Autómato Finito Não Determinístico (AFN) - em cada um dos seus estados e perante um símbolo, pode transitar para mais do que um estado. **Autómato Finito Determinístico (AFD) -** em cada um dos seus estados e perante um símbolo, pode transitar para um único estado. Os AFD são mais rápidos (tempo de computação do que os AFN. Os AFD ocupam mais espaço (têm mais estados) do que os AFN.

A = (S, Σ, s0, F, delta) S – conjunto de estados finito não vazio Σ – alfabeto de entrada s0 – estado inicial F – conjunto de estados finais delta – funções de transição (delta(s0,0) = {s0}	**G = (V, Σ, P, S)** V - Símbolos Não Terminais Σ - Símbolos Terminais P - Conjunto de Produções (Regras da Gramática) S - Símbolo Inicial

Conversão de AFN para AFD: 1. Copiar estado inicial; 2. Sempre que aparecerem novos conjuntos, criá-los na tabela; 3. Criar um nome para cada estado; 4. Substituir nomes nas transições; 5. Eliminar nomes antigos

Minimização de AFD: 1. Dividir a tabela em 2 grupos: estados finais e estados não finais; 2. Identificar cada um dos grupos com um símbolo (letra ou algarismo); 3. Para cada um dos grupos com mais de 1 linha, identificar o grupo destino das transições; 4. Subdividir grupos mantendo juntos apenas estados do mesmo grupo, com combinações iguais. (A ausência de transição conta como um grupo); 5. Se os grupos foram alterados, e ainda existem grupos com mais de um estado, voltar ao ponto 2. Se os grupos não foram alterados, ou só existem grupos com um estado, passar ao ponto 6. 6. Nos grupos com mais de um estado, manter um estado alterando todas as transições para os outros estados do grupo para esse estado.

Hierarquia de Chomsky

Tipo 0 - Recursivamente Enumeráveis. Gramáticas livres ou sem restrições, são as mais abrangentes possíveis. Podem gerar linguagens complexas, difíceis de reconhecer. As produções são da forma alfa -> beta na qual tanto alfa como beta são sequências arbitrárias de símbolos terminais e não terminais. O lado esquerdo da produção não pode ser vazio;

Tipo 1 - Dependentes do Contexto. Geram linguagens menos complexas que as anteriores mas ainda difíceis de reconhecer. As produções são da forma alfa A beta->alfa beta gama na qual alfa, beta e gama são sequências arbitrárias de símbolos terminais e não terminais, sendo que gama não é nulo e A é um não terminal singular. Por outras palavras, A pode ser substituído por gama sempre que seja precedido por alfa e sucedido por beta, isto é, num determinado contexto. Estas gramáticas têm que respeitar a condição |alfa A beta|->|alfa beta gama|, existindo uma única exceção a esta regra, a produção inicial pode ser do tipo S->vazio para permitir a palavra vazia;

Tipo 2 - Independentes do Contexto. Geram linguagens mais restritas que os níveis anteriores, mas suficientemente poderosas para serem usadas na definição das linguagens de programação; existem algoritmos bem conhecidos e eficientes para as reconhecer. As produções são da forma A -> alfa na qual alfa é uma sequência arbitrária de símbolos terminais e não terminais, sendo A um símbolo não terminal singular. Tal significa que qualquer ocorrência de A pode ser substituída por alfa independentemente do contexto;

Tipo 3 – Regulares. São equivalentes às expressões regulares utilizadas na especificação dos tokens (é sempre possível construir uma gramática regular a partir de uma expressão regular, gerando exactamente a mesma linguagem, e vice-versa). As produções são da forma A -> a, A -> aB ou A -> vazio na qual A e B são não terminais singulares e a um terminal. Estas são as formas de gramáticas mais restritas em termos de poder de representação.

Módulo de Optimização de um Compilador - Tentativa de melhoramento da representação intermédia de forma a facilitar a produção de melhor código máquina.

Optimizações do código intermédio: Reconhecer e propagar valores constantes, Mover cálculos para locais onde o número de execuções é menor, Reconhecer cálculos redundantes e eliminá-los, Remover código que é redundante ou inalcançável.

Otimização final: certas sequências podem ser substituídas por equivalentes mais eficientes. Incremento em vez de soma, deslocamento (shift) em vez de multiplicação, eliminação de instruções redundantes, inúteis ou inalcançáveis

Cross-compiler - Conceito em que um compilador corre sobre uma máquina e gera código objecto para outra máquina.

Bootstrapping - Uma técnica que consiste na construção de um compilador, usando uma linguagem de implementação, que compile essa própria linguagem.

Exemplos de Erros: Lexicos: regex (break mal escrito); **Semanticos:** data types (só se dá conta em runtime, v.length()); **Sintáticos:** gramáticas (dá-se conta quando se está a escrever o código; faltar um ponto e vírgula)

Tipo de Tradutores:

Assembler – Traduz assembly para linguagem máquina.

Compilador – Traduz o programa fonte no programa em linguagem máquina que depois é executado como um todo (ex: javac, gcc, etc…).

Interpretador – Traduz e executa instrução a instrução até concluir o programa.

Compilação – duas fases (análise e síntese). Análise divide o texto fonte em partes, cria representação intermédia e é independente da máquina. Síntese constrói programa-alvo a partir de código intermédio e é dependente da máquina

Linguagem de Programação – linguagem que permite codificar um algoritmo para realizar determinada tarefa; tem léxico, semântica e sintaxe própria; é uma linguagem artificial

Linguagem-fonte – linguagem de alto nível, que se aproxima da linguagem do programador; necessita de ser traduzida para uma linguagem-máquina (código-alvo) por um compilador/interpretador

Atributos Sintetizados: se na totalidade do conjunto de ações de uma gramática apenas houver atributos que dependam dos atributos dos filhos numa árvore de parse, diz-se que essa gramática (estendida com atributos e ações) é S-attributed e esses atributos dizem-se sintetizados.

Atributos Herdados: se houver dependências entre atributos de símbolos do lado direito das regras (filhos), ou estes dependerem de atributos do não-terminal esquerdo (pai), estes atributos dizem-se herdados. Se, para os atributos herdados, estes dependerem apenas de atributos à sua esquerda ou do pai, diz-se que essa gramática é L-attributed

Ambiguidade - Diz-se que uma gramática é ambígua se existir pelo menos uma frase da linguagem, gerada pela gramática, com mais do que uma árvore de parse. A ambiguidade poderá indicar que uma mesma frase pode ter mais do que um significado. A ambiguidade elimina-se definindo para cada operador uma precedência e uma associatividade

Tipos de Análise Sintática:

Top-Down (Descendente) (Com/Sem Retrocesso (recursivo/preditivo(LL))): a árvore de parse é construída partindo da raiz, até se chegar à sequência de tokens do texto da entrada. Geram sempre uma derivação mais à esquerda e os nós da árvore de parse construída são visitados em pré-ordem (descida recursiva) ou próximo. Método da descida recursiva associa uma rotina a cada não-terminal da gramática que é chamada quando se pretende reescrever esse símbolo. Método da análise preditiva não-recursiva necessita de uma stack auxiliar e é guiada por uma tabela de parse. LL – Left to right parse, Leftmost derivation

Bottom-Up (Ascendente) (Shift-reduce / Shift-reduce com análise de precedência / LR (LR(0), SLR(1), LALR(1), LR(1)): decidem qual a regra gramatical a aplicar tendo visto, na prática, apenas um símbolo do seu lado direito. Neste tipo as regras gramaticais só são aplicadas depois de se ter visto e reconhecido toda a sua parte direita e possivelmente mais o(s) símbolo(s) seguinte(s). Obriga a escolher a operação reduzir ou deslocar, e se existir mais do que uma produção do lado direito igual, tem que se decidir qual escolher. 2 tabelas – ação e goto. LR – left to right parse, rightmost derivation

Tabela de Símbolos e Tipos - As tabelas de símbolos são extensamente utilizadas nos compiladores para armazenarem os nomes de variáveis, tipos, classes, e outras estruturas de programação. Além do nome, cada entrada contém outras informações acerca do objecto nomeado, como seja o seu tipo, endereço (“offset” no segmento de dados estático, ou no registo de activação), número de referências, etc.

- **Ocorrências definidores de identificadores –** inserção na tabela de símbolos
- **Ocorrências aplicadas –** pesquisa na tabela de símbolos (obtenção do tipo…)

Verificação de tipos – o compilador deve verificar se os tipos de dados dos operandos, fornecidos a um operador, e a respetiva definição do operador são legais e compatíveis.

Sistemas de tipos - coleção de regras para associar expressões de tipos às várias partes de um programa

Conversão entre tipos – conversão automática pelo compilador de um tipo de um operando, de modo a torna-lo compatível com o seu contexto (coerção de tipos – soma de int+real, há conversão do int para real antes) ou então o programador pode explicitamente converter o tipo de dados de uma expressão noutra (cast)

Expressões de tipos – qualquer expressão cujo resultado é um tipo: pode ser um tipo básico (boolean, char, integer, real, void), ou o resultado da aplicação de um operador designado por construtor de tipo, a qualquer outra expressão de tipos

– Flex - fast lexical analyzer generator. It is a tool to generate lexical analyzers.
– Bison: is a general-purpose parser generator that converts an annotated context-free
grammar into an LALR(1)
– YACC means: Yet Another Compiler Compiler
– ANTLR: ANother Tool for Language Recognition

Front end: Reconhecer programas válidos, Produzir mensagens de erro, Produzir a representação intermédia, Produzir um mapa de armazenamento preliminar
Back end: Traduzir a representação intermédia em código alvo, Escolher as instruções correspondentes a cada operação definida na representação intermédia, Decidir que informação manter nos registos do processador, Assegurar a concordância com os formatos usados por outros componentes do sistema de desenvolvimento de software

XSD

```
<xs:complexType name="Tpaises">
    <xs:sequence>
        <xs:element name="pais" type="Tpais" minOccurs="0" maxOccurs="32" />
        <xs:unique><xs:selector xpath="Tpais" /><xs:field xpath="sigla" /></xs:unique>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="Tpais">
     <xs:simpleContent>
        <xs:extension base="xs:string">
           <xs:attribute name="nome" type="Tnome" />
           <xs:attribute name="sigla" type="Tsigla" />
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:simpleType name="Tnome">
    <xs:restiction base="xs:string">
        <xs:maxLength value="11" />
        <xs:pattern value="[A-Z][a-zA-Z]{0,10}" />
    </xs:restiction>
</xs:simpleType>
```

XSLT

```
<xsl:for-each select="//dados/equipa">
    <xsl:sort select="pontos"/>
    <xsl:sort select="DG"/>
    <xsl:variable name="total" select="count()"/>
    <xsl:if test="$total <= position()">
        <tr>
            <td><xsl:value-of select="position()" /></td>
            <td><xsl:value-of select="{//paises/pais[@sigla=@sigla]}" /></td>
            <td><xsl:value-of select="pontos" /></td>
        </tr>
    </xsl:if>
</xsl:for-each>
```