UNIVERSIDADE FEDERAL DO PIAUÍ – UFPI
CENTRO DE CIÊNCIAS DA NATUREZA – CCN
DEPARTAMENTO DE COMPUTAÇÃO – DC
DISCIPLINA: COMPILADORES
DOCENTE: GLAUBER DIAS GONÇALVES

Trabalho Final - Especificação

Gabriel Lopes Bastos
José Victor Vieira de Oliveira
Pedro Emanuel Moreira Carvalho

Teresina
2025

TypeScript Simplificado

TypeScript Simplificado é um subconjunto da linguagem TypeScript projetado para ser compilado. A linguagem é fortemente tipada, operando com os tipos básicos number, string e boolean, além de arrays (number[ ], string[ ]) como tipos derivados. Ela adota o escopo de bloco moderno com let e const, excluindo var. Variáveis podem ser declaradas sem inicialização, porém devem ser atribuídas antes de serem utilizadas, sendo essa verificação realizada estaticamente durante a compilação. A linguagem não suporta aninhamento de funções e possui um conjunto de funções nativas para operações matemáticas e de saída através de Math, console.log e funções de conversão de tipos.

1. Tipos Básicos e Derivados

TypeScript Simplificado é fortemente tipado e oferece suporte aos seguintes tipos:

1.1. Tipos Básicos

1.1.1. *number*

O tipo number no TypeScript pode representar tanto um inteiro (int) quanto um número com ponto flutuante (float).

Exemplos:

```
let idade: number = 25;
const pi: number = 3.14159;
let negativo: number = -42;
let zero: number;
```

1.1.2. *boolean*

Representa valores lógicos: verdadeiro ou falso.

Exemplos:

```
const sim: boolean = true;
let concluido: boolean = false;
let flag: boolean;
```

1.1.3. *string*

Representa sequências de caracteres (texto).

Exemplos:

```
let nome: string = "Maria";
let mensagem: string = 'Olá, mundo!';
let vazia: string = "";
```

1.1.4. *void*

Representa a ausência de valor. Utilizado exclusivamente como tipo de retorno de funções que não retornam nenhum valor.

Exemplos:

```
function exibirMensagem(texto: string): void {
    console.log(texto);
}

function processar(): void {
    // código sem retorno
}
```

Observação: Variáveis não podem ser declaradas com o tipo `void`. Este tipo é válido apenas para retorno de funções.

1.2. Tipos Derivados

1.2.1. *number[ ]*

Representa um array (lista) de valores numéricos de tamanho fixo.

Exemplos:

```
let numeros: number[] = [1, 2, 3, 4, 5];
let notas: number[] = [7.5, 8.0, 9.2];
let vazio: number[] = [];
```

Operações:

- Acesso por índice: `numeros[0]` retorna o primeiro elemento
- Tamanho do array: `numeros.length` retorna a quantidade de elementos
- Atribuição: `numeros[2] = 10` modifica o terceiro elemento

1.2.2. *string[ ]*

Representa um array (lista) de strings de tamanho fixo.

Exemplos:

```
let nomes: string[] = ["Ana", "Bruno", "Carlos"];
let palavras: string[] = ["typescript", "compilador",
"projeto"];
let vazio: string[] = [];
```

Operações:

- Acesso por índice: `nomes[0]` retorna a primeira string
- Tamanho do array: `nomes.length` retorna a quantidade de elementos
- Atribuição: `nomes[1] = "Beatriz"` modifica o segundo elemento

2. Operadores e a precedência dos operadores

2.1. Operadores

- ‘+’: soma - binário (number)
- ‘-’: subtração - binário (number)
- ‘-’: menos, inverte o valor ao qual foi aplicado – unário (number)
- ‘*’: multiplicação - binário (number)
- ‘/’: divisão - binário (number)
- ‘**’: potência - binário (number)
- ‘%’: resto/módulo - binário (number)
- ‘>’: maior que - binário (number)
- ‘<’: menor que - binário (number)
- ‘>=’: maior que ou igual - binário (number)
- ‘<=’: menor que ou igual - binário (number)
- ‘==’: igual - binário (number, string, boolean)
- ‘!=’: diferente - binário (number, string, boolean)
- ‘!’: negação - unário (boolean)
- ‘&&’: and lógico - binário (boolean)
- ‘||’: or lógico - binário (boolean)

2.2. Precedência de Operadores

Operador	Aridade	Tipos aplicados	Tipo de retorno
()	Agrupamento	Qualquer	O tipo da expressão
!	1	boolean	boolean
-	1	number	number
****	2	number	number
***, /, %**	2	number	number
+, -	2	number	number
>, <, >=, <=	2	number	boolean
==, !=	2	number, string, boolean	boolean
&&, ||	2	boolean	boolean

3. Regras para uso de variáveis, constantes e seu escopo

Esta seção detalha como identificadores, variáveis e constantes são definidos e gerenciados na linguagem, incluindo suas regras de visibilidade (escopo).

3.1. Identificadores

São os nomes usados para identificar variáveis, constantes e funções.

- Regras de Nomenclatura: Devem obrigatoriamente começar com uma letra (a-z, A-Z) ou underscore (_). Após o caractere inicial, podem conter letras, dígitos (0-9) ou underscores.
- Case-Sensitivity: TypeScript Simplificado é sensível a maiúsculas e minúsculas. O identificador nome é diferente de Nome ou NOME.
- Palavras Reservadas: Identificadores não podem ter o mesmo nome de uma palavra reservada da linguagem (ex: let, const, if, while, for, function, return, number, string, boolean, void, etc.).

3.2. Declaração de Variáveis e Constantes

TypeScript Simplificado utiliza as palavras-chave let e const para declarações, e **não** utiliza var. A especificação do tipo é obrigatória na declaração.

3.2.1. Variáveis (let)

Usada para declarar variáveis cujo valor pode ser alterado após a inicialização.

Sintaxe:

let <identificador>: <tipo> [= <valor_inicial>];

- Inicialização Opcional: O valor inicial não é obrigatório no momento da declaração.
- Análise Estática: Se uma variável for declarada sem valor, o compilador deve garantir (através de análise estática) que ela receba uma atribuição antes de ser utilizada em qualquer expressão. A tentativa de uso antes da atribuição deve gerar um erro de compilação.

Exemplos:

```
let idade: number;
let nome: string = "João";
let notas: number[] = [8, 7.5, 9];
idade = 30; // Válido
nome = "Ana"; // Válido
// let teste: number = idade + 10; // Válido, pois
'idade' foi atribuída
// let outro: number = novaVar + 5; // ERRO: 'novaVar'
não foi atribuída
// let nome: string = "Júlia"; // ERRO: 'nome' já foi
declarada
```

3.2.2. Constantes (const)

Usada para declarar constantes, cujo valor é fixo e não pode ser alterado após a inicialização.

- Sintaxe: const <identificador>: <tipo> = <valor_inicial>;
- Inicialização Obrigatória: Constantes devem ser inicializadas no momento da declaração. A tentativa de declará-la sem um valor deve gerar um erro de compilação.
- Imutabilidade: Qualquer tentativa de reatribuir um valor a uma constante após sua declaração deve gerar um erro de compilação.

Exemplos:

```
const PI: number = 3.14159;
const ADMIN_USER: string = "admin";
const ATIVO: boolean = true;
// PI = 3.14; // ERRO: Não é possível reatribuir uma
constante
// const TAXA: number; // ERRO: Constante precisa de
inicializador
```

3.3. Escopo

O escopo define a visibilidade de um identificador (variável, constante ou função) onde pode ser acessado.

- Escopo Global: Identificadores declarados fora de qualquer função ou bloco ({ ... }) são considerados globais e acessíveis de qualquer parte do programa.
- Escopo de Bloco (Local): let e const possuem escopo de bloco. Isso significa que são visíveis apenas dentro do bloco ({ ... }) em que foram declaradas (incluindo blocos de funções, if, while, etc.) e em sub-blocos.
- Re-declaração: Não é permitido declarar novamente um identificador usando let ou const no mesmo escopo.

Exemplos:

```
const TAXA_GLOBAL: number = 0.05;
let nomeGlobal: string = "App";
function calcular(valor: number): number {
    let taxaLocal: number = 0.10; // Escopo local da função

    if (valor > 100) {
        const BONUS: number = 10; // Escopo local do 'if'
        // 'TAXA_GLOBAL', 'nomeGlobal', 'valor', 'taxaLocal' e
'BONUS' são visíveis aqui
        return (valor * taxaLocal) + BONUS + TAXA_GLOBAL;
}
```

```
    // 'BONUS' não é visível aqui (fora do escopo do 'if')
    // return valor * BONUS; // ERRO: 'BONUS' não definido

    return valor * taxaLocal; // Válido
}

// 'taxaLocal' e 'BONUS' não são visíveis aqui (escopo global)
// let novoValor: number = taxaLocal; // ERRO
```

4. Declaração de Funções

Esta seção define como funções são declaradas, chamadas e gerenciadas na linguagem.

4.1. Sintaxe de Declaração

A declaração de uma função no TypeScript Simplificado segue a sintaxe padrão do TypeScript, usando a palavra-chave function.

```
function <nome_funcao>(<lista_de_parametros>): <tipo_retorno>
{
    // Corpo da função (bloco de código)
    // Declarações de 'let' e 'const' locais

    return <valor_retorno>;
}
```

Onde:

- `nome_funcao`: O nome segue o padrão dos identificadores e deve ser único.
- `lista_de_parametros`: Uma lista (podendo ser vazia) de parâmetros separados por vírgula.
- `tipo_retorno`: O tipo do valor que a função retorna. Deve ser void se a função não retornar valor, ou seja, o `return <valor_retorno>;` vira opcional.
- `valor_retorno`: O valor de retorno deve ser do mesmo tipo da função

4.2. Parâmetros

4.2.1. Sintaxe de Parâmetros:

`<identificador>: <tipo>`

4.2.2. Regras:

- Todos os parâmetros devem ter tipo explícito
- Parâmetros são separados por vírgula
- Não há suporte para parâmetros opcionais ou valores padrão

Exemplos:

```
// Função sem parâmetros
function obterPi(): number {
    return 3.14159;
}

// Função com um parâmetro
function dobro(x: number): number {
    return x * 2;
}

// Função com parâmetros de tipos diferentes
function cumprimentar(nome: string, idade: number):
string {
     return "Olá " + nome + ", você tem " + idade + "
anos";
}

// Função com array como parâmetro
function somarArray(numeros: number[]): number {
    let total: number = 0;
    let i: number = 0;
    while (i < numeros.length) {
        total = total + numeros[i];
        i = i + 1;
    }
    return total;
}
```

4.3. Tipo de Retorno

4.3.1. Funções que Retornam Valor

Funções que retornam valor devem:

- Declarar o tipo de retorno após os parênteses dos parâmetros
- Usar a instrução return seguida de uma expressão do tipo declarado
- Garantir que todos os caminhos de execução retornem um valor

Exemplo:

```
function media(n1: number, n2: number): number {
    return (n1 + n2) / 2;
}

function absoluto(x: number): number {
    if (x < 0) {
        return -x;
```

```
    }
    return x;  // Caminho alternativo também retorna
}
```

Erro - caminho sem retorno:

```
function maiorQue(a: number, b: number): number {
    if (a > b) {
        return a;
    }
    // ERRO: falta 'return' para o caso onde a <= b, além
de ser obrigatório no final da função
}
```

4.3.2. Funções void

Funções que não retornam valor usam void como tipo de retorno:

- Não devem usar return com valor
- Podem usar return; sozinho para interromper a execução

```
function imprimirMensagem(texto: string): void {
    console.log(texto);
}

function processarNumero(n: number): void {
    if (n < 0) {
        console.log("Número negativo");
        return;  // Interrompe a execução
    }
    console.log("Número positivo");
}

function mostrarLista(itens: string[]): void {
    let i: number = 0;
    while (i < itens.length) {
        console.log(itens[i]);
        i = i + 1;
    }
    // Não precisa de 'return' explícito
}
```

4.4. Chamada de Funções

Sintaxe:

```
<identificador>(<lista_argumentos>)
```

Regras:

- O número de argumentos deve corresponder ao número de parâmetros
- Cada argumento deve ser do tipo esperado pelo parâmetro correspondente

- A ordem dos argumentos importa

Exemplos:

```
function soma(a: number, b: number): number {
    return a + b;
}

function exibir(mensagem: string, valor: number): void {
    console.log(mensagem);
    console.log(valor);
}

// Chamadas válidas
let resultado: number = soma(10, 20);
let x: number = 5;
let y: number = soma(x, 15);
exibir("Total:", resultado);

// Chamadas inválidas
// soma(10);              // ERRO: faltam argumentos
// soma(10, 20, 30);      // ERRO: muitos argumentos
// soma("10", 20);        // ERRO: tipo incorreto no primeiro
argumento
// let z: string = soma(5, 10);  // ERRO: tipo de retorno
incompatível
```

5. Funções Nativas

Esta seção define as funções e propriedades globais que são providas nativamente pelo compilador. Elas estão disponíveis em qualquer escopo, sem a necessidade de declaração.

5.1. Objeto console - `console.log`

Função nativa de saída de dados. Imprime um ou mais valores no console padrão, seguido por uma quebra de linha.

Sintaxe:

```
console.log(<arg1>[, <arg2>, ...]);
```

Regras:

- Aceita um ou mais argumentos dos tipos básicos (number, string, boolean).
- Os argumentos são impressos na ordem em que aparecem, separados por um espaço.

Exemplos:

```
let idade: number = 30;
```

```
let nome: string = "Ana";
console.log("Usuário:"); // Saída: Usuário:
console.log(nome, idade, true); // Saída: Ana 30 true
```

5.2. Funções Globais de Conversão de Tipo

Essas funções são essenciais para lidar com a conversão entre os tipos string e number.

5.2.1. parseInt(texto: string): number

Analisa um argumento string e retorna um number inteiro. A análise para quando encontra o primeiro caractere não numérico.

Exemplo:

```
let s: string = "123.45";
let n: number = parseInt(s);
// n agora vale 123
```

5.2.2. parseFloat(texto: string): number

Analisa um argumento string e retorna um number de ponto flutuante.

Exemplo:

```
let s: string = "3.14159";
let n: number = parseFloat(s);
// n agora vale 3.14159
```

5.3. Objeto Math

Um objeto nativo que agrupa funções matemáticas.

5.3.1. Math.sqrt(x: number): number

Retorna a raiz quadrada de x.

Exemplo:

```
let raiz: number = Math.sqrt(16); // raiz vale 4
```

5.3.2. Math.pow(base: number, expoente: number): number

Retorna a base elevada ao expoente. (Isso serve como alternativa ao operador **).

Exemplo:

```
let pot: number = Math.pow(2, 3); // pot vale 8
```

5.4. Propriedades Nativas

5.4.1. .length (Propriedade)

Uma propriedade nativa que retorna a quantidade de elementos em um array ou a quantidade de caracteres em uma string.

Tipos Aplicáveis: number[], string[], string

Tipo de Retorno: number

Exemplos:

```
let nomes: string[] = ["Ana", "Bruno"];
let totalNomes: number = nomes.length; // totalNomes vale
2

let msg: string = "Olá";
let tamMsg: number = msg.length; // tamMsg vale 3
```

6. Sentenças de Atribuição e Controle

Esta seção define as estruturas de código que controlam o fluxo de execução e a atribuição de valores.

6.1. Sentença de Atribuição

Atribui o valor de uma expressão a um identificador (variável let) ou a uma posição de um array.
Sintaxe (Variável): <identificador> = <expressao>;
Sintaxe (Array): <identificador_array>[<indice>] = <expressao>;
Regras:

- O identificador à esquerda deve ser uma variável (let). Não é permitido atribuir a uma const após sua inicialização.
- A expressão à direita deve resultar em um valor do mesmo tipo declarado da variável/array.
- No caso de array, o <indice> deve ser uma expressão do tipo number.

Exemplos:

```
let idade: number = 20;
idade = 21; // Válido [cite: 328]
let notas: number[] = [7.0, 8.0];
notas[0] = 7.5; // Válido [cite: 273]
```

6.2. Sentenças de Controle

6.2.1. if / else

Executa um bloco de código se uma condição booleana for verdadeira e, opcionalmente, outro bloco se for falsa.

Sintaxe:

```
if (<condicao_booleana>) {
        // Bloco de código 'if'
} [else {
        // Bloco de código 'else' (opcional)
}]
```

Regras:

- A <condicao_booleana> deve ser uma expressão que resulta em boolean.

- O `else` é associado ao `if` de acordo com a sintaxe de bloco “{}”.

Exemplo:
```
function absoluto(x: number): number {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```

6.2.2. while
Executa um bloco de código repetidamente enquanto uma condição booleana permanecer verdadeira.
Sintaxe:
```
while (<condicao_booleana>) {
  // Bloco de código (loop)
}
```
Regras: A condição é avaliada *antes* de cada iteração.

Exemplo:
```
function somarArray(numeros: number[]): number {
    let total: number = 0;
    let i: number = 0;
    while (i < numeros.length) {
        total = total + numeros[i];
        i = i + 1;
    }
    return total;
}
```

6.2.3. for
Um loop que combina inicialização, condição e incremento em uma única declaração.
Sintaxe:
```
for ([inicializacao]; [condicao_booleana]; [incremento])
{
  // Bloco de código (loop)
}
```

Regras:
- `inicializacao`: (Opcional) Executada uma vez antes do loop. Geralmente uma declaração let ou atribuição.
- `condicao_booleana`: (Opcional) Avaliada antes de cada iteração.

Se omitida, o loop é infinito.

- `incremento`: (Opcional) Executado ao final de cada iteração.

Exemplo:

```
function imprimirNomes(nomes: string[]): void {
    for (let i: number = 0; i < nomes.length; i = i + 1)
{
        console.log(nomes[i]);
    }
}
```