Universidade Católica do Salvador
Curso de Bacharelado em Informática
Disciplina Compiladores
Professor: Osvaldo Requião Melo

Documentação do projeto Comp2018-1

Especificação para construção do projeto de compiladores: um Static Checker sobre a linguagem Comp2018-1

Documentação fornecida às equipes em 15/05/2018. (Versão 1.0)

01. Sobre a especificação do projeto de compiladores

O projeto de implementação da disciplina Compiladores será a construção de um **Static Checker** (executando as tarefas de análise léxica e análise sintática) sobre a linguagem Comp2018-1, construída pelo professor, baseada na especificação da linguagem de programação C. Observação: você não precisa entender a especificação completa da linguagem C para a implementação do trabalho de compiladores, você deve entender a especificação da linguagem Comp2018-1 que será detalhada a seguir.

A **linguagem Comp2018-1** é definida inicialmente:

a) pela **gramática formal** especificada abaixo (escrita em PG) 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 Comp2018-1 define: tanto a sintaxe da linguagem e dos comandos da linguagem (que deverão ser usados para a construção do analisador sintático), como 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 e saídas, 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 (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 Comp2018-1. Todas as equipes estarão implementando programas para tratar a mesma linguagem de entrada.

As questões que porventura estejam omissas neste documento de especificação serão decididas em sala de aula, durante a explicação de dúvidas das equipes, ou pelo esclarecimento de dúvidas mandadas por e-mail, mensagens de whatsapp ou respostas de comentários no google classroom. As decisões tomadas nestes momentos passarão a valer também como especificação do projeto e devem ser obrigatoriamente seguidas por todas as equipes.

Qualquer implementação de trabalhos de outras fontes que venha a ser identificada na entrega do qualquer parte do projeto fará com que o projeto seja considerado plágio e irá significar o **zeramento completo** da nota do projeto para todas as etapas do processo e para todos os componentes da equipe.

02. A formação das equipes

O projeto será realizado por uma equipe formada de alunos de compiladores em um número máximo de quatro componentes. A linguagem de programação a ser utilizada pela equipe para a construção do projeto é livre, desde que combinada previamente com o professor. A formação da equipe, assim como a escolha da linguagem e ambiente de desenvolvimento escolhido devem ser formalizadas por comentário postado no Google Classroom enviado por um dos componentes da equipe em tópico específico do projeto. Neste comentário, devem ser indicadas obrigatoriamente as informações a seguir: os nomes completos de todos os componentes da equipe, qual a linguagem sugerida pela equipe e qual o ambiente e compilador sugeridos pela equipe para o desenvolvimento do projeto. Este envio deve acontecer obrigatoriamente antes do primeiro prazo de entrega do trabalho.

Após o envio das informações das equipes, será retornado pelo professor em e-mail para a lista de discussão os códigos de cada uma das equipes formadas e a confirmação das linguagens e compiladores aprovados. Estes códigos deverão ser usados em todas as entregas que serão efetuadas pelas equipes daí em diante.

Após o e-mail de retorno do professor, a relação de linguagens e ambientes aprovados estará fechada e não mais será aceita a inclusão ou troca para outra linguagem.

03. Entradas e saídas do Static Checker

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 **.181**. 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.181**, deve ser passado como parâmetro apenas o nome **MeuTeste** e o programa Static Checker deve procurar no disco a existência do texto fonte de nome **MeuTeste.181** 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.

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 três arquivos de saída em separado na mesma pasta onde o texto fonte parâmetro se encontra: um arquivo com o relatório da análise léxica, outro com o relatório da tabela de símbolos e um último com o relatório da análise sintática. 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**.181** 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 devem conter) serão detalhados abaixo.

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 número da equipe, os nomes, e-mails e telefones de todos os componentes da equipe que participaram da elaboração desta etapa do projeto. 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 e o índice deste símbolo na tabela de símbolos (quando for um símbolo que seja armazenado nesta estrutura de dados). Caso a equipe julgue interessante podem ser incluídas outras informações no relatório da análise léxica.

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 análises 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.

04. O roteiro para a construção do projeto de compiladores e entregas correspondentes

A construção do projeto de compiladores será composta de quatro etapas de entrega, sendo:

- Etapa 1 (LIN): transformação e simplificação da gramática da linguagem até 20/05
- Etapa 2 (BRU): documentação do processo de obtenção do autômato bruto equivalente à linguagem do projeto até dia 27/05
- Etapa 3 (OTI): documentação do processo de obtenção do autômato ótimo equivalente à linguagem do projeto até 03/06
- Etapa 4 (LEX): entrega da implementação completa do Static Checker funcionando com programa principal, a análise léxica e tabela de símbolos (fontes e executável) até 10/06.

A entrega do projeto de compiladores acontece com o envio de arquivo compactado em formato ZIP livre de contaminação por vírus através de e-mail diretamente ao professor (nunca para a lista de discussão da disciplina) usando os endereços fornecidos em sala de aula. No título do e-mail deve ser colocado o código da equipe seguido do código da entrega seguido de texto dizendo “entrega da etapa LEX da equipe 99 (Exemplo: N01LEX – Entrega da etapa LEX da equipe N01 do projeto de compiladores”). No corpo do e-mail de entrega devem constar obrigatoriamente as informações: o código da equipe, o nome de todos os componentes da equipe que participaram daquela etapa, acompanhados de telefones de contato e endereços de e-mail. O arquivo a ser entregue deve estar compactado no formato zip tendo como nome o código da equipe concatenado com o código da etapa ou parte correspondente (exemplo: N01LEX.ZIP). O código da equipe é o fornecido pelo professor. O código da etapa é o definido na lista acima. Neste arquivo devem estar todos os itens solicitados para esta etapa de entrega do projeto.

O roteiro do que deve ser feito pelas equipes em cada uma das etapas é o seguinte (as sugestões de roteiro estão na ordem) :

- 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 ou pela lista, principalmente os itens relativos ao projeto como um todo, autômato ótimo, analisador léxico e tabela de símbolos.
- Tentar entender a lógica da linguagem Comp2018-1 (tentar compreender o que pode ou não pode ser gerado pela linguagem, tentar escrever alguns textos fontes na linguagem e comparar com a especificação)
- Simplificar a linguagem transformando a mesma para a notação de Wirth com eliminação de recursão
- Transformar a gramática fornecida em um conjunto de autômatos ótimos equivalentes (para a implementação de parte do analisador sintático da linguagem), conforme processo a ser visto em sala de aula
- Planejar as rotinas para funcionamento isolado do analisador léxico (pensar nas rotinas, variáveis, lógica, estruturas, interface).
- Planejar as rotinas para funcionamento isolado da tabela de símbolos (pensar nas rotinas, variáveis, lógica, estruturas, interface).

- Planejar as rotinas gerais para que o analisador léxico e tabela de símbolos funcione (pensar nas rotinas, variáveis, lógica, estruturas, interface).
- Documentar planejamento das rotinas.
- Gerar arquivo de entrega com: a situação final da gramática depois de toda a simplificação realizada, cada um dos autômatos brutos transformados na notação de tabelas de transição, 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 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.
- Implementar o código das rotinas gerais de inicialização e abertura do texto fonte (no programa principal que vai se tornar o sintático no futuro)
- Implementar o código da análise léxico
- Implementar o código da tabela de símbolos
- Implementar as rotinas gerais de controle de escopo (no programa principal)
- Testar o código implementado
- Elaborar pequena documentação do projeto. Deve conter: informações de uso do projeto, particularidades de implementação, decisões tomadas pela equipe para o desenvolvimento do projeto, nome do executável, recursos adicionais que a equipe tenha implementado, e outras informações sobre o projeto que a equipe achar adequadas
- Gerar arquivo de entrega com: o código fonte do compilador com parte da análise 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. Última versão.

05. Geração dos autômatos ótimos equivalentes

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 Comp2018-1.

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).

06. 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. 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 Comp2018-1 será fornecida logo abaixo no item 10 desta especificação. 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.

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 seqüência 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 Comp2018-1, 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 Comp2018-1 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: 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 podem ser iniciados também com a cadeia " // " (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 Comp2018-1, os átomos possuem um limite máximo de 35 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 35 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 35 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 35 caracteres de um átomo. A leitura deve continuar mesmo depois dos 35 primeiros caracteres até encontrar o ponto onde vai existir algum caractere delimitador para aquele átomo.

No limite de 35 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 35, ou garantir que os números após truncar os elementos após a posição 35 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 .181 e irá gerar o relatório .LEX.

07. Tabela de Símbolos

Apenas os átomos identificadores serão armazenados na tabela de símbolos. As palavras e símbolos reservados da linguagem Comp2018-1 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 35 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: FL,(ponto flutuante) IN (inteiro), ST (string), CH (character), BO (booleano), VO (void), AF,(array de ponto flutuante) AI (array de inteiro), AS (array de string), AC (array de character), AB (array de booleano),. Esta informação não deve ser modificada durante a análise.

As quantidades de caracteres do lexeme levando em conta apenas os 35 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 35 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.

10. Definição dos Átomos da Linguagem Comp2018-1

Átomo	Cód	Átomo	Cód	Átomo	Cód	Átomo	Cód	Átomo	Cód	Átomo	Cód
Bool	A01	End	A09	!=	B01	!	B12	Character	C01	Submáquina1	M01
While	A02	Return	A10	#	B01	%	B13	Constant-String	C02	Submáquina2	M02
Break	A03	False	A11	&	B02	)	B14	Float-Number	C03	Submáquina3	M03
Void	A04	Program	A12	(	B03	*	B15	Function	C04	...	
Char	A05	Float	A13	/	B04	,	B16	Identifier	C05	Submáquinan	M0n
True	A06	Int	A14	;	B05	]	B17	Integer-Number	C06		
Else	A07	If	A15	[	B06	|	B18				
String	A08			{	B07	}	B19				
				+	B08	<	B20				
				<=	B09	==	B21				
				=	B10	>	B22				
				>=	B11	-	B23				

11. Definição da Gramática da Linguagem COMP2018-1 – Sintaxe dos comandos

Todos os símbolos marcados em azul no texto abaixo são símbolos do alfabeto da linguagem e não da notação BNF
Todos os não terminais marcados em vermelho são átomos da linguagem e do ponto de vista sintático devem ser considerados como se fossem terminais.

Gramática para a sintaxe dos comandos:

G = (V, Σ, P, program)

V = { program, declaration-list, declaration, variable-declaration, function-declaration, type-specifier, var-decl-list, var-decl-id, type-specifier, function-declaration, params, param-list, param-type-list, param-id-list, param-id, compound-stmt, local-declarations, statement-list, statement, expression-stmt, selection-stmt, iteration-stmt, return-stmt, break-stmt, expression, var, simple-expression, or-expression, unary-rel-expression, rel-expression, relop, add-expression, addop, term, mulop, unary-expression, factor, constant, number, string, call, args, arg-list, begin, end, [,] , ; , , , Identifier, Integer-Number, float, int, string, void, bool, char, (,), { , } , if, else, while, return, break, |, &, !, =, >=, <, <=, ==, !=, #, +, -, *, /, true, false, Float-number, Constant-String, Character, Function }

Σ = { begin, end, [,] , ; , , , Identifier, Integer-Number, float, int, string, void, bool, char, (,), { , } , if, else, while, return, break, |, &, !, =, >=, <, <=, ==, !=, #, +, -, *, /, true, false, Float-number, Constant-String, Character, Function }

P = {

program	→ begin declaration-list end	,
declaration-list	→ declaration-list declaration	,
declaration-list	→ declaration	,
declaration	→ variable-declaration	,
declaration	→ function-declaration	,
variable-declaration	→ type-specifier var-decl-list	,
variable-declaration	→ type-specifier [ ] var-decl-list ;	,
var-decl-list	→ var-decl-list , var-decl-id	,
var-decl-list	→ var-decl-id	,
var-decl-id	→ Identifier	,
var-decl-id	→ Identifier [Integer-Number]	,
type-specifier	→ float	,
type-specifier	→ int	,
type-specifier	→ string	,
type-specifier	→ void	,
type-specifier	→ bool	,
type-specifier	→ char	,
function-declaration	→ type-specifier Function (params) statement	,
params	→ param-list	,
params	→ ε	,
param-list	→ param-list ; param-type-list	,
param-list	→ param-type-list	,
param-type-list	→ type-specifier param-id-list	,
param-id-list	→ param-id-list , param-id	,
param-id-list	→ param-id	,
param-id	→ Identifier	,
param-id	→ Identifier [Integer-Number]	,
compound-stmt	→ { local-declarations statement-list }	,
local-declarations	→ local-declarations var-declaration	,
local-declarations	→ ε	,
statement-list	→ statement-list statement	,
statement-list	→ ε	,
statement	→ expression-stmt	,
statement	→ compound-stmt	,
statement	→ selection-stmt	,
statement	→ iteration-stmt	,
statement	→ return-stmt	,
statement	→ break-stmt	,
expression-stmt	→ expression ;	,
expression-stmt	→ ;	,
selection-stmt	→ if (expression) statement	,
selection-stmt	→ if (expression) statement else statement	,
iteration-stmt	→ while (expression) statement	,
return-stmt	→ return ;	,
return-stmt	→ return expression ;	,
break-stmt	→ break ;	,
expression	→ var = expression	,
expression	→ simple-expression	,

```
var                  → Identifier                                    ,
var                  → Identifier [ Integer-Number ]                 ,
simple-expression    → simple-expression |  or-expression            ,
simple-expression    → or-expression                                 ,
or-expression        → or-expression &  unary-rel-expression         ,
or-expression        → unary-rel-expression                          ,
unary-rel-expression → ! unary-rel-expression                        ,
unary-rel-expression → rel-expression                                ,
rel-expression       → add-expression relop add-expression           ,
rel-expression       → add-expression                                ,
relop                → =                                             ,
relop                → >                                             ,
relop                → >=                                            ,
relop                → <                                             ,
relop                → <=                                            ,
relop                → ==                                            ,
relop                → !=                                            ,
relop                → #                                             ,
add-expression       → add-expression addop term                     ,
add-expression       → term                                          ,
addop                → +                                             ,
addop                → -                                             ,
term                 → term mulop unary-expression                   ,
term                 → unary-expression                              ,
mulop                → *                                             ,
mulop                → /                                             ,
unary-expression     → - unary-expression                            ,
unary-expression     → factor                                        ,
factor               → ( expression )                                ,
factor               → var                                           ,
factor               → call                                          ,
factor               → constant                                      ,
constant             → number                                        ,
constant             → string                                        ,
constant             → true                                          ,
constant             → false                                         ,
number               → Integer-number                                ,
number               → Float-number                                  ,
string               → Constant-String                               ,
string               → Character                                     ,
call                 → Function ( args )                             ,
args                 → arg-list                                      ,
args                 → ε                                             ,
arg-list             → arg-list , expression                         ,
arg-list             → expression                                    }
```

12. Definição da Gramática da Linguagem COMP2018-1 – Padrões Léxicos de formação

```
< Identifier >          ::= < letra > | _ | < Identifier > <letra> | < Identifier > < digito > < Identifier > _
< Function >            ::= < letra > | < Identifier > <letra> | < Identifier > < digito >
< Integer-Number >      ::= < decimal_digits >
< decimal_digits >      ::=  < digito > |  < decimal_digits > < digito > .
< Float-Number >        ::= < decimal_digits > .  < decimal_digits >  | < decimal_digits > .  < decimal_digits > < exponent_part >
< exponent_part >       ::=  e < decimal_digits > | e - < decimal_digits > | e + < decimal_digits >
< Constant-String >     ::= "'" < Middle-String > "'"            //inicia e termina com aspas duplas
< Middle-String >       ::= ( < letra > | < branco > | < digito> | $ | _ | . ) < Middle-String > | < letra > | < branco > | < digito> | $ | _ | .
< Character >           ::= "'" < 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".
```