FrontEnd vs. BackEnd
– Existência de uma representação intermédia do programa fonte (máquina abstracta)
– A vanguarda mapeia o programa fonte numa representação intermédia
– A retaguarda produz o código alvo(máquina concreta) a partir da representação intermédia
– Simplifica a produção de compiladores para várias máquinas concretas
– Simplifica a produção de compiladores para várias linguagens fonte
– Duas passagens → código mais eficiente que numa única passagem

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

Representação Intermédia:
O programa fonte depois de verificado semanticamente é transformado numa representação intermédia:
• deve ser fácil de produzir
• representar bem todas as características da linguagem fonte
• representar bem as operações disponíveis na máquina alvo
• deve ser fácil de traduzir para instruções máquina (alvo)
Algumas formas de representação intermédia: • Gráficas: Árvores sintácticas abstractas com anotações (os atributos da análise semântica) • Lineares: Sequência de instruções para uma máquina genérica e abstracta. Existem várias formas:
•máquinas de 0 ou 1 endereços (máquinas de stack; p-code)
•máquinas de 2 endereços (próximas de alguns processadores reais)
•máquinas de 3 endereços (de mais alto nível) • Híbridas: Grafos de fluxo entre blocos de instruções

Optimizações do cod. 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
PEEPHOLE: Certas sequências podem ser substituídas por outras equivalentes, – mas mais eficientes – Incremento em vez de soma – deslocamento (shift) em vez de multiplicação – Eliminação de instruções redundantes, inúteis ou
Inalcançáveis (localmente, no código maquina.

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 gama beta|, 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:
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

Apontamentos sobre Schema Definition
Caracteres reservados: |< <| |> >| |& &| |" “| |' ‘| Filho de um complexType tem de ser um dos seguintes:
<xs:sequence> : Uma sequência com uma ordem fixa
<xs:choice> : Uma escolha de um elemento, entre os elementos declarados
<xs:all> : Elementos sem ordem fixa

<xs:element name=" " type="" maxOccurs=”” minOccurs =”” default=”” fixed=”” ref="..." />
Default-> define o valor por omissão.
Fixed-> Força um valor
Ref -> Permite “clonar” outros elementos (têm que ser definidos no 1o nível)
Tipo dados
Exemplos -> xs:integer, xs:string, xs:float, xs:date (ccyy-mm-dd), xs:anyURI, xs:decimal, xs:time (hh:mm:ss.sss), xsd:positiveInteger
xs:ID -> É uma palavra e tem que iniciar com uma letra e é usado para validar um identificador único e usado em atributos ou elementos simples;
xs:IDREF -> é usado para validar atributos ou elementos que no documento existam como xs:ID, se não existir no documento um xs:ID com o mesmo valor é gerado um erro pelo validador XML

Restriçoes: usamos simples type e lá dentro introduzimos todas as restriçoes <xs:simpleType name="percentagem">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="100"/>
  </xs:restriction>
</xs:simpleType>

Exemplos restriçoes
Maior e maior ou igual <xs:**minExclusive** value={}> <xs:**minInclusive** value={}> Menor e menor ou igual <xs:**maxExclusive** value={}> <xs:**maxInclusive** value={}> Tamanho (string, listas) <xs:**length** value={}> <xs:**minlength** value={}> <xs:**maxlength** value={}>
O número máximo de dígitos <xs:**totaldigits** value={}>
O número de casas decimais tem de ser menor que totaldigits <xs:**fractionDigits** value={}>
Expressão regular <xs:**pattern value**={}>
Valores numa enumeração <xs:**enumeration** value={}>
Preservação dos espaços, tabs <xs:**whiteSpace** value="preserve"/>

<xsd:pattern value="[A-Z][A-Z][A-Z]"/>
<xsd:whiteSpace value="collapse"/> (ou replace ou collapse)

Atributos: Nota que os atributos são sempre depois das sequências de elementos
<xs:attribute name="..." type=”” use=””>
use-> required | optional | prohibited
Type-> primitivo ou derivado

<xs:assert> e <xs:assertion> : Permite a validação dos elementos usando o conteúdo do documento XML
<xs:alternative> : Elementos com tipos alternativos condicionais, baseados no valor de um elemento ou atributo
<xs:any>/<xs:anyAttribute>/<xs:openContent> : Altera o mecanismo para inclusão de dados que não estejam definidos no XSD

<!—Exemplo XSD -->
```
<xs:complexType name="TCães">
    <xs:sequence>
      <xs:element name="cão" type="TCão" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TCão">
    <xs:all> <!-- não interessa a ordem. Outras formas são CHOICE ou SEQUENCE -->
      <xs:element name="chip" type="TChip" minOccurs="1" maxOccurs="1"/>
      <xs:element name="raça" type="TRaça" minOccurs="1" maxOccurs="1"/>
      <xs:element name="ativo" type="xs:boolean" minOccurs="1" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
  <xs:simpleType name="TChip">
    <xs:restriction base="integer">
      <xs:pattern value="([0-9]){9}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:unique name="uniqueChip">
    <xs:attribute name="modeloChip" type="string"/>
    <xs:selector xpath="Animais"/>
    <xs:field xpath="@chip"/>
  </xs:unique>
  <xs:simpleType name="TRaça">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Barbado da Terceira"/>
      <xs:enumeration value="Podengo"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="age">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="password">
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
```

Funções
concat(str1,str2,...)
starts-with(str1,str2)
contains(str1,str2)
substring(str,num,comp)
substring-before(str1,str2)
substring-after(str1,str2)
string-length(str)
normalize-space(str)
translate(str1,str2,str3)

A

Funções	D
position()	D d d
last()	D n s d
count(xpath-exp)	D d
id(identificador)	D id

B

Group:
```
<xsd:group name="persongroup">
            <xsd:sequence>
                    <xsd:element name="firstname" type="xsd:string"/>
                    <xsd:element name="lastname" type="xsd:string"/>
                     <xsd:element name="birthday" type="xsd:date"/>
            </xsd:sequence>
</xsd:group>
```

Referenciando...
```
<xsd:element name="person" type="personinfo"/>
<xsd:complexType name="personinfo">
            <xsd:sequence>
                    <xsd:group ref="persongroup"/>
                    <xsd:element name="country" type="xsd:string"/>
            </xsd:sequence>
</xsd:complexType>
```

<!—Exemplo XSLT -->
```
<xsl:for-each select="distritos/distrito[@nome='$nomeDistrito']/cidade">
  <xsl:sort select="nome" order="descending"/> <!-- sort -->
  <xsl:value-of select="."/> <!-- select -->
</xsl:for-each>

<xsl:template match="cidade">
  <xsl:variable name="tamanhoDistrito"> <!-- cria variavel -->
    <xsl:value-of select="../@tamanho"/>
  </xsl:variable>
  <xsl:choose>
    <xsl:when test="@Tamanho &gt; $tamanhoDistrito">
      <xsl:value-of select="@Tipo"/> <!-- attribute -->
    </xsl:when>
    <xsl:otherwise>a</xsl:otherwise>
  </xsl:choose>
  <xsl:if test="position()!=1">,</xsl:if>
</xsl:template>
```

Funções Booleanas: boolean(arg), not(boolexp), true(), false(), lang(str), string(arg);
Fuções Numericas: number(arg), sum(xpath-exp), floor(num), ceiling(num), round(num)
Manip String: (A)
Manip Lista Nós: (B)

Expressões XPath	Resultado
/AAA	Selecciona o nó filho da raiz com nome AAA.
/AAA/CCC	Selecciona nós de nome CCC filhos do nó principal AAA que é filho da raiz.
//BBB	Selecciona todos os nós de nome BBB existentes na árvore documental.
//DDD/BBB	Selecciona todos os nós de nome BBB que sejam filhos de nós DDD posicionados em qualquer ponto da árvore documental.
/AAA/CCC/DDD/*	Selecciona todos os nós filhos de nós DDD que, por sua vez, são filhos de nós CCC que são filhos do nó principal AAA que é filho da raiz.
/*/*/*/BBB	Selecciona todos os nós BBB posicionados no quarto nível da árvore documental.
//*	Selecciona todos os nós do tipo elemento existentes na árvore documental.
/AAA/BBB[1]	Selecciona o primeiro filho com nome BBB do nó AAA que é filho da raiz.
/AAA/BBB[last()]	Selecciona o último filho de nome BBB do nó AAA que é filho da raiz.
//BBB[@ident]	Selecciona todos os nós com nome BBB que tenham um atributo de nome *ident* instanciado.
//BBB[@*]	Selecciona todos os nós com nome BBB que tenham pelo menos um atributo instanciado.
//BBB[not(@*)]	Selecciona todos os nós com nome BBB que não têm nenhum atributo instanciado.
//BBB[@ident='b1']	Selecciona todos os nós com nome BBB que tenham um atributo de nome ident instanciado com o valor b1.
//BBB[normalize-space(@nome)='bbb']	Selecciona todos os nós com nome BBB que tenham um atributo de nome nome cujo valor normalizado (sem espaços no fim e no início e com as sequências de caracteres brancos reduzidas a um espaço em branco) seja igual a bbb.
//*[count(BBB)=2]	Selecciona todos os nós da árvore documental que tenham exactamente dois filhos com nome BBB.
//*[count(*)=2]	Selecciona todos os nós da árvore documental que tenham exactamente dois filhos.
//*[name()= BBB]	Selecciona todos os nós da árvore documental com nome igual a BBB.
//*[starts-with(name(), 'B')]	Selecciona todos os nós da árvore documental cujo nome se inicie por B.
//*[contains(name(),'C')]	Selecciona todos os nós da árvore documental cujo nome contenha o carácter C.
//*[string-length(name())=3]	Selecciona todos os nós da árvore documental cujo nome seja constituído por 3 caracteres.
//*[string-length(name()) < 3]	Selecciona todos os nós da árvore documental cujo nome seja constituído por menos de 3 caracteres.
//BBB | //CCC	Selecciona todos os nós na árvore documental com nome BBB ou CCC.
/descendant::*	Selecciona todos os nós descendentes do nó raiz.
//CCC/descendant::*	Selecciona todos os nós descendentes de nós com nome CCC.
//CCC/descendant::*/DDD	Selecciona todos os nós descendentes de nós com nome CCC e cujo nome seja DDD.