Geração de código intermediário

Novembro 2006

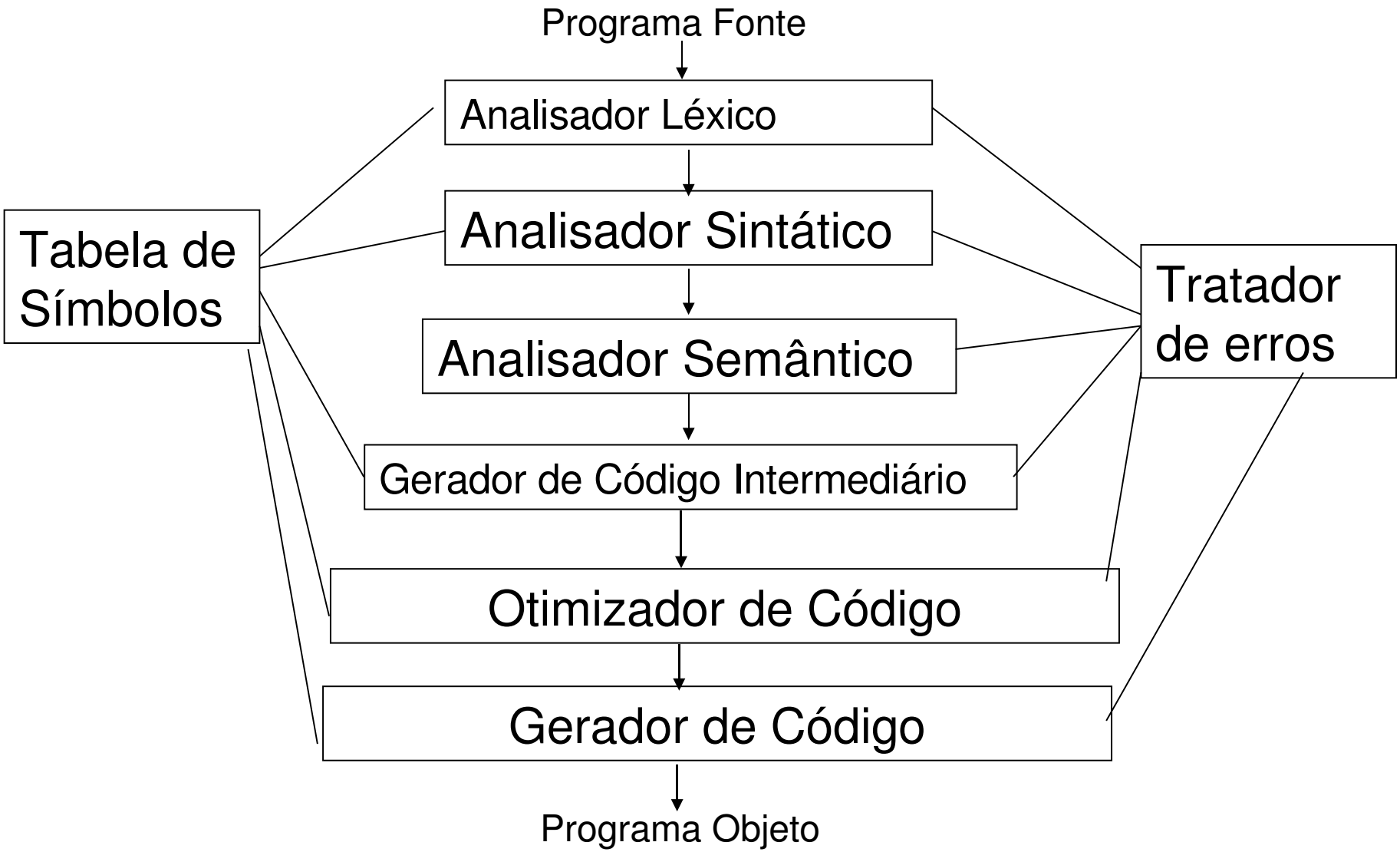
Introdução

- Vamos agora explorar as questões envolvidas na transformação do código fonte em uma possível representação intermediária
- Como vimos, nas ações semânticas já é possível fazer a geração direta de código (assembly por exemplo)

Introdução

- No entanto, a maioria dos compiladores comerciais realizam a geração de uma representação intermediária entre o código fonte e o código de máquina
  - Linguagem intermediária
- Esta representação intermediária é chamada de linguagem intermediária

Fases de um Compilador



Tradução de uma sentença

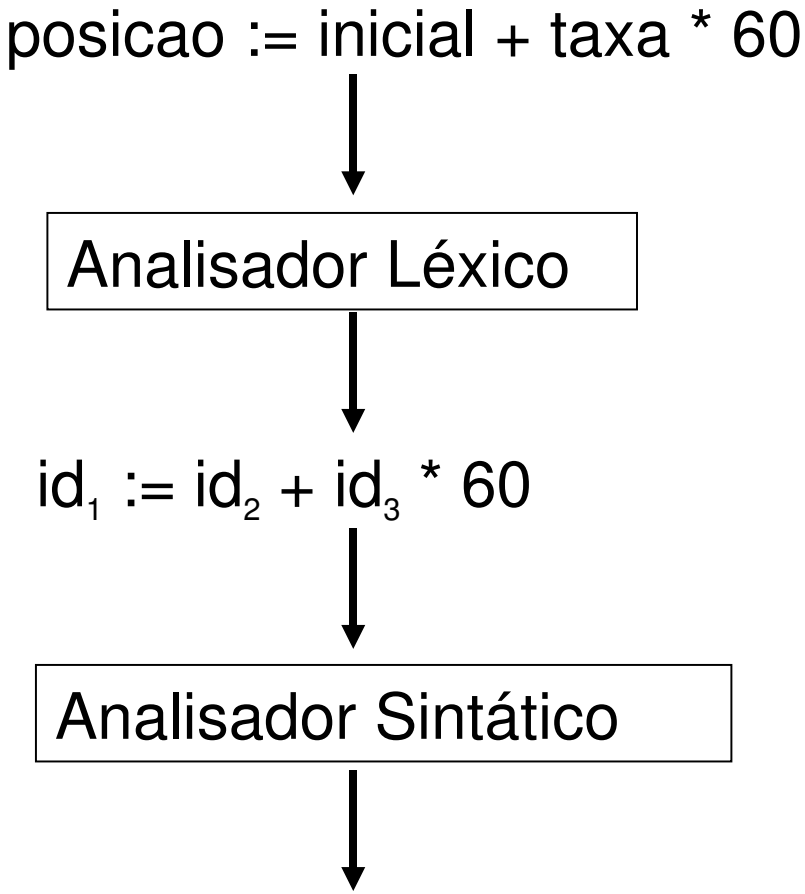
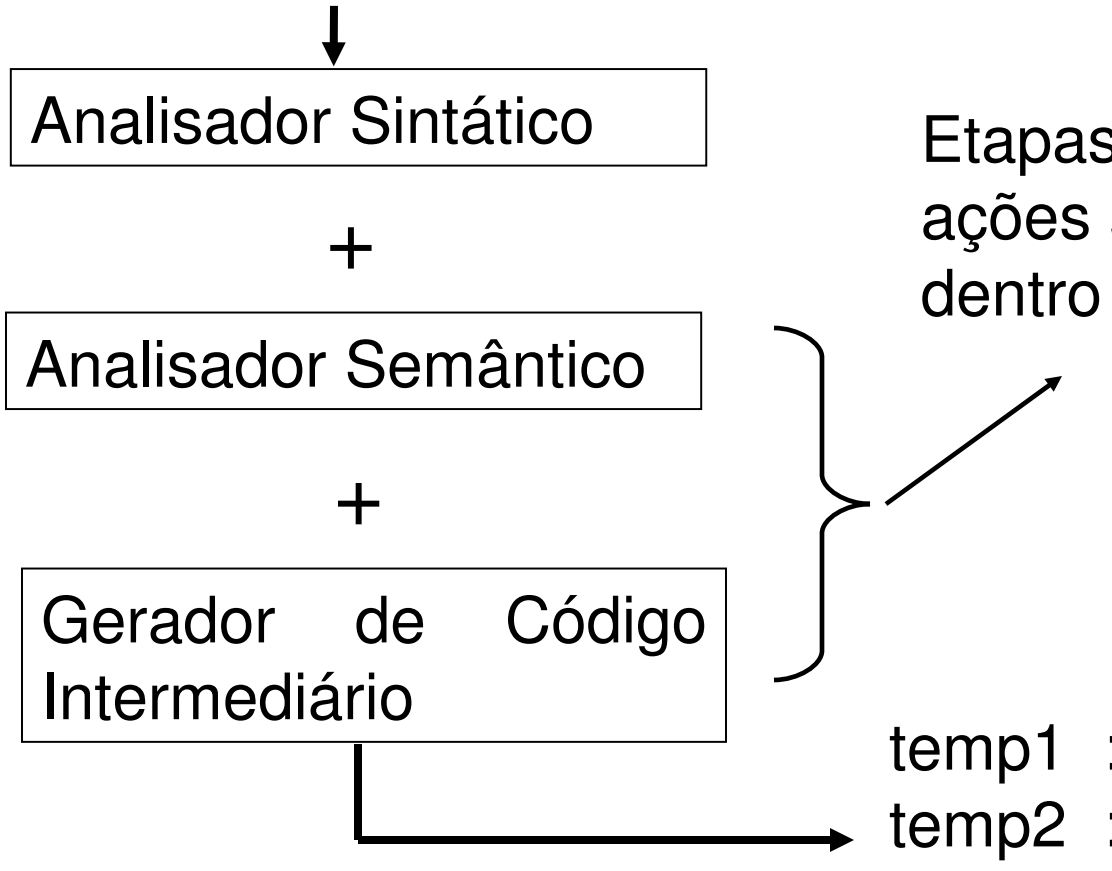


Tabela de Símbolos

1	id_1	position
2	id_2	initial
3	id_3	rate
4		

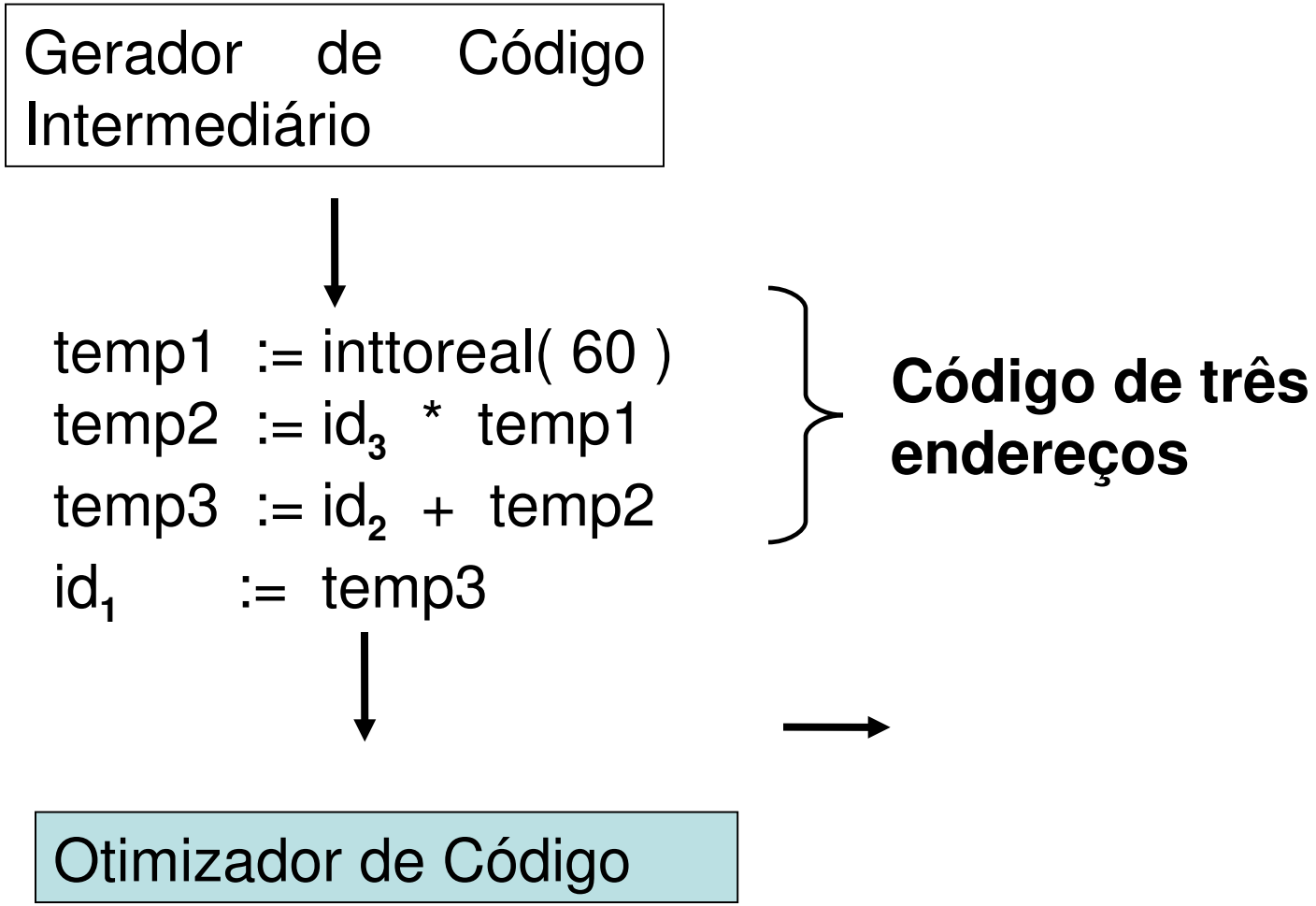
Tradução de uma sentença



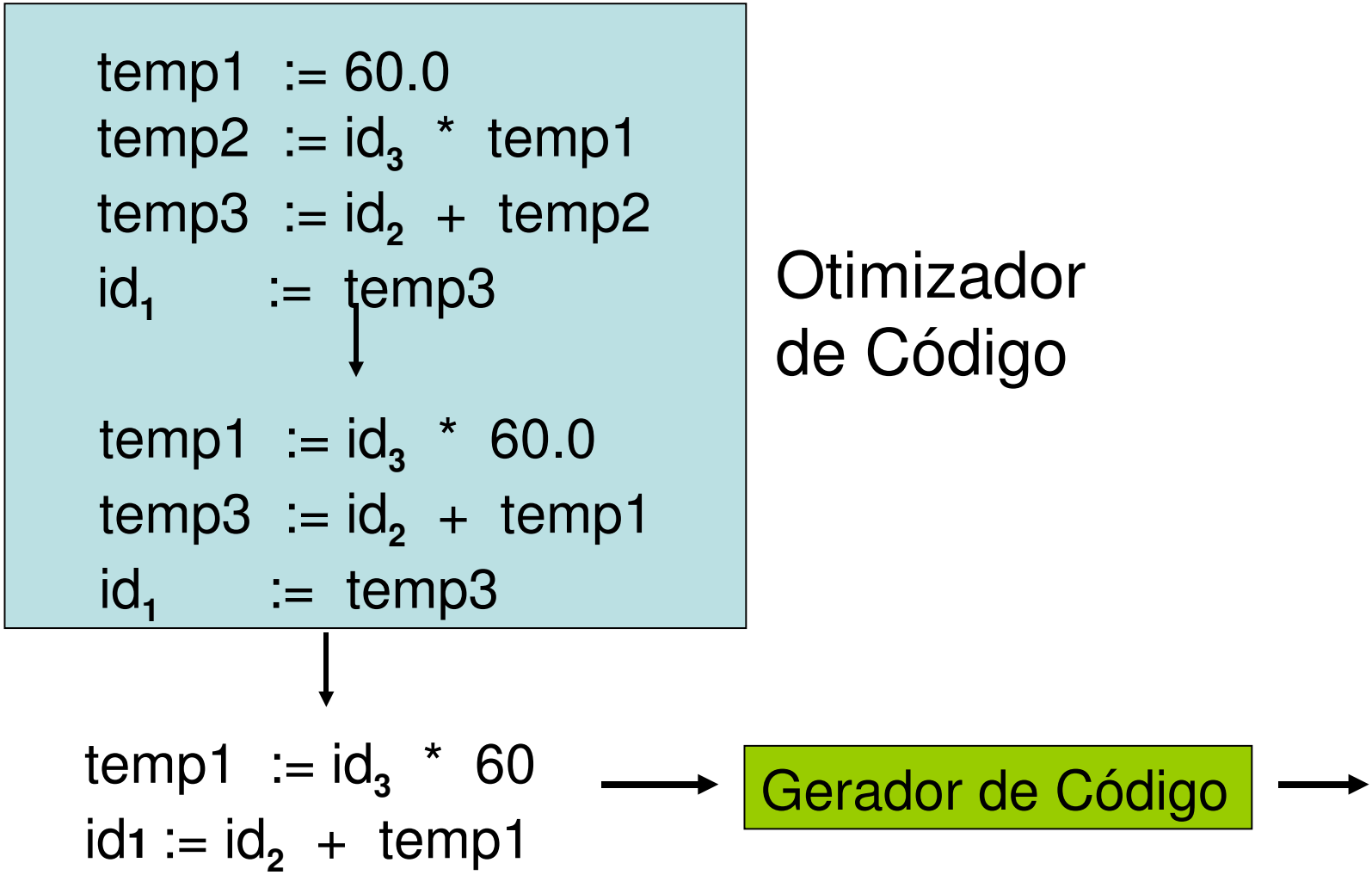
Etapas realizadas através de ações semânticas inseridas dentro do analisador sintático

temp1 := inttoreal(60)
temp2 := id_3 * temp1
temp3 := id_2 + temp2
id_1 := temp3

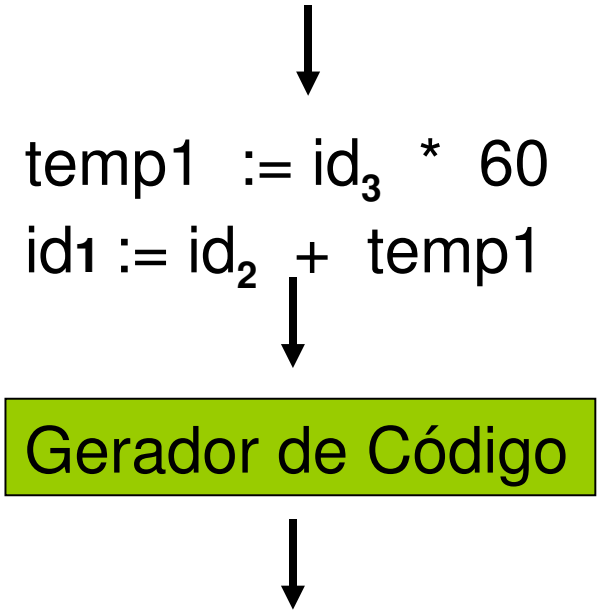
Tradução de uma sentença



Tradução de uma Sentença



Tradução de uma sentença



Linguagem Intermediária

- Como observamos, o objetivo de se realizar a geração de código intermediário é permitir a realização de otimizações.
- Qual seria um outro uso de linguagens intermediárias atualmente?
  - Java!

Linguagem Intermediária

- Existe uma variedade de linguagens intermediárias atualmente
- Quando se está projetando um compilador, certamente a decisão de qual linguagem intermediária utilizar é crucial

Linguagem Intermediária

- Dependendo da linguagem intermediária, otimizações de código poderão:
  - ser mais facilmente serem implementadas
  - ser mais difíceis de serem implementadas
  - demandar mais tempo de computação se as linguagens intermediárias forem inapropriadas

Linguagem Intermediária

- Outro ponto importante é observarmos quão independente de máquina a linguagem intermediária é
  - Quanto mais dependente de máquina, será mais difícil de se implementar certas otimizações
- O que muitos compiladores fazem é a utilização de várias representações intermediárias

Linguagem Intermediária

- Durante o processo de compilação, inicialmente gera-se uma primeira representação intermediária, na qual são aplicadas otimizações.
- Em seguida, esta representação otimizada é convertida em outra representação intermediária, conveniente para a aplicação de outras otimizações

Linguagem Intermediária

- Um exemplo de compilador que realiza tal estratégia é o compilador da Hewlett-Packard para PA-RISC
- As linguagens intermediárias podem ser classificadas em:
  - Linguagens intermediárias de alto nível
    - Próximas da linguagem fonte
  - Linguagens intermediárias de nível médio
  - Linguagens intermediárias de baixo nível
    - Próximas da linguagem alvo

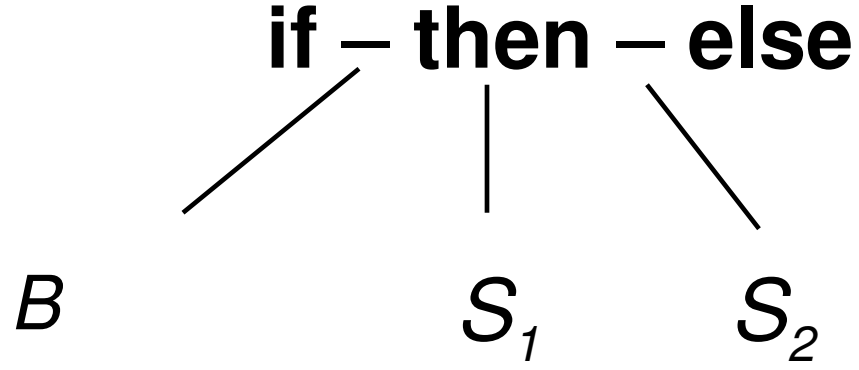
Linguagem Intermediária

- Um compilador pode gerar mais de uma destas representações durante o processo de compilação
- Uma possível forma de representação intermediária é a árvore sintática

Árvores Sintáticas

- Uma **árvore sintática** é uma forma condensada de árvore gramatical, útil para a representação das construções de linguagem
- A produção S ::= **if** *B* **then** S_1 **else** S_2 poderia aparecer numa árvore sintática como:

Árvores Sintáticas



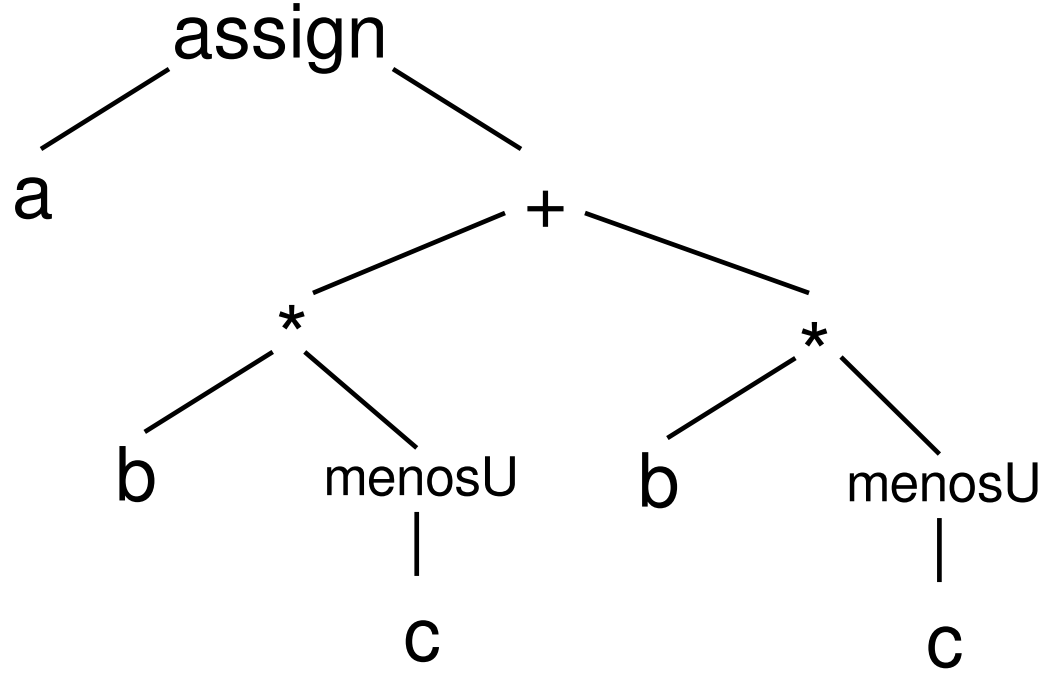
- Numa árvore sintática os operadores e as palavras chaves não figuram como folhas, ao contrário, são associados a um nó interior

Árvores Sintáticas

- Exemplo: Para a expressão

  a := b * -c + b*-c

  teremos a seguinte árvore sintática



Árvores Sintáticas

- Árvores sintáticas permitem que a geração de código seja desacoplada da análise sintática
- A geração das árvores sintáticas é feita através de ações semânticas
- Exemplo: para um comando de atribuição poderíamos ter a gramática com as seguintes ações semânticas:

Árvores Sintáticas

	Regra Semântica
S -> **id** := E	S.nptr := criar_no('assign', criar_folha(**id**, **id**.local), E.nptr)
E -> E_1 + E_2 |	E.nptr := criar_nó('+', E_1.nptr, E_2.nptr)
E_1 * E_2 |	E.nptr := criar_nó('*', E_1.nptr, E_2.nptr)
-E_1 |	E.nptr := criar_nó('menosU', E_1.nptr)
(E_1) |	E.nptr := E_1.nptr
id	E.nptr := criar_folha(**id**, **id**.local

Grafos Dirigidos Acíclicos

- Um Grafo Dirigido Acíclico GDA também é uma representação intermediária de um compilador
  - Um (GDA) para uma expressão identifica as subexpressões comuns existentes na mesma
  - Como uma árvore sintática, um GDA possui um nó para cada subexpressão de uma expressão
  - Um nó interior representa um operador e os filhos representam seus operandos

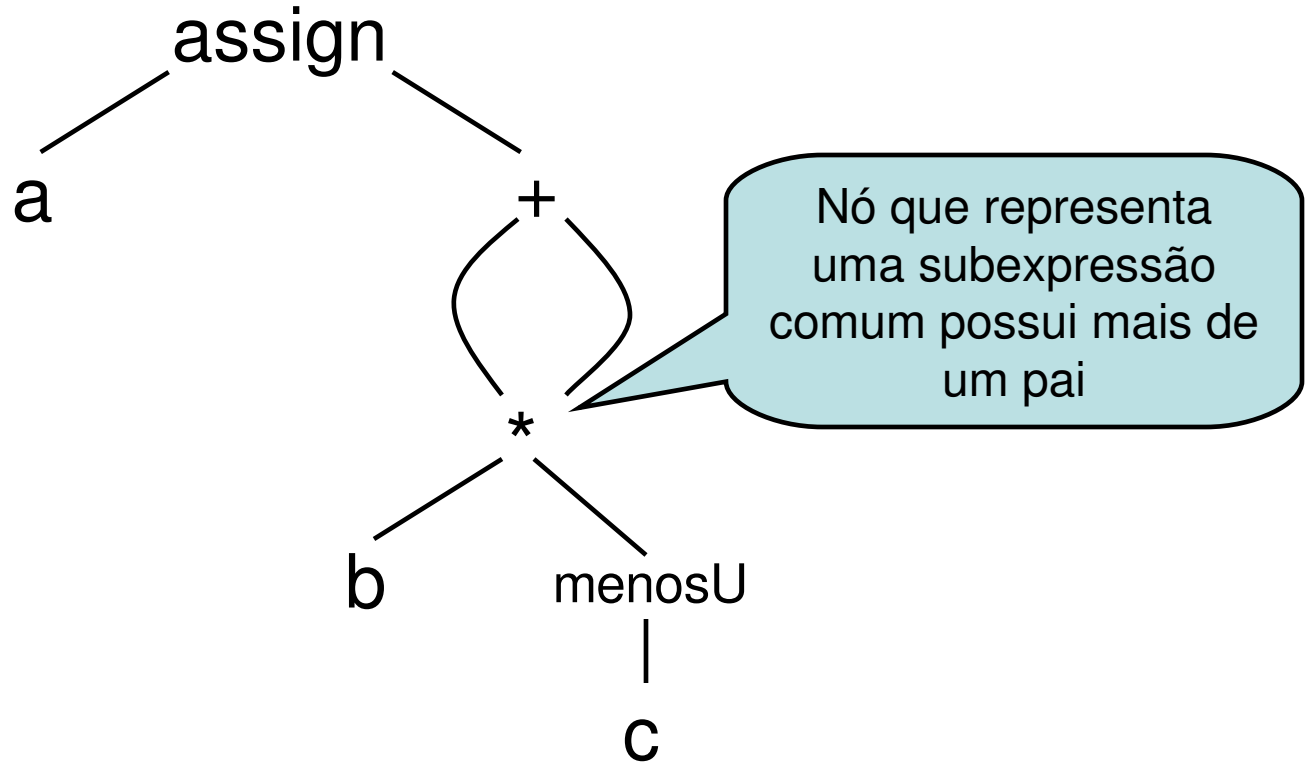
Grafos Dirigidos Acíclicos

- A diferença entre um GDA e uma árvore sintática (AS) é que, um nó de um GDA representando uma subexpressão comum, possui mais de um pai
  - O GDA fornece as mesmas informações da AS, porém de maneira mais compacta
- Logo, no exemplo visto anteriormente teríamos:

Grafos Dirigidos Acíclicos

- Exemplo: Para a expressão

  a := b * -c + b*-c teremos o GDA



Grafos Dirigidos Acíclicos

- As regras semânticas podem criar os GDA ´s
- As mesmas regras vistas anteriormente poderiam ser usadas
- A função criar_nó (op, esquerdo, direito) deverá verificar, antes de criar um novo nó, se já não existe um nó com raiz *op* e filhos *direito* e *esquerdo*

Grafos Dirigidos Acíclicos

- Exercício

Monte o GDA para a seguinte expressão:

(a + b) * a + a + (a + b) * c

Código de Três endereços

- É composto por uma seqüência de instruções envolvendo operações binárias ou unárias e uma atribuição
  - O nome ‘três endereços’ está associado à especificação, em uma instrução, de no máximo três operandos: dois para os operadores binários e uma para o resultado
- Ex: t1 = -c

  t2 := b * t1

  c := t2

Código de Três Endereços

- O código de três endereços é uma seqüência de enunciados da forma geral

  $x := y \; op \; z$

- Onde, *x*, *y* e *z* são nomes, constantes ou objetos de dados temporários criados pelo compilador
- *op* está no lugar de qualquer operador

Código de Três Endereços

- Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução.
  - Dessa forma, obtém-se um código mais próximo da estrutura da linguagem *assembly*

Código de Três Endereços

- No código de três endereços não são permitidas expressões aritméticas construídas, já que só há um operador
- Logo, uma expressão da forma x + y * z poderia ser traduzida na seqüência

  $t_1 := y * z$

  $t2 := x + t_1$

- Os temporários são construídos para os nós interiores da árvore sintática

Código de Três Endereços

- Os enunciados de três endereços são semelhantes ao código de montagem (linguagem assembly)

Código de Três Endereços

- Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções:
  - Expressões com atribuição
  - Acesso indexado e indireto
  - Desvios
  - Invocação de rotinas

Instruções de atribuição

- São aquelas nas quais o resultado de uma operação é armazenado na variável especificada à esquerda do operador de atribuição, aqui denotado por :=
- Há três formas para esse tipo de instrução:
  - Na primeira, a variável recebe o resultado de uma operação binária

    x := y op z

Instruções de atribuição

- Na segunda, o resultado pode ser obtido a partir da aplicação de um operador unário:

  x := op y;

- Na terceira forma, pode ocorrer uma simples cópia de valores de uma variável para outra:

  x := y

Instruções de atribuição - Exemplo

- Por exemplo, a instrução em linguagem de alto nível:

  a = b + c * d;

  seria traduzida nesse formato para as instruções:

  _t1 := c * d

  a := b + _t1

Acesso Indexado e Indireto

- Para atribuições indexadas, as duas formas básicas são

  x := y[i]

  x[i] := y
- A forma x := y[i] atribui à x o valor amazenado i posições de memória a partir do endereço de memória y
- A forma x[i] := y atribui o valor y à posição de memória com deslocamento i partir do endereço de memória x

Instruções de Desvio

- As **instruções de desvio** podem assumir duas formas básicas.
- Uma instrução de desvio incondicional tem o formato:

  goto L

  onde L é um rótulo simbólico que identifica uma linha do código.

Instruções de Desvio

- A outra forma de desvio é o desvio condicional, com o formato:

  if x opr y goto L

  onde:

  – opr é um operador relacional de comparação (<, <=, ... etc)

  – L é o rótulo da linha que deve ser executada se o resultado da aplicação do operador relacional for verdadeiro;

  – caso contrário, a linha seguinte é executada.

Instruções de Desvio

- Por exemplo, a seguinte iteração em C

```
while (i++ <= k)
    x[i] = 0;
x[0] = 0;
```

Instruções de Desvio

Poderia ser traduzida em:

```
_L1:  if i > k goto _L2
          i := i + 1
          x[i] := 0
          goto _L1

_L2: x[0] := 0
```

Otimização de Código

- Várias são as otimizações realizadas por um compilador:
  - Alocação de registradores: Tenta evitar a gravação/leitura de valores de/para a memória principal fazendo que os mesmos residam em registradores
  - Propagação de constantes: Propaga sempre que possível constantes em expressões

Invocação de rotinas

- A **invocação de rotinas** ocorre em duas etapas:
  - Inicialmente, os argumentos do procedimento são ‘registrados’ com a instrução param.
  - Após a definição dos argumentos, a instrução call completa a invocação da rotina

Invocação de rotinas

- A instrução return indica o fim de execução de uma rotina.
  - Opcionalmente, esta instrução pode especificar um valor de retorno, que pode ser atribuído na linguagem intermediária a uma variável como resultado de call

Invocação de rotina - Exemplo

- Por exemplo, considere a chamada de uma função f que recebe três argumentos e retorna um valor:

```
f(a, b, c);
```

Invocação de rotina - Exemplo

- Neste exemplo em C, esse valor de retorno não é utilizado. De qualquer modo, a expressão acima seria traduzida para

  param a

  param b

  param c

  _t1 := call f,3

Invocação de rotina - Exemplo

- Onde o número após a vírgula indica o número de argumentos utilizados pelo procedimento f.
- Com o uso desse argumento adicional é possível expressar sem dificuldades as chamadas aninhadas de procedimentos.

Atribuições de ponteiro e endereço

- As instruções em formato intermediário também utilizam um formato próximo àquele da linguagem C:

  x := &y

  w := *x

  *x := z

Implementação de código de Três Endereços

- A representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em:
  - Tabelas com quatro colunas(quadrúplas)
  - Tabelas com três colunas (triplas)

Quádruplas

- Na abordagem que utiliza **quádruplas**, cada instrução é representada por uma linha na tabela com a especificação do:
  - Operador
  - Primeiro argumento
  - Segundo argumento
  - e do resultado.

Quádruplas

- Por exemplo, a tradução da expressão

  a = b + c * d;

  resultaria no seguinte trecho da tabela:

	operador	arg_1	arg_2	resultado
1	*	c	d	_t1
2	+	b	_t1	a

Quádruplas

- Para algumas instruções, como aquelas envolvendo operadores unários ou desvio incondicional, algumas das colunas estariam vazias.

Triplas

- Na outra forma de representação, por **triplas**, evita-se a necessidade de manter nomes de variáveis temporárias ao fazer referência às linhas da própria tabela no lugar dos argumentos.

	operador	arg_1	arg_2	resultado
1	*	c	d	**_t1**
2	+	b	**_t1**	a

Triplas

- Nesse caso, apenas três colunas são necessárias, uma vez que o resultado está sempre implicitamente associado à linha da tabela.

Triplas

- No mesmo exemplo apresentado para a representação interna por quádruplas, a representação por triplas seria:

	operador	arg_1	arg_2
1	*	c	d
2	+	b	(1)