Interativa

Unidade II

COMPILADORES E COMPUTABILIDADE

Prof. Leandro Fernandes

Roteiro

- **Análise sintática ascendente:**
  - **Analisadores LR(1).**
- **Análise semântica:**
  - **Gramática de atributos.**
  - **Tabela de símbolos.**
- **Geração de código:**
  - **Linguagens intermediárias.**
  - **Tradução dirigida pela sintaxe.**
- **Otimizações.**
- ***Assemblers*, ligadores e carregadores.**

Análise sintática ascendente analisadores LR(1)

O nome LR(1) indica que:

- **A cadeia de entrada é examinada da esquerda para a direita (*left-to-right*), isto é, do início para o fim do arquivo.**
- **O analisador procura construir uma derivação direta (*rightmost*) invertida:**
  - **Torna-se invertida para que a entrada possa ser examinada do início para o fim.**
- **Considera-se apenas o 1º símbolo do restante da entrada.**

Análise sintática ascendente analisadores LR(1)

- Decidimos qual regra $A \rightarrow \beta$ deve ser aplicada encontrando os nós vizinhos rotulados com os símbolos de β.
  - A redução para A consiste em acrescentá-lo à árvore como um que agrupe todos os símbolos de β como seus nós filhos.
- Considera duas informações:
  - O estado atual da análise.
  - O símbolo corrente da entrada.
- Uma tabela M codifica as operações a serem realizadas de acordo com o autômato de reconhecimento.

Construção do analisador

- Várias possibilidades precisam ser consideradas em um mesmo momento.
- Um item $A \rightarrow \alpha \bullet \beta$ indica o ponto atual em que se encontra a análise, ou seja:
  - A regra $A \rightarrow \alpha\beta$ foi usada na derivação da cadeia de entrada.
  - Os símbolos terminais derivados de α já foram encontrados.
  - Falta encontrar os símbolos terminais derivados de β.
- Um estado do processo de análise é representado por um conjunto de itens.

Gramática aumentada

Suponha a gramática (dada a esquerda):

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow a$

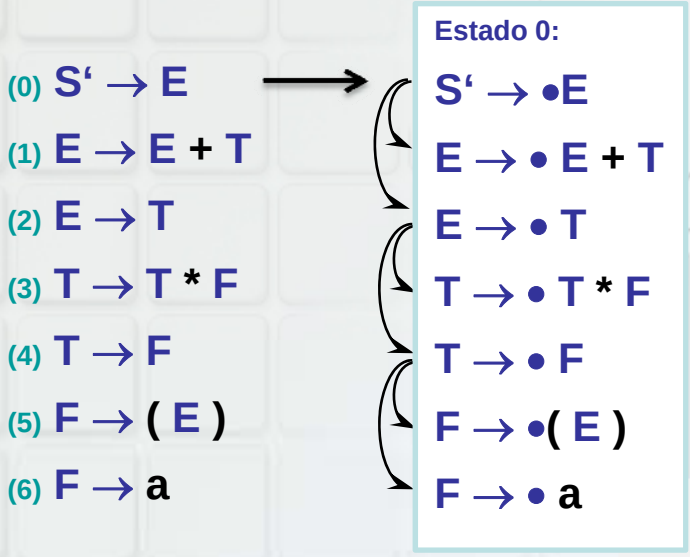
(0) $S' \rightarrow E$
(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow a$

- **Acrescenta-se a nova regra (regra 0) para que seja possível a identificação correta da raiz da árvore sintática, diferenciando-a de outras ocorrências do símbolo inicial.**

UNIP Interativa

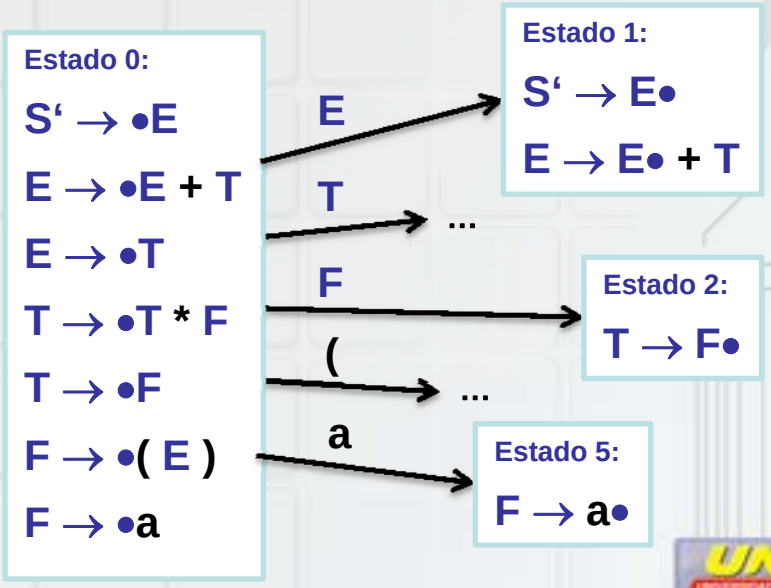
Construção do analisador: definindo os estados

- O estado inicial é dado pelo item formado a partir da regra 0 e contém todos os outros itens associados ao fechamento do estado.



Construção do analisador: definindo os estados

- Os demais estados do autômato são obtidos qual o símbolo esperado para os itens do estado.



Operações do analisador LR(1)

A tabela do analisador define as ações de:

- **Empilhamento (*shift*): ocorrerá quando uma transição com um terminal no estado corrente (topo da pilha) for realizada.**
- **Redução: quando existe um item completo $B \rightarrow \gamma\bullet$ e se o símbolo da entrada pertencer ao Follow(B), é feita uma redução pela regra $B \rightarrow \gamma$.**
  - **Os $|\gamma|$ estados correspondentes a γ devem ser retirados da pilha e o estado $\delta(q,B)$ deve ser empilhado, representando B.**
- **Aceitação: quando ocorre a redução pela regra zero.**
  - **Programa sintaticamente correto!**

Tabela do analisador LR(1)

	E	T	F	(	A	+	*	)	$
0	1	2	3	4	5	-	-	-	-
1	-	-	-	-	-	6	-	-	*r0*
2	-	-	-	-	-	*r2*	7	*r2*	*r2*
3	-	-	-	-	-	*r4*	*r4*	*r4*	*r4*
4	8	2	3	4	5				
5	-	-	-	-	-	*r6*	*r6*	*r6*	*r6*
6	-	9	3	4	5	-	-	-	-
7	-	-	10	4	5	-	-	-	-
8	-	-	-	-	-	6		11	
9	-	-	-	-	-	*r1*	7	*r1*	*r1*
10	-	-	-	-	-	*r3*	*r3*	*r3*	*r3*
11	-	-	-	-	-	*r5*	*r5*	*r5*	*r5*

Analisando a sentença: a+a

Pilha	Entrada	Regra
0	a+a	M[0, a] = 5 Empilha: a
5 0	+a	M[5, +] = *r6* Reduz: F→a
Desempilha o estado 5 (ref. símbolo a), volta para o estado 0 e empilha 3 (ref. símbolo F)		
3 0	+a	M[3, +] = *r4* Reduz: T→F
2 0	+a	M[2, +] = *r1* Reduz: E→T
1 0	+a	M[1, +] = 6 Empilha: +
6 1 0	a	M[6, a] = 5 Empilha: a
5 6 1 0	ε	M[5, $] = *r6* Reduz: F→a
3 6 1 0	ε	M[3, $] = *r4* Reduz: T→F
9 6 1 0	ε	M[9, $] = *r1* Reduz: E→E+T
Desempilha os estados 9, 6 e 1 (ref. símbolos E+T), volta para o estado 0 e empilha 1 (ref. símbolo E)		
1 0	ε	M[1, $] = *r0*

Interatividade

A respeito dos analisadores sintáticos LR(1), não se pode afirmar que:

a) **São analisadores redutores (estilo *shift-reduce* ou empilha-reduz) ascendentes. São eficientes e leem a sentença em análise da esquerda para a direita, produzindo uma derivação mais à direita ao reverso.**

b) **Entre as vantagens, pode-se afirmar que são capazes de reconhecer praticamente todas as estruturas sintáticas definidas por GLC.**

c) **São capazes de descobrir erros sintáticos durante a leitura da sentença em análise.**

d) **O YACC gera analisadores ascendentes.**

e) **Os erros são identificados sempre no momento mais tarde, isto é, na leitura de *tokens*.**

Análise semântica: 3ª etapa do processo de análise

- *Tokens*
- Árvore sintática
- Tabela de símbolos

Tarefas da análise semântica

É responsável por três tarefas:

- **Construir a descrição interna dos tipos e das estruturas de dados definidos no programa do usuário.**
- **Armazenar na tabela símbolos as informações sobre os identificadores (de constante, tipos, variáveis, procedimentos, parâmetros e funções) que são usados no programa.**
- **Verificar o programa quanto a erros semânticos (erros dependentes de contexto) e checagens de tipos com base nas informações contidas na tabela de símbolos.**

O componente semântico

- Verificar a utilização adequada dos identificadores.
  - Análise contextual: declarações prévias de variáveis, escopo de uso etc.
  - Checagem de tipos e compatibilidade.
- Essas tarefas estão além do domínio da sintaxe (Gram. Livres de Contexto - GLC).
  - Aumenta a GLC e completa a definição do que são programas válidos.
- A análise ocorre em dois aspectos:
  - Semântica estática.
  - Semântica de tempo de execução.

O componente semântico: semântica estática

- Conjunto de restrições que determinam se programas sintaticamente corretos são válidos.
- As atividades compreendidas são:
  - A checagem de tipos.
  - A análise de escopo de declarações.
  - A verificação da quantidade e dos tipos dos parâmetros em sub-rotinas.
- Pode ser especificada formalmente por uma gramática de atributos.

O componente semântico: semântica de tempo de execução

- É usada para especificar o que o programa faz, isto é, a relação do programa-fonte (objeto estático) com a sua execução dinâmica.
- Exemplo:

```
L: goto L;
if (i<>0) && (K/I > 10) ...
```

- Importante para a geração de código.
- Geralmente, é especificada de modo informal, mas é possível o uso de formalismos, tais com as gramáticas de atributos (dentre outros).

Gramática de atributos

- **É uma gramática livre de contexto estendida para fornecer sensitividade ao contexto através de atributos ligados a terminais e não terminais.**
  - **Um atributo é qualquer propriedade de uma construção da linguagem.**

(1) D → T L	L.*in* := T.*tipo*
(2) T → int	T.*tipo* := “inteiro”
(3) T → float	T.*tipo* := “real”
(4) L → L_1, id	L_1.*in* := L.*in* incluirTS(id.*token*, L.*in*)
(5) L → id	incluirTS(id.*token*, L.*in*)

Calculando os atributos

- Com base na árvore sintática explícita.
- *Ad hoc* (“comandada” pelo *parser*).
- Podem ser calculados tanto durante a compilação quanto na execução.
- Exemplos:
  - Tipo de dado de uma variável (compilação).
  - Valor de uma expressão (execução, exceto expressões que tratem de constantes).
  - Endereço do início do código objeto de um procedimento (compilação).
  - Declaração de objeto no contexto (compilação, para linguagens que exigem declaração prévia).

Tabela de símbolos

- Armazena as informações sobre todos os identificadores do código fonte:
  - Captura a sensitividade ao contexto e as ações executadas no decorrer do programa.
- Está atrelada a todas as etapas da compilação, sendo a estrutura principal do processo.
- Fundamental para:
  - Realizar a análise semântica.
  - A geração de código.

Operações (inserção e busca) envolvendo a tabela de símbolos

Podem ser implementadas como:

- **Chamadas na gramática de atributos.**
  - **$L \rightarrow L_1$, id** **if (buscaTS(id) == false)**
    incluirTS(id, L.*tipo*)
    else
    ERRO("Já declarado")
- **Diretamente na análise sintática.**
  - **Inserção: quando analisa declarações de variáveis, sub-rotinas, parâmetros.**
  - **Busca: em atribuições, expressões, chamadas de sub-rotinas ou qualquer outro uso de um identificador em um bloco de comandos.**

UNIP Interativa

Interatividade

Analise as mensagens de erro a seguir:

I. Identificador já declarado no escopo atual.

II. Identificador de tipo esperado.

III. Quantidade de parâmetros incompatível com a função.

IV. Função ou variável não definida (lado esquerdo de atribuições).

Quais destes são de natureza semântica?

a) Apenas o item I.

b) Itens I e II.

c) Itens I, III e IV.

d) Itens I, II e IV.

e) Itens I, II, III e IV.

UNIP
Interativa

Geração de código: enfim, a tradução efetivamente!

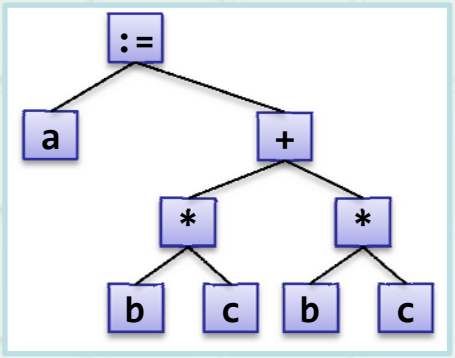
- Corresponde à 1ª etapa do processo de síntese (modelo de análise e síntese).
- Em geral, ocorre em duas fases:
  - Tradução da estrutura construída na análise sintática para um código em linguagem intermediária, usualmente independente do processador.
  - Tradução do código em linguagem intermediária para a linguagem simbólica do processador-alvo.
- Produção do código binário é realizada por outro programa (montador).

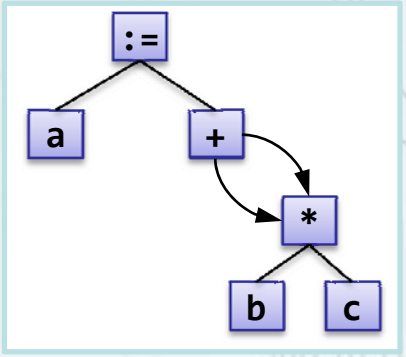
Código intermediário

- Há várias formas de representação de código intermediário, sendo as mais comuns:
  - Árvore e grafo de sintaxe:
    - Notações pós-fixadas e pré-fixadas.
    - Representações linearizadas.
  - Código de três endereços:
    - Quádruplas ou triplas.
  - Instruções *assembler.*
- HIR, MIR e LIR – *High*, *Medium* e *Low Intermediate Representation.*

Código intermediário: árvore e grafo de sintaxe

- A árvore de sintaxe mostra a estrutura hierárquica de um programa fonte.
- O grafo de sintaxe inclui simplificações da árvore de sintaxe.
- Exemplo: a = b * c + b * c





Código intermediário: código de três endereços

- Cada instrução terá, no máximo, três variáveis (dois operandos e o resultado):
  - Formato independente e fácil de traduzir para linguagem simbólica de qualquer processador.
- Expressões complexas devem ser decompostas em várias expressões:
  - Necessitam de variáveis temporárias!
- Exemplos de instruções:
  - A := B op C , A := op B , A := B
  - goto L
  - if A op_rel B goto L

Código intermediário: código de três endereços

- Exemplo: a = b + c * d
- Quádruplas:

	Op	Arg1	Arg2	Res
1	*	c	d	_t1
2	+	b	_t1	a

- Triplas:

	Op	Arg1	Arg2
1	*	c	d
2	+	b	(1)
3	:=	a	(2)

Geração de código: tradução dirigida pela sintaxe

- Construída a partir do mecanismo empregado na verificação de tipos, isto é, uma gramática de atributos.
- Adicionam-se regras que permitam a geração de código intermediário simultaneamente a ações semânticas.

$S \rightarrow$ id := E	geracod(id.valor ":=" E.valor)
$E \rightarrow E_1$ + E_2	E.val = geratemp(); geracod(E.val ":=" E_1.val "+" E_2.val)
$E \rightarrow E_1$ * E_2	E.val = geratemp(); geracod(E.val ":=" E_1.val "*" E_2.val)
$E \rightarrow$ (E_1)	E.val = E_1.val;
$E \rightarrow$ id	E.val = id.val;

Interatividade

Analise as seguintes afirmativas:

I. A geração de código intermediário torna o compilador mais portável, mas a otimização é mais difícil por estar longe do código alvo.

II. O problema de gerar código ótimo é indecidível. Geralmente, são usadas técnicas heurísticas que, na maior parte do tempo, geram bom código.

III. São exemplos de código intermediário as notações pré-fixas, pós-fixas e o código de três endereços.

Pode-se afirmar ser correta a alternativa:

a) Item I.

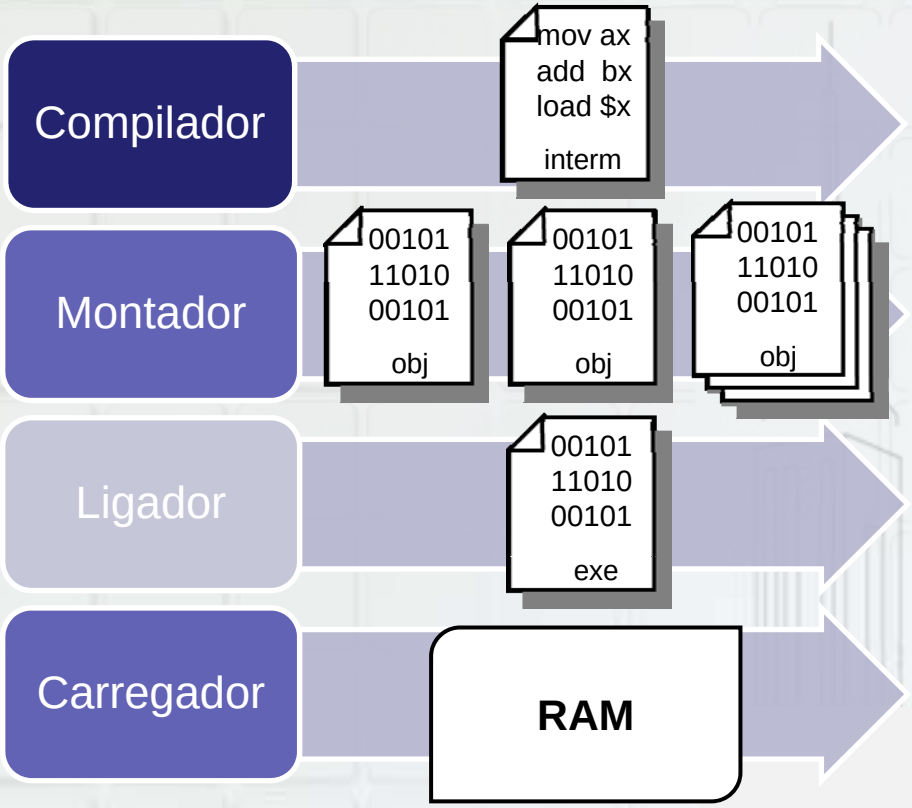
b) Item II.

c) Itens I e II.

d) Itens II e III.

e) Itens I, II e III.

Montadores, ligadores (*linkers*) e carregadores (*loaders*)



Montadores (*assemblers*)

As funções da montagem compreendem:

- **Substituir os mnemônicos pelos *opcodes* do conjunto de instruções do processador.**
- **Determinar de maneira absoluta ou relativa (termos do valor do registrador *Program Counter*) o endereço de destino dos rótulos.**
- **Reservar espaço para dados de acordo com o tipo associado a cada variável.**
- **Gerar constantes em memória para variáveis e constantes, determinando o valor associado ao modo de endereçamento do operando.**

Assemblers (montadores)

Programa em linguagem de alto nível	Programa em linguagem de montagem (*assembly*)			Programa em linguagem de máquina		
	Rótulo	Mnemônico	Oper	End.	Opcod	Oper
int a,b,c;		INPUT	N1	00	12	13
read(a)		INPUT	N2	02	12	14
read(b)		LOAD	N1	04	10	13
c = a + b;		ADD	N2	06	01	14
write(c);		STORE	N3	08	11	15
		OUTPUT	N3	10	13	15
		STOP		12	14	
	N1:	SPACE		13	??	
	N2:	SPACE		14	??	
	N3:	SPACE		15	??	

Formato do arquivo objeto

Cabeçalho	Dado pela identificação de tipo, tamanho do código e, eventualmente, o arquivo de origem.
Código gerado	Contém as instruções e os dados em formato binário.
Relocação	Contém as posições no código em que ocorrerão mudanças quando for definida a posição de carregamento.
Tabela de símbolos	Lista de símbolos globais definidos no módulo e símbolos externos, que devem vir de outros módulos.
Depuração	Contém referências para o código fonte (ex.: número de linha e nomes de identificadores).

Ligadores (*linkers*)

- **Reunir os vários módulos, objetos obtidos da tradução dos vários arquivos fontes em um único programa, o módulo absoluto de carga.**
- **Deve ser capaz de resolver referências cruzadas – endereços dados pelos módulos devem ser atualizados (problema de relocação).**
- **Quando existe um procedimento A que chama a um procedimento B, o endereço absoluto de B só é conhecido após a ligação (problema de referência externa).**

Tarefas do *linker*

- Construir uma tabela com todos os módulos objetos e seus respectivos comprimentos.
- Atribuir um endereço de carga a cada módulo objeto.
- Relocar todas as instruções que contêm um endereço, adicionando uma constante de relocação (endereço inicial de cada módulo).
- Encontrar todas as instruções que referenciam outros procedimentos e inserir nelas o endereço absoluto dos mesmos.

Carregador (*loader*)

- Copiar um programa para a memória principal e preparar sua execução.
- Atividades:
  - Verificar se o programa existe.
  - Avaliar a quantidade de memória necessária e solicitá-la ao SO.
  - Copiar o conteúdo do arquivo (código) para a memória.
  - Ajustar os endereços do código executável de acordo com a posição base de carregamento.
- Tipos de carregadores: absolutos, relocador e dinâmico.

Tipos de carregadores

- **Absoluto: considera que programa é carregado sempre no mesmo endereço.**
- **Relocador: se a carga do programa na posição X da memória, adiciona X a cada uma das referências do programa.**
- **Dinâmico: em situações de *swapping*, pois os processos não necessariamente retornam à mesma posição!**
  - **Executa relocação no momento em que a posição for referenciada.**
  - **Os endereços devem ser relativos ao início do módulo na memória.**

Interatividade

Analise as afirmativas:

I. Os montadores (*assemblers*) realizam a conversão de programas em linguagem de montagem para a linguagem de máquina.

II. Um editor de ligação, ou ligador (*linker*), permite combinar módulos montados separadamente em um único programa.

III. A função principal de um programa carregador (*loader*) é permitir a edição de um programa em linguagem de alto nível.

Está correta a alternativa:

a) Item I.

b) Item III.

c) Itens I e II.

d) Itens II e III.

e) Todos os itens estão corretos.

ATÉ A PRÓXIMA!

UNIP
UNIVERSIDADE PAULISTA
Interativa