

README.doc - Descrição do trabalho

Nome do software: Compilador

Integrantes da equipe: 1. Levi Cicero Arcanjo

Descrição da linguagem X:

- Subconjunto de C, incluindo declaração de variáveis int, char, float. Permitindo o uso dos operadores de soma, multiplicação, subtração, divisão, operadores lógicos maior e menor, comandos condicionais (if e else) e loops (while e for).
- Nomes variáveis podendo iniciar com qualquer letra de a-z, A-Z ou underline seguidas por qualquer letra de a-z, A-Z, underline e números.
- Palavras reservadas são: int, char, float, void, if, else, while, main, return e for

Classes de Tokens:

Palavra int	-> Expressão regular: <code>^[i][n][t]\$</code>
Palavra char	-> Expressão regular: <code>^[c][h][a][r]\$</code>
Palavra float	-> Expressão regular: <code>^[f][l][o][a][t]\$</code>
Palavra void	-> Expressão regular: <code>^[v][o][i][d]\$</code>
Palavra if	-> Expressão regular: <code>^[i][f]\$</code>
Palavra else	-> Expressão regular: <code>^[e][l][s][e]\$</code>
Palavra while	-> Expressão regular: <code>^[w][h][i][l][e]\$</code>
Palavra return	-> Expressão regular: <code>^[r][e][t][u][r][n]\$</code>
Palavra for	-> Expressão regular: <code>^[f][o][r]\$</code>
símbolo abre parêntesis	-> Expressão regular: <code>^([)]\$</code>
Símbolo fecha parênteses	-> Expressão regular: <code>^)]\$</code>
Símbolo abre chaves	-> Expressão regular: <code>^{[]\$</code>
Símbolo fecha chaves	-> Expressão regular: <code>^}]\$</code>
Símbolo de soma	-> Expressão regular: <code>^[+]\$</code>
Símbolo de subtração	-> Expressão regular: <code>^[+]\$</code>
Símbolo de multiplicação	-> Expressão regular: <code>^[*]\$</code>
Símbolo de divisão	-> Expressão regular: <code>^[/]\$</code>
Símbolo ponto e vírgula	-> Expressão regular: <code>^[;]\$</code>
Símbolo de vírgula	-> Expressão regular: <code>^[,]\$</code>
Símbolo de atribuição	-> Expressão regular: <code>^[=]\$</code>
Símbolo and	-> Expressão regular: <code>^[&]\$</code>
Símbolo ou	-> Expressão regular: <code>^[\$]</code>
Símbolo maior	-> Expressão regular: <code>^[>]\$</code>
Símbolo menor	-> Expressão regular: <code>^[<]\$</code>
Símbolo abre colchetes	-> Expressão regular: <code>^[[]\$</code>
Símbolo fecha colchetes	-> Expressão regular: <code>^[\\]\$</code>

Caracter	-> Expressão regular: <code>^[^\.]*\$</code>
String	-> Expressão regular: <code>^[^"].*"\$</code>
Número	-> Expressão regular: <code>^[0-9]+\$</code>
Número decimal	-> Expressão regular: <code>^[0-9]+\.[0-9]+\$</code>
Identificador	-> Expressão regular: <code>^[a-zA-Z][a-zA-Z0-9]*\$</code>

Trecho de código associados às classes de tokens:

A variável “tokenClasses” é atribuída com expressões regulares que serão utilizadas no programa. Neste código a variável “tokenClasses” é um array imutável de objetos, sendo cada objeto composto por uma nomenclatura da classe de token (atributo “tokenClass”) e sua expressão regular correspondente (atributo “re”).

```
/**
 * Definição das classes de token da linguagem
 */
const tokenClasses = [
  { re: /^[i][n][t]$/, tokenClass: "int" },
  { re: /^[c][h][a][r]$/, tokenClass: "char" },
  { re: /^[f][l][o][a][t]$/, tokenClass: "float" },
  { re: /^[v][o][i][d]$/, tokenClass: "void" },
  { re: /^[i][f]$/, tokenClass: "if" },
  { re: /^[e][l][s][e]$/, tokenClass: "else" },
  { re: /^[w][h][i][l][e]$/, tokenClass: "while" },
  { re: /^[r][e][t][u][r][n]$/, tokenClass: "return" },
  { re: /^[f][o][r]$/, tokenClass: "for" },
  { re: /^[()]/, tokenClass: "openParentheses" },
  { re: /^[)]$/, tokenClass: "closeParentheses" },
  { re: /^[{]/, tokenClass: "openBrace" },
  { re: /^[}]/, tokenClass: "closeBrace" },
  { re: /^[+]/, tokenClass: "plus" },
  { re: /^[ - ]$/, tokenClass: "minus" },
  { re: /^[*]/, tokenClass: "multiply" },
  { re: /^[ / ]$/, tokenClass: "div" },
  { re: /^[;]/, tokenClass: "semicolon" },
  { re: /^[,]/, tokenClass: "coma" },
  { re: /^[=]/, tokenClass: "equal" },
  { re: /^[&]/, tokenClass: "andBinary" },
  { re: /^[>]/, tokenClass: "greater" },
  { re: /^[<]/, tokenClass: "less" },
  { re: /^[[/, tokenClass: "openBracket" },
  { re: /^[\\]/, tokenClass: "closeBracket" },
  { re: /^[ ' ]$/, tokenClass: "character" },
  { re: /^[ " ]$/, tokenClass: "string" },
  { re: /^[0-9]+$/, tokenClass: "number" },
  { re: /^[0-9]+\.[0-9]+$/, tokenClass: "decimal" },
  { re: /^[a-zA-Z][a-zA-Z0-9]*$/, tokenClass: "id" },
];
```

Regras sintáticas em notação similar a EBNF:

Notação não-terminais: <NÃO-TERMINAIS>

Notação terminais: terminais

Notação palavra vazia: λ

Regras Sintáticas		
<S>	::=	<TYPE> <IDENTIFIER> <S0> <S>
<S>	::=	λ
<S0>	::=	; <S0_>
<S0>	::=	<FUNCTION_>
<DECLARATION>	::=	<TYPE> <IDENTIFIER> <DECLARATION_>
<DECLARATION_>	::=	;
<DECLARATION_>	::=	, <IDENTIFIER> <DECLARATION_>
<IDENTIFIER>	::=	id
<TYPE>	::=	int
<TYPE>	::=	float
<TYPE>	::=	char
<TYPE>	::=	void
<VALUE>	::=	number
<VALUE>	::=	decimal
<VALUE>	::=	string
<VALUE>	::=	character
<FUNCTION>	::=	<TYPE> <IDENTIFIER> <FUNCTION_>
<FUNCTION_>	::=	(<F0>) <STATEMENT>
<F0>	::=	<TYPE> <IDENTIFIER> <F1>
<F0>	::=	λ
<F1>	::=	, <F0>
<F1>	::=	[<F2>]
<F1>	::=	λ
<F2>	::=	number
<F2>	::=	λ
<ITERATION>	::=	while (<EXPRESION>) <STATETMENT>
<ITERATION>	::=	for (<ITERATION_> ; <ITERATION_> ; <ITERATION_>) <STATEMENT>
<SELECTION>	::=	if (<EXPRESION>) <STATEMENT> <ELSE>
<ELSE>	::=	else <STATEMENT>

<ELSE>	::=	λ
<OPERATOR>	::=	+ - * / < > = &
<STATEMENT__>	::=	<SELECTION>
<STATEMENT__>	::=	<ITERATION>
<STATEMENT__>	::=	<EXPRESION> ;
<STATEMENT__>	::=	<RETURN>
<RETURN>	::=	return <ITERATION_> ;
<S0_>	::=	<S>
<S0_>	::=	λ
<ASSIGNMENT>	::=	<PRIMARY> <ASSIGNMENT_>
<ASSIGNMENT_>	::=	<OPERATOR> <ASSIGNMENT>
<ASSIGNMENT_>	::=	λ
<PRIMARY>	::=	<IDENTIFIER>
<PRIMARY>	::=	<VALUE>
<EXPRESION>	::=	<ASSIGNMENT> <EXPRESION_>
<EXPRESION_>	::=	, <ASSIGNMENT> <EXPRESION_>
<EXPRESION_>	::=	λ
<STATEMENT>	::=	{ <STATEMENT_> }
<STATEMENT_>	::=	<STATEMENT__> <STATEMENT_>
<STATEMENT_>	::=	<DECLARATION> <STATEMENT_>
<STATEMENT_>	::=	λ
<ITERATION_>	::=	<EXPRESION>
<ITERATION_>	::=	λ

Trecho de código associados à regra gramatical:

Regra associada neste trecho de código:

$\langle \text{EXPRESION_} \rangle ::= , \langle \text{ASSIGNMENT} \rangle \langle \text{EXPRESION_} \rangle \mid \lambda$

```
/**
 * Função que representa uma regra gramatical EXPRESION_
 */
function EXPRESION_() {
  if (symbol.tokenClass === "coma") {
    tree.push("<EXPRESION_> ::= , <ASSIGNMENT> <EXPRESION_>");

    getNextSimbol();

    ASSIGNMENT();
    EXPRESION_();
  } else {
    tree.push("<EXPRESION_> ::= λ");
  }
}
```

Regras sintáticas e gramática de atributos correspondente:

Regras Sintáticas		Gramática de atributos
$\langle S \rangle$	$::= \langle \text{TYPE} \rangle \langle \text{IDENTIFIER} \rangle \langle S_0 \rangle \langle S \rangle$	$\langle \text{IDENTIFIER} \rangle.\text{type} = \langle \text{TYPE} \rangle$
$\langle S \rangle$	$::= \lambda$	
$\langle S_0 \rangle$	$::= ; \langle S_0 \rangle$	
$\langle S_0 \rangle$	$::= \langle \text{FUNCTION_} \rangle$	
$\langle \text{DECLARATION} \rangle$	$::= \langle \text{TYPE} \rangle \langle \text{IDENTIFIER} \rangle \langle \text{DECLARATION_} \rangle$	$\langle \text{IDENTIFIER} \rangle.\text{type} = \langle \text{TYPE} \rangle$
$\langle \text{DECLARATION_} \rangle$	$::= ;$	
$\langle \text{DECLARATION_} \rangle$	$::= , \langle \text{IDENTIFIER} \rangle \langle \text{DECLARATION_} \rangle$	
$\langle \text{IDENTIFIER} \rangle$	$::= \text{id}$	$\text{id.type} = \langle \text{IDENTIFIER} \rangle.\text{type}$

<TYPE>	::= int	
<TYPE>	::= float	
<TYPE>	::= char	
<TYPE>	::= void	
<VALUE>	::= number	<TYPE>.type = int
<VALUE>	::= decimal	<TYPE>.type = float
<VALUE>	::= string	<TYPE>.type = char
<VALUE>	::= character	<TYPE>.type = char
<FUNCTION>	::= <TYPE> <IDENTIFIER> <FUNCTION_>	<IDENTIFIER>.type = <TYPE>
<FUNCTION_>	::= (<F0>) <STATEMENT>	
<F0>	::= <TYPE> <IDENTIFIER> <F1>	<IDENTIFIER>.type = <TYPE>
<F0>	::= λ	
<F1>	::= , <F0>	
<F1>	::= [<F2>]	
<F1>	::= λ	
<F2>	::= number	
<F2>	::= λ	
<ITERATION>	::= while (<EXPRESION>) <STATEMENT>	<ITERATION>.scope = currentScope + 1
<ITERATION>	::= for (<ITERATION_> ; <ITERATION_> ; <ITERATION_>) <STATEMENT>	<ITERATION>.scope = currentScope + 1
<SELECTION>	::= if (<EXPRESION>) <STATEMENT> <ELSE>	<SELECTION>.scope = currentScope + 1
<ELSE>	::= else <STATEMENT>	<ELSE>.scope = currentScope + 1
<ELSE>	::= λ	
<OPERATOR>	::= + - * / < > = &	
<STATEMENT__>	::= <SELECTION>	
<STATEMENT__>	::= <ITERATION>	
<STATEMENT__>	::= <EXPRESION> ;	
<STATEMENT__>	::= <RETURN>	
<RETURN>	::= return <ITERATION_> ;	
<S0_>	::= <S>	
<S0_>	::= λ	
<ASSIGNMENT>	::= <PRIMARY> <ASSIGNMENT_>	<PRIMARY>.type =

			<ASSIGNMENT_>.type, <ASSIGNMENT>.value = <PRIMARY>.value
<ASSIGNMENT_>	::=	<OPERATOR> <ASSIGNMENT>	
<ASSIGNMENT_>	::=	λ	
<PRIMARY>	::=	<IDENTIFIER>	<PRIMARY>.type = <IDENTIFIER>.type
<PRIMARY>	::=	<VALUE>	<PRIMARY>.type = <VALUE>.type
<EXPRESION>	::=	<ASSIGNMENT> <EXPRESION_>	<EXPRESION>.value = <ASSIGNMENT>.value
<EXPRESION_>	::=	, <ASSIGNMENT> <EXPRESION_>	
<EXPRESION_>	::=	λ	
<STATEMENT>	::=	{ <STATEMENT_> }	
<STATEMENT_>	::=	<STATEMENT__> <STATEMENT_>	
<STATEMENT_>	::=	<DECLARATION> <STATEMENT_>	
<STATEMENT_>	::=	λ	
<ITERATION_>	::=	<EXPRESION>	
<ITERATION_>	::=	λ	

Descrição do funcionamento e utilização do software:

- Funcionamento: funciona em um navegador (preferência chrome ou firefox)
- Execução: Abrir o index.html com o navegador (preferência chrome ou firefox)
- Uso: selecionar o arquivo clicando no botão "selecionar arquivo", clicar no botão "Fazer análise" para iniciar a análise (léxica, sintática e semântica). Caso existam erros, serão mostrados na tela. Caso não existam erros de compilação será iniciado o download do arquivo da árvore de derivação sintática.

Descrição do tratamento de erros:

Erros léxicos:

Neste analisador os erros léxicos são identificados através de expressões regulares, logo após ser constatado que o token está mal escrito é chamado um método denominado `describeLexicalError()` que determina qual tipo de erro o token apresenta.

Este analisador léxico reconhece os seguintes erros especificamente:

- Identificador mal formado -> Quando um identificador inicia com números
- Número mal formado-> Quando existe um número com letras e pontos
- Caráter mal formado -> Quando um caractere possui tamanho maior que um ou aspas simples a menos
- String mal formada -> Quando possui aspas duplas a menos
- Símbolo desconhecido -> Quando o símbolo apresentado não faz parte da linguagem
- Tamanho excessivo -> Quando o símbolo apresentado ultrapassa a quantidade permitida de caracteres.

Erros sintáticos:

Neste analisador sintático os erros são identificados através de métodos recursivos. Cada método possui comandos de seleção que verificam a ordem esperada de tokens durante a análise. Cada método recursivo, quando detecta a ocorrência de um erro, determina de forma específica o tipo de erro com base no token esperado. Caso o token seja diferente do esperado é empilhado um erro e a análise ignora esse token faltante, seguindo a análise sintática como se não houvessem ocorrido erros até o momento.

Os erros sintáticos são descritos por uma mensagem que diz qual o token esperado seguido de sua linha e o token anterior.

Erros Semânticos:

Neste analisador semântico os erros são identificados de forma específica, ou seja, para cada tipo de erro existe um trecho de código único para detectá-lo e tratá-lo.

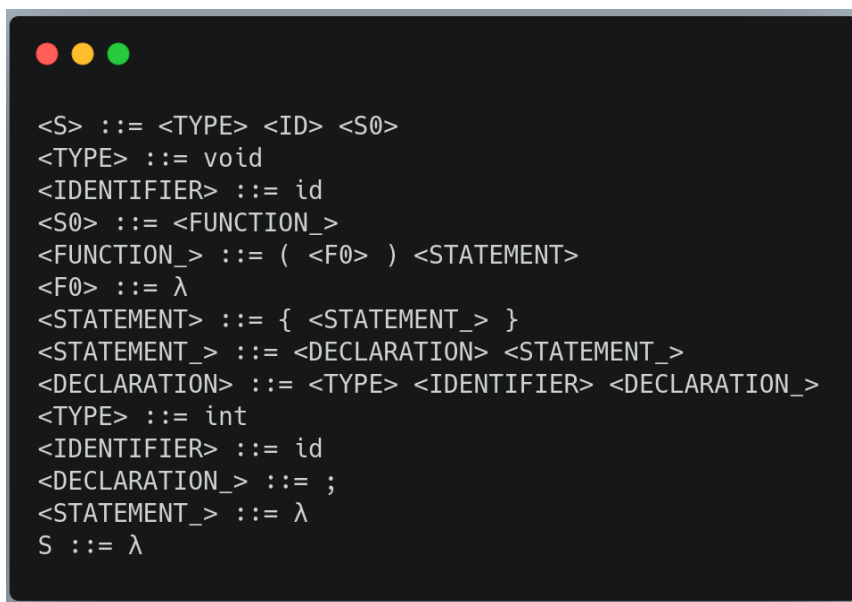
Os erros semânticos detectados neste analisador são:

- Identificador já declarado
- Identificador não declarado
- Divergência de tipos

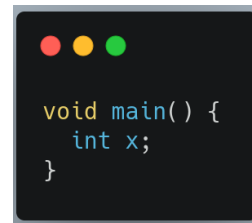
Descrição do processo de construção:

- Não é necessário nenhum tipo de processo de build
- Apenas é necessário ter instalado um navegador web

Árvore sintática(imagem a esquerda) referente um trecho de código (imagem a direita):



```
<S> ::= <TYPE> <ID> <S0>
<TYPE> ::= void
<IDENTIFIER> ::= id
<S0> ::= <FUNCTION_>
<FUNCTION_> ::= ( <F0> ) <STATEMENT>
<F0> ::= λ
<STATEMENT> ::= { <STATEMENT_> }
<STATEMENT_> ::= <DECLARATION> <STATEMENT_>
<DECLARATION> ::= <TYPE> <IDENTIFIER> <DECLARATION_>
<TYPE> ::= int
<IDENTIFIER> ::= id
<DECLARATION_> ::= ;
<STATEMENT_> ::= λ
S ::= λ
```



```
void main() {
    int x;
}
```

Referências usadas:

- <https://www.w3schools.com/>
- <https://developer.mozilla.org/>
- <https://javascript.info/>
- <https://pt.stackoverflow.com/>
- Slides de compiladores (Aulas 0, 1 e 2, 3, 4, 5, 6)