

CONSTRUÇÃO DE APLICAÇÃO PARA GERAR AUTÔMATO FINITO DETERMINÍSTICO.

Carlos Eduardo Thomas

RESUMO

O artigo descreve o desenvolvimento de um programa que converte uma string contendo tokens e gramáticas BNF em um Autômato Finito Determinístico (AFD). O programa utiliza a abordagem requisição/resposta e as funções de match e split da linguagem JavaScript para identificar os símbolos de cada token e gramática. A determinização é realizada baseada no Teorema da Determinização. **As palavras chave são:** Autômato Finito Determinístico, BNF, determinização, linguagem JavaScript.

1. INTRODUÇÃO

Para o seguinte trabalho foi solicitado a análise de uma sequência de caracteres ("*string*"), contendo tokens e gramáticas no formato bnfs no formato AFND (Autômato Finito Não Determinístico), para então serem convertidos no formato AFD (Autômato Finito Determinístico) por meio de um programa desenvolvido conforme as preferências do aluno seguindo os parâmetros estabelecidos pelo professor.

A abordagem utilizada para analisar a entrada de dados foi utilizar do método requisição/resposta, onde é esperado a string contendo os tokens e a gramática separados por um "\n" para que o sistema entenda o que deve ser feito.

2. DESENVOLVIMENTO

Para o desenvolvimento do programa, foram utilizados conceitos básicos de Autômatos Finitos Determinísticos (AFD) e não Determinísticos (AFND), que são modelos matemáticos utilizados para representar sistemas de computação.

Enquanto os AFDs possuem uma transição bem definida para cada estado e símbolo de entrada, os AFNDs podem possuir múltiplas transições para o mesmo estado e símbolo de entrada. Além disso, foram utilizados conceitos da Gramática BNF (Backus Naur Form), que é um formato de representação de gramática utilizado para definir formalmente as linguagens de programação. Com base nesses conceitos, foram selecionados os métodos e técnicas mais adequados para a implementação do programa em questão.

Após receber a requisição o sistema separa a gramática dos tokens identificando os blocos da string que possuem o caractere "<" que nada mais é do que um padrão utilizada pela BNF (*Backus Naur Form*), formato de representação de gramática, que na aplicação em questão torna mais fácil a diferenciação de conteúdos que chegam até o programa.

A diferenciação de símbolos para cada token e gramática é feita utilizando as funções de match e split para string do JavaScript, linguagem utilizada para desenvolvimento do programa, como podemos observar nos blocos abaixo:

```
token.split("");          grammar.match(/[\a-z]/g);
```

Após a identificação dos símbolos são encontrados os estados alcançados por cada uma das situações utilizando como base um contador e a tabela ASCII, conforme o exemplo abaixo:

```
'<'+String.fromCharCode(65+contador)+'>::='+simbol+String.fromCharCode(65+contador);
```

O exemplo gera um BNF da seguinte forma:

<A>::=eB

A partir deste BNF gerado, cada linha como a acima é tratado como um elemento de um array, e desta forma o array é analisado, observando os estados de origem e estados a serem alcançados, e um objeto é levado no seguinte formato para a aplicação Next.js:

```
{ origem: 'A', simbol: 's', estado: 'S' }
```

Onde é montada uma tabela com cada coluna sendo um símbolo, e cada origem/estado sendo uma linha, gerando a seguinte tabela:

NEXT.js

13

Autômato

```

se
entao
senao
<S> ::= a<A> | e<A> | i<A> | o<A> | u<A>
<A> ::= a<A> | e<A> | i<A> | o<A> | u<A> | ε

```

GERAR ➤

Origem/Estado	s	e	n	t	a	o	i	u
S	A, H	C, M			M	M	M	M
A		B						
B		*B						
C			D					

Para montar a tabela, uma matriz é gerada com o intuito de construir a tabela utilizando componentes da biblioteca MUI/React, o trecho de código a seguir exemplifica a montagem com matriz da tabela:

```
const createTable = (arr: Item[]) => {
  // Troca a posição das propriedades origem e estado em cada objeto de arr
  const newArr = arr.map(item => ({origem: item.estado, simbol: item.simbol, estado: item.origem}));

  // Agrupa os estados e símbolos em uma matriz
  const tableData = newArr.reduce((acc: Record<string, any>, curr) => {
    const { origem, simbol, estado } = curr;
    if (!acc[estado]) {
      acc[estado] = {};
    }
    acc[estado][simbol] = acc[estado][simbol] ? [...acc[estado][simbol], origem] : [origem];
    return acc;
  }, {});

  // Obtém todos os símbolos únicos
  const symbols = Array.from(new Set(newArr.map((item) => item.simbol)));

  // Cria a tabela com as células preenchidas com os estados
  return (
    <Table sx={{display: 'table', marginLeft: '20%', maxWidth: '1024px'}}>
      <TableHead>
        <TableRow>
          <TableCell>Origem/Estado</TableCell>
          {symbols.map((symbol, i) => (
            <TableCell key={i}>{symbol}</TableCell>
          ))}
        </TableRow>
      </TableHead>
      <TableBody>
        {Object.keys(tableData).map((estado, i) => (
          <TableRow key={i}>
            <TableCell>{estado}</TableCell>
            {symbols.map((symbol, j) => (
              <TableCell key={j}>{tableData[estado][symbol]}.join(", ")</TableCell>
            ))}
          </TableRow>
        ))}
      </TableBody>
    </Table>
  );
};
```

Para ocorrer a determinização, baseado no Teorema da Determinização, o sistema realiza a construção de uma nova matriz para geração da tabela, para encontrar o AFD foram identificados todos os estados alcançados pelos símbolos, e por meio de um “*loop while*”, são verificados os estados gerados a partir de cada símbolo, e estes símbolos quando alcançarem mais de um estado por origem, irão formar um novo estado, e este novo estado terá os próximos estados mapeados. Exemplo da implementação no código da parte que de fato realiza determinização abaixo:

```
const dfaTable: DFAItem[] = [{ state: 'S', transitions: {} }];
const visitedStates: Record<string, boolean> = { [dfaTable[0].state]: true };
const symbols = Array.from(new Set(newArr.map(item => item.simbol)));

// Enquanto houver novos estados a serem visitados
let index = 0;
while (index < dfaTable.length) {
  const currentState = dfaTable[index];
  index++;
  for (const symbol of symbols) {
    const nextStates = new Set<string>();
    for (const state of currentState.state.split('')) {
      const transitions = tableData[state]?.[symbol] || [];
      transitions.forEach(t => nextStates.add(t));
    }

    if (nextStates.size === 0) {
      continue;
    }

    const nextState = Array.from(nextStates).sort().join('');
    if (!visitedStates[nextState]) {
      visitedStates[nextState] = true;
      dfaTable.push({ state: nextState, transitions: {} });
    }

    currentState.transitions[symbol] = nextState;
  }
}
```

3. CONCLUSÃO

Ao fim da execução do projeto, por explorar ferramentas como o Next.js e outras tecnologias modernas, com certeza muito conhecimento além do conteúdo relacionado a disciplina foi gerado. Em relação aos Autômatos, a determinização e geração de AFND se fez muito mais clara ao desenvolver, por ser necessário entender cada elemento do processo.

De maneira geral, o desenvolvimento do projeto fez com que além do conteúdo, muito outro conhecimento agregado foi adquirido graças à resolução da proposta feita pelo professor.

4. REFERÊNCIAS BIBLIOGRÁFICAS

SCHEFFEL, Roberto M.. Apostila Linguagens Formais e Autômatos. **Unisul - Universidade do Sul de Santa Catarina**, Unisul - Universidade do Sul de Santa Catarina, v. 1, n. 1, p. 1-77, 03 mar. 2023. Disponível em: https://moodle-academico.uffs.edu.br/pluginfile.php/743888/mod_resource/content/1/APOSTILA-Roberto-Scheffel.pdf. Acesso em: 03 mar. 2023.

VERCEL.COM. **NEXT JS**. Disponível em: <https://nextjs.org/>. Acesso em: 03 mar. 2023.