtribCommand, Variable, LogicalExp, AritmEx, AddExp, RelatOperator , AddOper, Term, MultOper, UnaryExp, Factor, Constant, Numbers, Strings, **programa, nomPrograma, declaracoes, fimDeclaracoes, funcoes, fimFuncoes, fimPrograma, ; , tipoVar, : , [,] , : , , , variavel, consInteiro, real, inteiro, cadeia, logico, caracter, vazio, tipoFunc, nomFuncao, (,) , fimFunc, ?, tipoParam, { , }, se, fimSe, senao, enquanto, fimEnquanto, retorna, pausa, imprime , :=, <= , <, >, >=, == , !=, #, -, +, *, /, %, true, false, consInteiro, consReal, consCadeia, consCaracter**
}

Σ = {
programa, nomPrograma, declaracoes, fimDeclaracoes, funcoes, fimFuncoes, fimPrograma, ; , tipoVar, : , [,] , : , , , variavel, consInteiro, real, inteiro, cadeia, logico, caracter, vazio, tipoFunc, nomFuncao, (,) , fimFunc, ?, tipoParam, { , }, se, fimSe, senao, enquanto, fimEnquanto, retorna, pausa, imprime , :=, <= , <, >, >=, == , !=, #, -, +, *, /, %, true, false, consInteiro, consReal, consCadeia, consCaracter
}

P = {
FileProgram -> **programa** nomPrograma **declaracoes** DeclarationList **fimDeclaracoes funcoes** FunctionList **fimFuncoes fimPrograma** ,
FileProgram -> **programa** nomPrograma **declaracoes** DeclarationList **fimDeclaracoes fimPrograma** ,
DeclarationList -> DeclarationList **;** DeclarationVar ,
DeclarationList -> DeclarationVar ,
DeclarationVar -> **tipoVar** TypeSpecification**:** VariableList ,
DeclarationVar -> **tipoVar** TypeSpecification **[ ]:** VariableListVetor ,
VariableList -> VariableList **,** variavel ,
VariableList -> variavel ,
VariableListVetor -> VariableListVetor **,** variavel **[** consInteiro **]** ,
VariableListVetor -> variavel **[** consInteiro **]** ,
TypeSpecification -> **real** ,
TypeSpecification -> **inteiro** ,
TypeSpecification -> **cadeia** ,
TypeSpecification -> **logico** ,

TypeSpecification -> **caracter** ,
TypeSpecification -> **vazio** ,
FunctionList -> FunctionList **;** DeclarationFunc ,
FunctionList -> DeclarationFunc ,
DeclarationFunc -> **tipoFunc** TypeSpecification**:** nomFuncao **(** Parameters **)** Command **fimFunc** ,
Parameters -> ParamTypeList ,
Parameters -> **?** ,
ParamTypeList -> ParamTypeList **;** ParamType ,
ParamTypeList -> ParamType ,
ParamType -> **tipoParam** TypeSpecification**:** ParamList ,
ParamList -> ParamList **,** Param ,
ParamList -> Param ,
Param -> variavel ,
Param -> variavel **[** consInteiro **]** ,
Command -> BlockCommand ,
Command -> AtribCommand ,
Command -> PrintCommand ,
Command -> IfCommand ,
Command -> WhileCommand ,
Command -> ReturnCommand ,
Command -> PauseCommand ,
BlockCommand -> **{** ListCommand **}** ,
BlockCommand -> Command ,
ListCommand -> ListCommand **;** Command ,
ListCommand -> Command ,
IfCommand -> **se (** LogicalExp **)** Command **fimSe** ,
IfCommand -> **se (** LogicalExp **)** Command **senao** Command **fimSe** ,
WhileCommand -> **enquanto (** LogicalExp **)** Command **fimEnquanto** ,
ReturnCommand -> **retorna** ,
ReturnCommand -> **retorna** AritmExp ,
PauseCommand -> **pausa** ,
PrintCommand -> **imprime** AritmExp ,
AtribCommand -> Variable **:=** AritmExp ,
AtribCommand -> Variable **:=** LogicalExp ,
Variable -> variavel ,
Variable -> variavel **[** consInteiro **]** ,
LogicalExp -> LogicalExp RelatOperator AritmExp ,
LogicalExp -> **(** LogicalExp **)** ,
LogicalExp -> AritmExp ,
AritmExp -> AddExp ,
RelatOperator -> **<=** ,
RelatOperator -> **<** ,
RelatOperator -> **>** ,
RelatOperator -> **>=** ,
RelatOperator -> **==** ,
RelatOperator -> **!=** ,
RelatOperator -> **#** ,
AddExp -> AddExp AddOper Term ,
AddExp -> Term ,

AddOper -> - ,
AddOper -> + ,
Term -> Term MultOper UnaryExp ,
Term -> UnaryExp ,
MultOper -> * ,
MultOper -> / ,
MultOper -> % ,
UnaryExp -> - UnaryExp ,
UnaryExp -> Factor ,
Factor -> (AritmExp) ,
Factor -> Variable ,
Factor -> Constant ,
Constant -> Numbers ,
Constant -> Strings ,
Constant -> true ,
Constant -> false ,
Numbers -> consInteiro ,
Numbers -> consReal ,
Strings -> consCadeia ,
Strings -> consCaracter
}

APÊNDICE C) Definição da Gramática da Linguagem **ORMPlus2024-1** – Padrões Léxicos de formação

<nomPrograma> ::= <letra> | <nomPrograma> <letra> | <nomPrograma> <digito>
<variavel> ::= <letra> | _ | <variavel> <letra> | <variavel> <digito> | <variavel> _
<nomFuncao> ::= <letra> | <nomFuncao> <letra> | <nomFuncao> <digito>
<consInteiro> ::= <digitos-decimal>
<digitos-decimal> ::= <digito> | <digitos-decimal><digito>
<consReal> ::= <digitos-decimal> . <digitos-decimal> | <digitos-decimal> . <digitos-decimal> <parte-exponencial>
<parte-exponencial> ::= e <digitos-decimal> | e - <digitos-decimal> | e + <digitos-decimal>
<consCadeia> ::= "'" <miolo-cadeia> "'" //inicia e termina com aspas duplas
<miolo-cadeia> ::= (<letra> | <branco> | <digito> | $ | _ | .) <miolo-cadeia> | <letra> | <branco> | <digito> | $ | _ | .
<consCaracter> ::= "'" <letra> "'" //inicia e termina com aspas simples
<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<branco> ::= *"caracter de espaço em branco".*