Universidade de Santa Cruz do Sul – UNISC
Departamento de informática

COMPILADORES

Síntese

Prof. Geovane Griesang
geovanegriesang@unisc.br

Síntese

Data	Conteúdo
18/11/2013	Análise sintática – Parte 01
25/11/2013	Análise sintática – Parte 02
02/12/2013	Síntese
09/12/2013	Prova 03
16/12/2013	EXAME

Sumário

Síntese

Geração de código intermediário

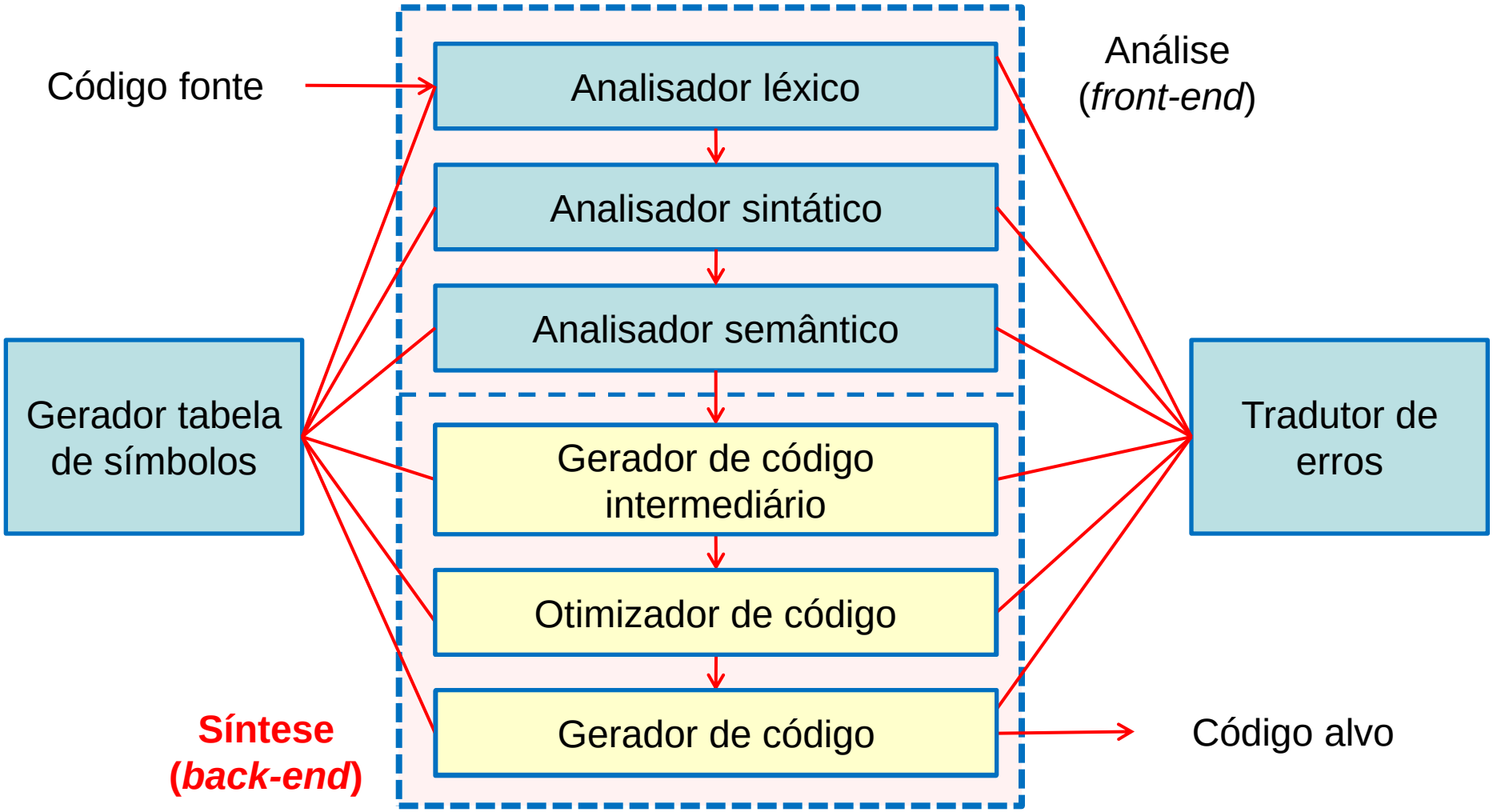
GAD: Grafos Acíclicos Dirigido

Código de três endereços

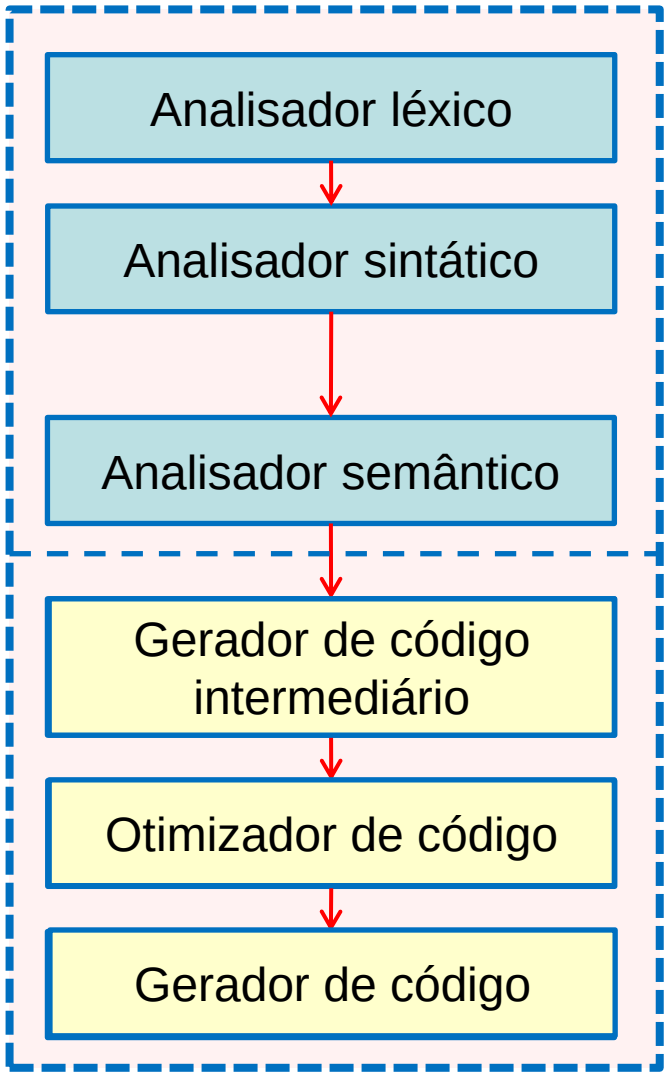
Otimizador de código

Gerador de código

Síntese


Código fonte
Analisador léxico
Analisador sintático
Analisador semântico
Gerador tabela de símbolos
Gerador de código intermediário
Otimizador de código
Gerador de código
Tradutor de erros
Código alvo
Análise (*front-end*)
Síntese (*back-end*)

Análise semântica



Gera um código mais fácil de traduzir.

Melhora o código intermediário para gerar um código de máquina mais rápido.

Instruções intermediárias são traduzidas em uma sequência de instruções de máquina

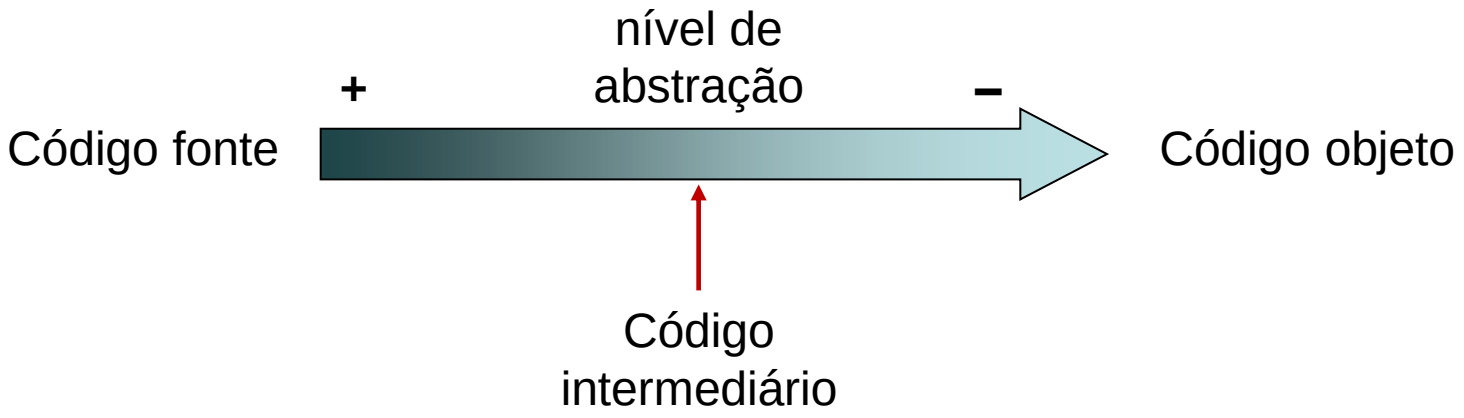
Síntese

A partir da árvore de derivação pode ser gerado o código objeto final

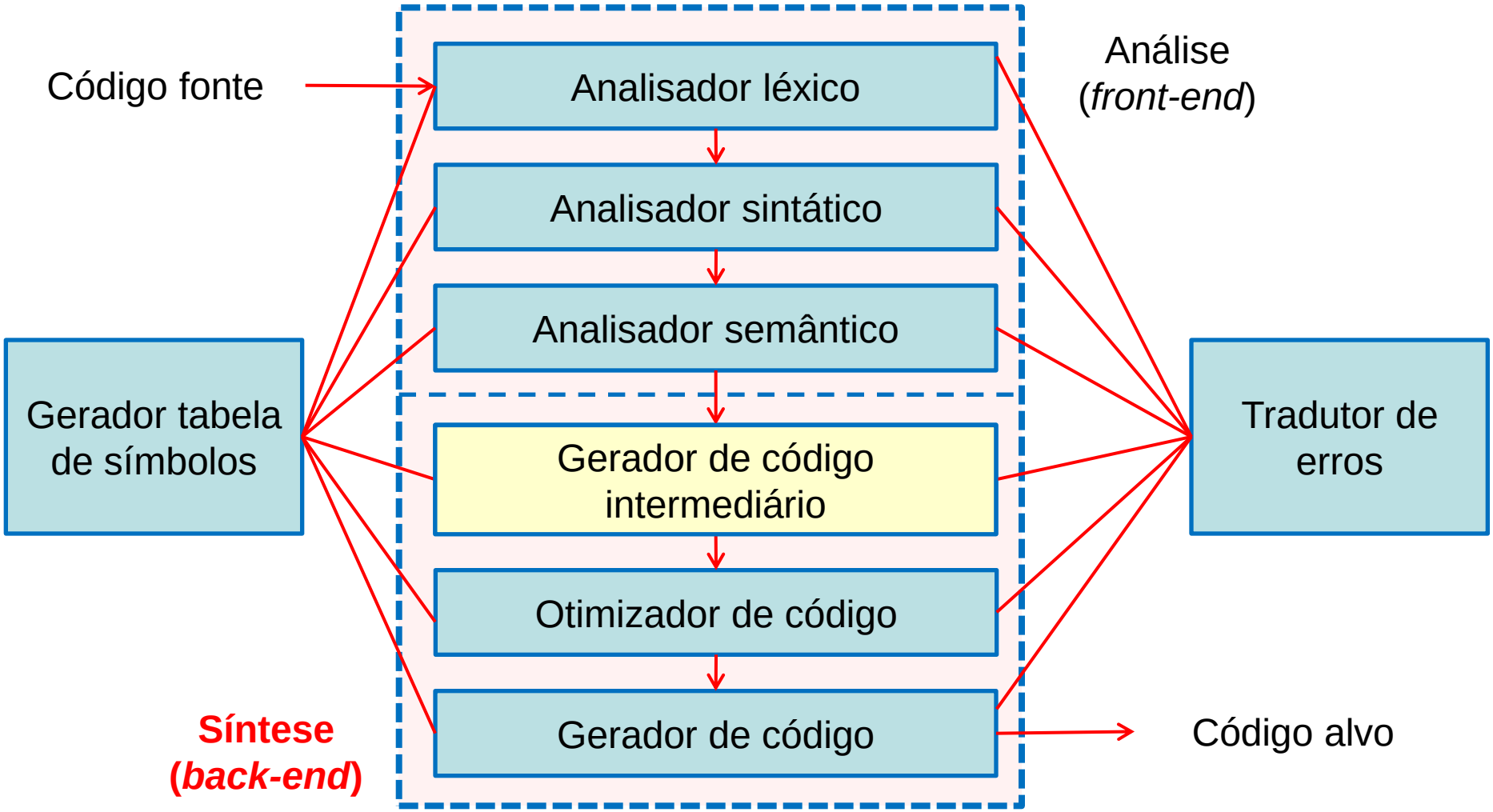
- Resolução de muita abstração em um passo
- Complexidade

Normalmente é feito um passo intermediário

- Geração de uma representação intermediária do código



Síntese


Código fonte
Analisador léxico
Analisador sintático
Analisador semântico
Gerador de código intermediário
Otimizador de código
Gerador de código
Análise (*front-end*)
Gerador tabela de símbolos
Tradutor de erros
Síntese (*back-end*)
Código alvo

Síntese

Geração de código intermediário

No modelo de análise e síntese de um compilador, o *front-end* analisa um programa-fonte e cria uma representação intermediária, a partir da qual o *back-end* gera o código objeto.

O ideal é que os detalhes da linguem fonte sejam confinados no *front-end* e os da máquina alvo fiquem no *back-end*.

Síntese

Geração de código intermediário

Com uma representação intermediária definida de forma adequada, um compilador para a linguagem i e a máquina j pode então ser construído, combinando o *front-end* para linguagem i com o *back-end* para máquina j.

Essa abordagem para criação de um conjunto de compiladores pode economizar muito esforço: *m x n* compiladores podem ser construídos escrevendo-se apenas *m front-ends* e *n back-ends*.

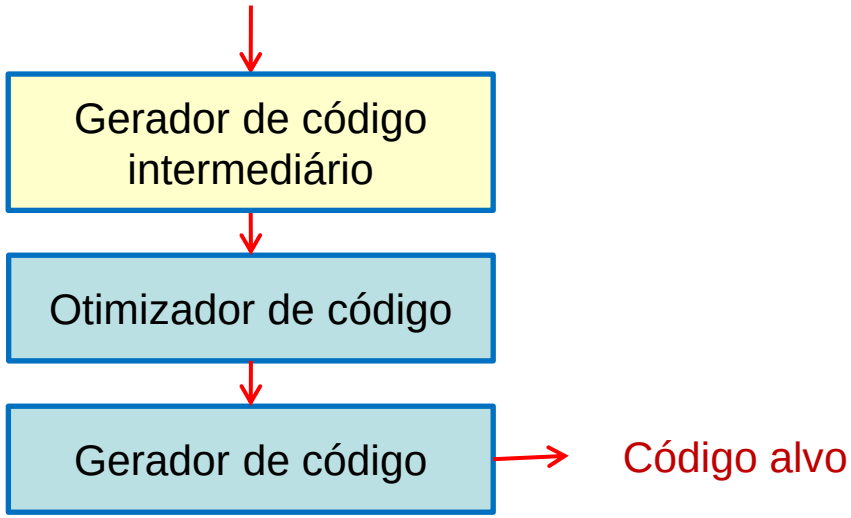
Síntese

Geração de código intermediário

Esse gerador de código intermediário usa estruturas produzidas pelo analisador sintático e verificadas pelo analisador semântico p/ criar uma sequência de instruções simples, denominada código intermediário.

- Está entre a linguagem de alto nível e a linguagem de baixo nível.

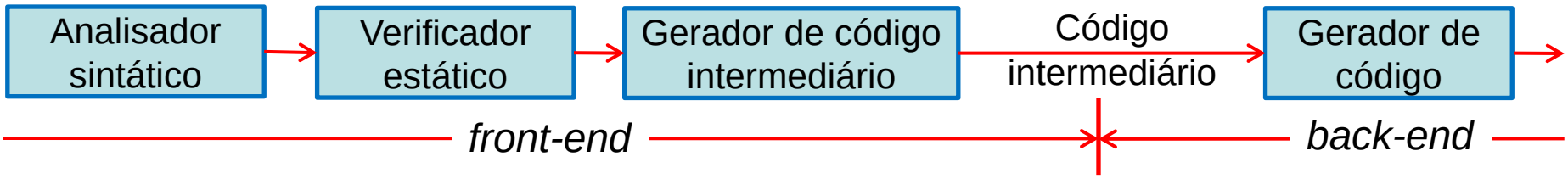
O código intermediário é o que mais se aproxima do programa executável, porém, ele passa por mais etapas até virar o executável final.



Síntese

Geração de código intermediário

Estrutura lógica de um *front-end* de um compilador:



A verificação estática inclui a verificação de tipo, capaz de garantir que os operadores são aplicados a operadores compatíveis. Além disso, também inclui verificações sintáticas que restarem após a análise sintática.

Por exemplo: a verificação estática garante que um comando *break* em C esteja incorporado em um comando *while*, *for* ou *switch*;
se não houver um dessas instruções envolventes, um erro é informado.

Síntese

Os nós de uma árvore de sintaxe representam construções do programa fonte, onde os filhos de um nó representam os componentes significativos de uma construção.

Um Grafo Acíclico Dirigido (DAG – *Directed Acycle Graph*) para uma expressão identifica a *subexpressão comum* (subexpressão que ocorrem mais de uma vez) da expressão.

Portanto, os DAGs podem ser construídos usando as mesmas técnicas de árvores de sintaxe.

Síntese

A diferença entre eles é que um nó N de um DAG tem mais de um pai se N representar uma subexpressão comum.

Numa árvore de sintaxe, a árvore p/ subexpressão comum seria replicado tantas vezes quantas ocorresse a subexpressão na expressão original.

Portanto, uma DAG não só representa as expressões mais sucintamente, como também fornece ao compilador dicas importantes em relação à geração do código eficiente para avaliar as expressões.

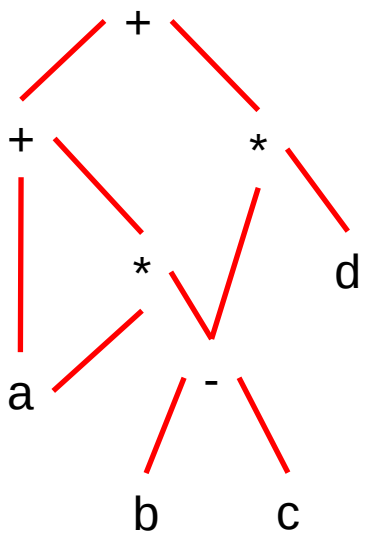
Síntese

Ex.: a figura mostra um DAG para a expressão: a + a * (b - c) + (b - c) * d

A folha p/ a possui dois pais, porque a aparece duas vezes na expressão.

As duas ocorrências da sub expressão comum b-c são representadas por um único nó, o nó rotulado com o operador "-".

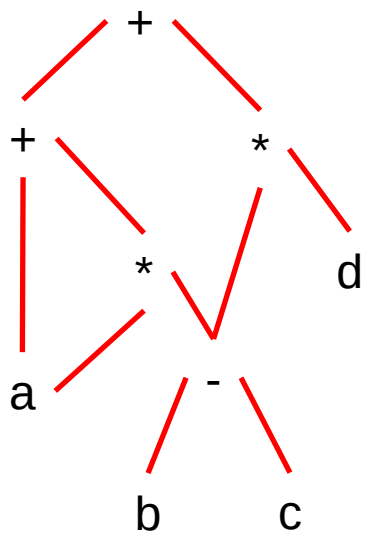
Esse nó tem dois pais, representando seus dois usos nas expressões a * (b - c) e (b - c) * d.



Síntese

Ex.: a figura mostra um DAG para a expressão: a + a * (b - c) + (b - c) * d

Mesmo que aparecem duas vezes na expressão completa, cada um possui só um nó pai, uma vez que os dois usos estão relacionados a subexpressão comum b-c.



Síntese

Geração de código intermediário

Grafos acíclicos dirigido (para expressões)

A tabela abaixo pode ser usada para construir árvores de sintaxe ou DAG.

Produção	Regras semânticas
1) $E \rightarrow E_1 + T$	E.*node* = **new** *Node*('+', E_1.*node*, T.*node*)
2) $E \rightarrow E_1 - T$	E.*node* = **new** *Node*('-', E_1.*node*, T.*node*)
3) $E \rightarrow T$	T.*node*
4) $T \rightarrow (E)$	E.*node*
5) $T \rightarrow$ **id**	T.*node* = **new** *Leaf*(***id***, **id**.*entry*)
6) $T \rightarrow$ **num**	T.*node* = **new** *Leaf*(**num**, **num**.*val*)

Uma DAG será construída se, antes de criar um novo nó, as funções *Leaf* e *Node* primeiro verificarem se um nó idêntico já existe.

Síntese

Geração de código intermediário

GAD

Produção	Regras semânticas
1) E → E_1 + T	E.*node* = **new** *Node*(‘+’, E_1.*node*, T.*node*)
2) E → E_1 – T	E.*node* = **new** *Node*(‘-’, E_1.*node*, T.*node*)
3) E → T	T.*node*
4) T → (E)	E.*node*
5) T → **id**	T.*node* = **new** *Leaf*(**id**, **id**.*entry*)
6) T → **num**	T.*node* = **new** *Leaf*(**num**, **num**.*val*)

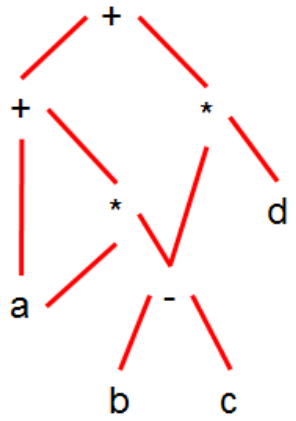
Se um nó idêntico já foi previamente criado, o nó existente é retornado.

Por exemplo, antes de construir um novo nó, *Node(op, Left, Right)* verifica se já existe um nó com o rótulo *op*, e os filhos *left* e *right*, nessa ordem.

Se houver, *Node* retorna o nó existente.

Caso contrário, cria um novo nó.

Síntese



Passos para construção da DAG
01) p_1 = Leaf(**id**, entry-a)
02) p_2 = Leaf(**id**, entry-a) = p_1
03) p_3 = Leaf(**id**, entry-b)
04) p_4 = Leaf(**id**, entry-c)
05) p_5 = Node('-', p_3, p_4)
06) p_6 = Node('*', p_1, p_5)
07) p_7 = Node('+', p_1, p_6)
08) p_8 = Leaf(**id**, entry-b) = p_3
09) p_9 = Leaf(**id**, entry-c) = p_4
10) p_{10} = Node('-', p_3, p_4) = p_5
11) p_{11} = Leaf(**id**, entry-d)
12) p_{12} = Node('*', p_5, p_{11})
13) p_{13} = Node('+', p_7, p_{12})

a + a * (b - c) + (b - c) * d

DAG acima é construído a partir da sequencia de passos abaixo, c/ a condição de que *Node* e *Leaf* retornem um nó existente, se possível.

Assumimos que *entry-a* aponta para a entrada da tabela de símbolos p/ a, e da mesma forma para os outros identificadores.

Quando a chamada *Leaf(id, entry-a)* é repetida no passo 2, o nó criado pela chama anterior é retornado, de modo que $p_2 = p_1$.

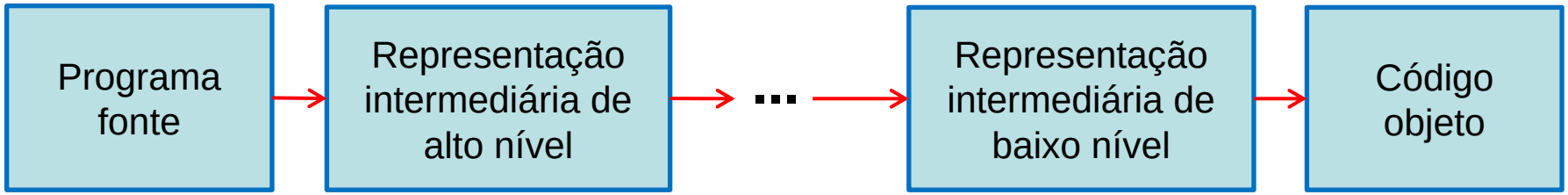
Os nós retornados nos passos 8 e 9 são iguais aos retornados em 3 e 4 ($p_8 = p_3$ e $p_9 = p_4$).

O nó da etapa 10 é igual ao 5 ($p_{10} = p_5$).

Síntese

Geração de código intermediário

As árvores de sintaxe são de alto nível; elas representam a estrutura hierárquica natural do programa fonte e são bem adequadas a tarefas como verificação de tipo estática.



Uma representação de baixo nível é adequada para tarefas dependentes de máquina, como a alocação de registradores e a seleção de instrução.

O código de 3 endereços pode variar de alto até baixo nível, dependendo da escolha dos operadores.

Síntese

Geração de código intermediário

Essas abordagens podem ser usadas com uma grande variedade de representações intermediárias, incluindo árvores de sintaxe e código de três endereços.

A razão do termo "*código de 3 endereços*" vem das instruções no formato geral "x = y op z" com três endereços: dois para os operadores y e z e um para o resultado x.

No processo de tradução de um programa, em uma dada linguagem fonte, para o código de determinada máquina alvo, um dado compilador pode construir uma sequencia de representações intermediárias.

As representações de alto nível estão próximas da linguagem fonte, e as representações de baixo nível estão próximas da máquina alvo.

Síntese

Geração de código intermediário

P/ comandos de *looping*, por exemplo, uma árvore de sintaxe representa os componentes de um comando, enquanto o código de três endereços possui rótulos e comandos de desvio para representar o fluxo de controle, como na linguagem de máquina.

Assim, a escolha ou o projeto de representação intermediária varia de um compilador para outro. Uma representação intermediária pode ser uma linguagem de alto nível corrente ou pode consistir em estrutura de dados internas que são compartilhadas pelas fases do compilador.

Ex: **C** é uma linguagem de programação de alto nível, embora frequentemente seja usado como uma forma intermediária, porque é flexível, compila para código de máquina eficiente e seus compiladores são encontrados c/ facilidade. O *front-end* do compilador **C++** original gera **C**, tratando o compilador **C** como *back-end*.

Síntese

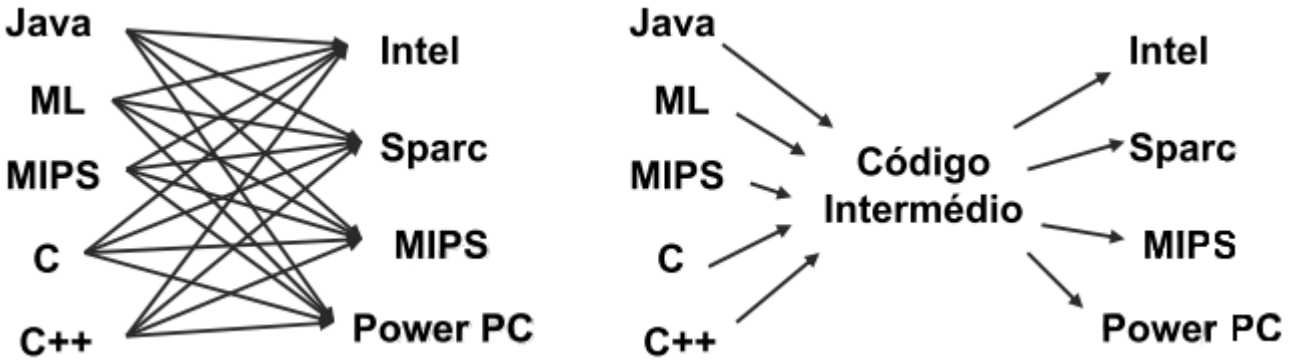
- Possibilita a otimização do código intermediário
  - Código objeto final mais eficiente

- Simplifica a implementação do compilador
  - Resolução gradativa da abstração das operações

O código intermediário abstrai detalhes da máquina alvo

Síntese

- Possibilita a tradução do código intermediário para diversas máquinas



Síntese

Desvantagem

O compilador precisa realizar um passo a mais, logo a tradução do código fonte para o objeto leva a uma compilação mais lenta.

Intermediário X objeto final

O intermediário não especifica detalhes da máquina alvo, tais como quais registradores serão usados, quais endereços de memória serão referenciados, etc.

Síntese

Geração de código intermediário

Representações intermediárias:

- representação gráfica: árvore sintática ou grafo
- Notação pós-fixada e pré-fixada
- Código de três endereços

Representação intermediária pode ser construída paralelamente à análise sintática:

- Tradução dirigida pela sintaxe

Síntese

No código de 3 endereços existe no máximo um operador do lado direito de uma instrução, ou seja, nenhum expressão aritmética construída com vários operadores é permitida.

Assim, uma expressão da linguagem fonte como x+y*z deve ser traduzida para sequencia de instruções de três endereços:

```
t1 = y * z
t2 = x + t1
```

, onde t1 e t2 são nomes temporários gerados pelo compilador.

Síntese


```
t1 = y * z
t2 = x + t1
```

Esse desdobramento de expressões aritméticas com múltiplos operadores de comando de fluxo de controle aninhados torna o código 3 endereços desejável para geração e otimização de código objeto.

O uso de nomes para os valores intermediários computados por um programa permite que o código de 3 endereços seja facilmente arranjado.

Síntese

Portanto, o código de três endereços é formado por uma sequência de comandos com o seguinte formato geral:

```
A := B op C
A := op B
A := B
goto L
if A oprel B goto L
```

Onde:

- A, B e C são nomes, constantes ou objetos de dados temporários criados pelo compilador;
- op está no lugar de qualquer operador aritmético;
- oprel é um operador relacional;
- L é um rótulo simbólico;

Síntese

Outros enunciados que serão usados para chamadas de procedimentos, por exemplo:

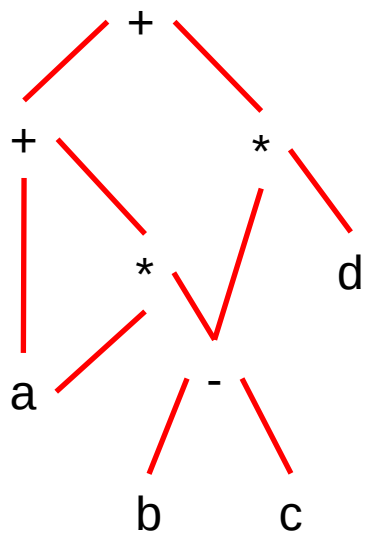
```
param X
call P, N
return Y
```

Onde:

- X é um parâmetro do procedimento;
- P é o nome do procedimento;
- N é número de parâmetros do procedimento;
- Y é o valor retornado (opcional);

Síntese

Exemplo: O código de três endereços é uma representação linear de uma árvore de sintaxe ou de um DAG (grafo direcionado acíclico), onde nomes explícitos correspondem aos nós interiores do grafo.

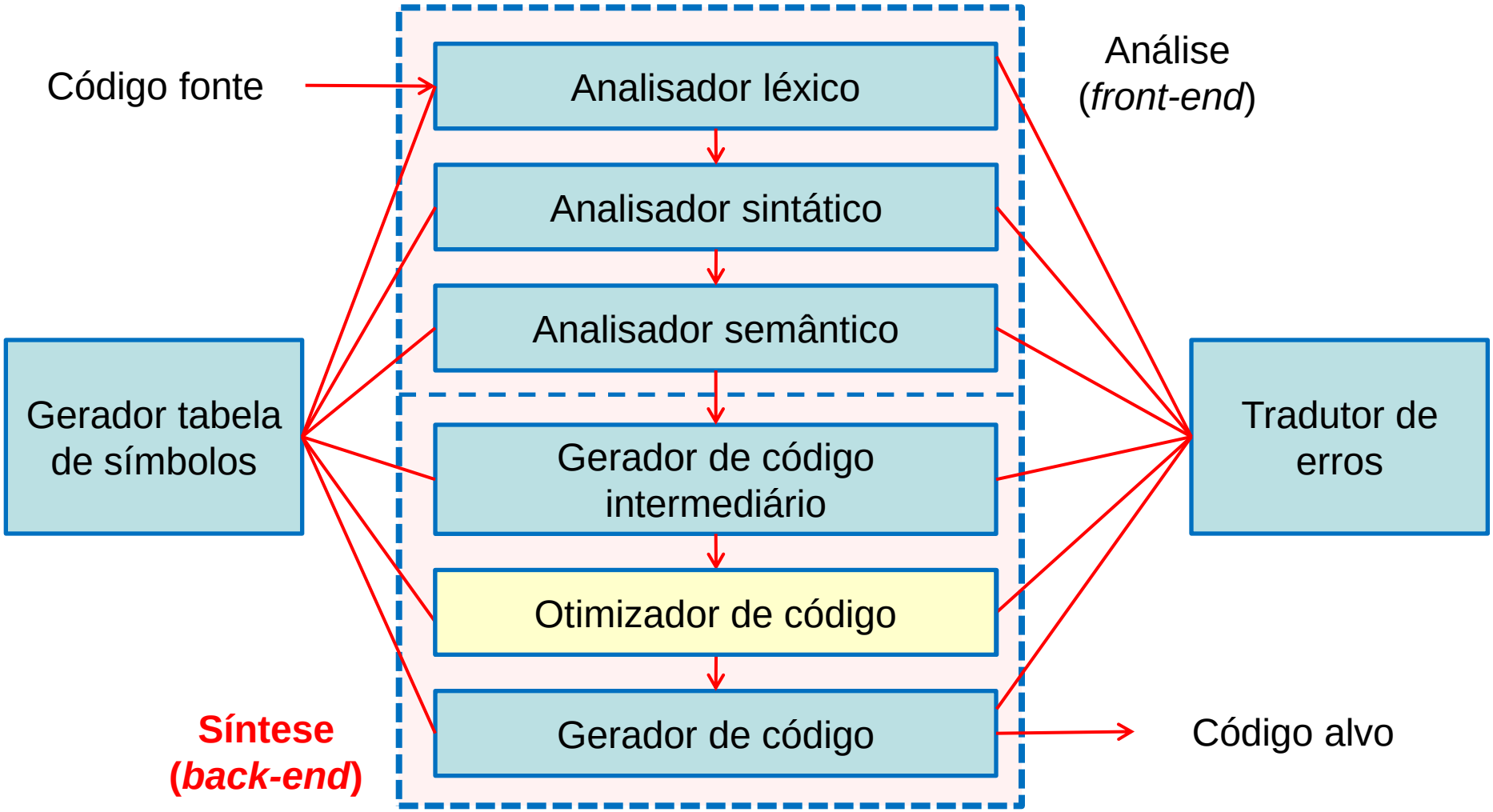


DAG de uma representação
a + a * (b - c) + (b - c) * d

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

Código de três endereços

Síntese


Código fonte
Analisador léxico
Analisador sintático
Analisador semântico
Gerador de código intermediário
Otimizador de código
Gerador de código
Gerador tabela de símbolos
Tradutor de erros
Código alvo
Análise (*front-end*)
Síntese (*back-end*)

Síntese

Otimizador de código

A otimização do código é a inferência de um conjunto de mudanças que melhoram a sequência de instruções de máquina (tempo de execução, memória ocupada, etc...) sem que seja modificada a semântica.

Apesar do termo otimização, são poucas às vezes que se pode garantir que o código obtido seja o melhor possível.

Síntese

Otimizador de código

A otimização de código faz com que o compilador gaste muito tempo de compilação.

Deve ser implementada se o uso do compilador realmente necessite de um código objeto (código de máquina) eficiente.

Síntese

Otimizador de código

O difícil é não modificar em nenhum caso o funcionamento do programa.

Exemplo: considere um trecho de programa em que aparece um comando x=a+b; Este programa pode ser melhorado (torna-se mais rápido e menor) se este comando for retirado.

P/ que o funcionamento do programa não seja alterado deve-se verificar algumas propriedades, por exemplo:

- comando é inútil, porque nenhum comando executado posteriormente usa o valor da variável x.
- comando x=a+b nunca é executado. Ex., está em seguida a um if cuja condição nunca é satisfeita:

```
if(0)
x=a+b;
```

Síntese

Otimizador de código

Oportunidades de otimização:

- Suponha que a mesma expressão ocorre mais de uma vez em um trecho de programa.
- Se as variáveis que ocorrem na expressão não tem seus valores alterados entre as duas ocorrências, é possível calcular seu valor apenas uma vez. Exemplo:

```
x=a+b;
...
y=a+b;
...
```

- Se os valores de a e de b não são alterados, pode-se guardar o valor da expressão a+b em uma temporária (t1) e usá-lo depois:

```
t1=a+b;
x=t1;
...
y=t1;
```

Síntese

Otimizador de código

Eliminação de expressões comuns:

- Operações que se repetem sem que seus argumentos sejam alterados podem ser realizadas uma única vez.

```
x = a + b + c;
...
Y = a + b + d;
```

Sem otimização

```
_t1 := a + b
x := _t1 + c
...
_t2 := a + b
y := _t2 + d
```

Com otimização

```
_t1 := a + b
x := _t1 + c
...
Y := _t1 + d
```

Síntese

Otimizador de código

Eliminação de código redundante:

- Instruções sem efeito podem ser eliminadas. Ex.:

Sem nenhuma atribuição a x ou y entre as instruções, a segunda instrução pode ser seguramente eliminada.

```
x := y
...
x := y
```

```
x := y
...
```

Propagação de cópias:

- Variáveis que mantém cópia de um valor, sem outros usos, podem ser eliminadas.

Ex.: sem atribuição a y e sem outros usos de x.

```
x := y
...
z := x
```

```
...
z := y
```

Pode ser reduzido a:

Síntese

Otimizador de código

Eliminação de desvios desnecessários:

- Desvio incondicional para a próxima instrução pode ser eliminado. Ex.:

```
        a := _t2
        goto _L6
_L6:    c := a + b
```

equivale a:

```
a := _t2
c := a + b
```

Uso de propriedades algébricas.

- Substituição de expressões aritméticas por formas equivalentes.

Original	Equivalente
x + y	y + x
x + 0	x
x – 0	x
x * y	y * x
x * 1	x
x / 1	x
2 * x	x + x
x^2	x * x

Síntese

Otimizador de código

Outro exemplo de otimização é a retirada de comandos de *loop* (laço de repetição:

```
for (i=0; i<N; i++) {
        a = j + 5;
        f(a * i);    }
```

```
a=j+5;
for (i=0; i<N; i++)
        f(a * i);
```

Poderia ser melhorado retirando-se o comando a=j+5; do *for*.

Por outro lado, se N=0, o programa foi "piorado", porque o comando a=j+5; que era executado 0 vezes, passou a ser executado 1 vez.

Pode haver um problema maior: se a variável a é usada após o *loop*, em vez de seu valor original, seu valor será, incorretamente, o resultado dado pela atribuição.

Síntese

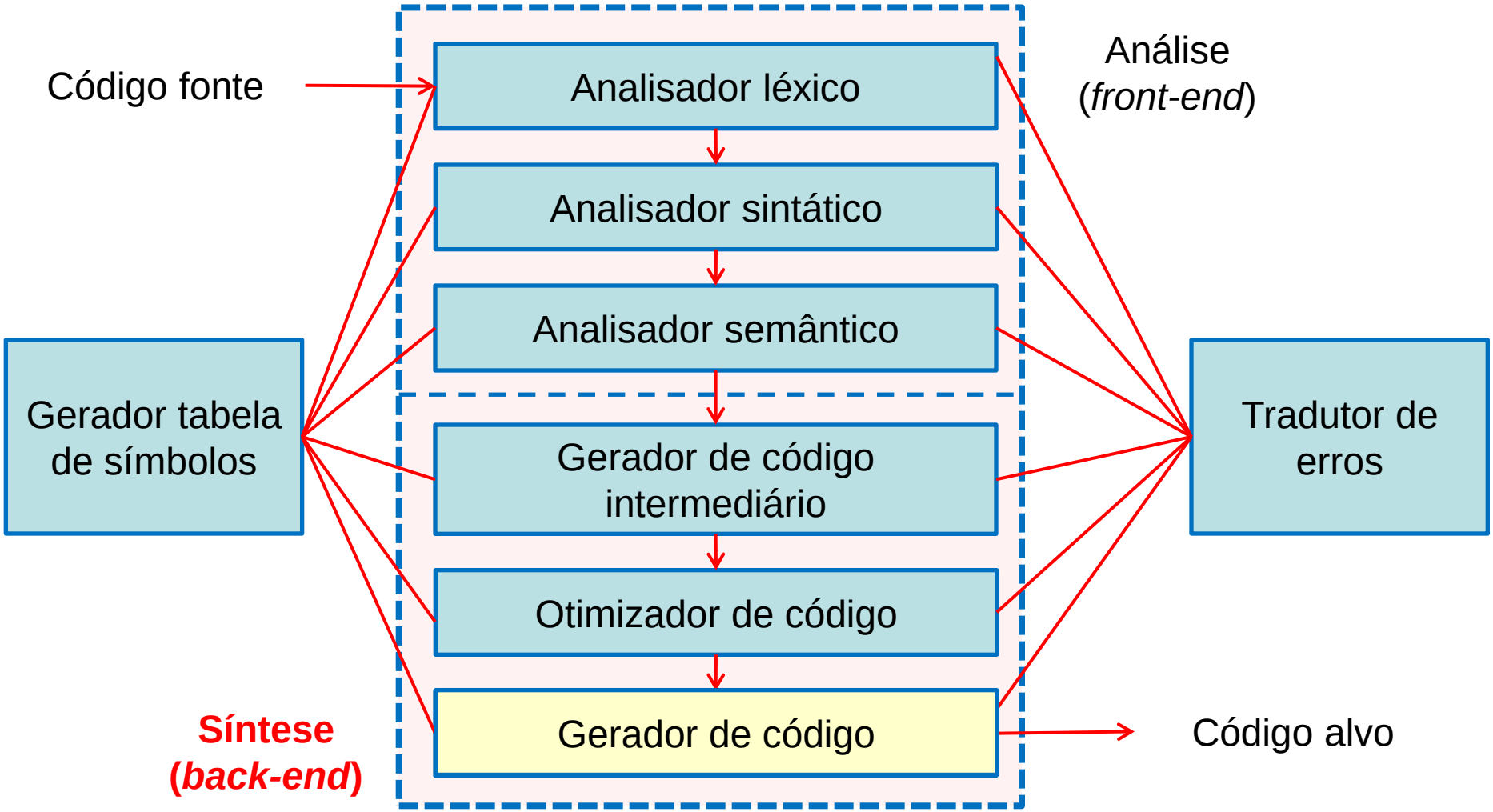
Otimizador de código

Outro problema é a quantidade de informação que se deseja manipular:

- examinar otimizações locais (em trechos pequenos de programas, por exemplo trechos sem desvios, ou seja, trechos em linha reta).
- otimizações em um nível intermediário (otimizações são consideradas apenas em funções, módulos, ou classes, dependendo da linguagem).
- otimizações globais (consideram as inter-relações de todas as partes de um programa).

A maioria dos compiladores oferece algumas otimizações do primeiro.

Síntese


Código fonte
Analisador léxico
Analisador sintático
Analisador semântico
Gerador de código intermediário
Otimizador de código
Gerador de código
Análise (*front-end*)
Gerador tabela de símbolos
Tradutor de erros
Síntese (*back-end*)
Código alvo

Síntese

Gerador de código

Enquanto a fase de análise é essencialmente dependente da linguagem de programação, a fase de geração de código depende principalmente da máquina alvo.

Sua principal função é gerar o código equivalente ao programa fonte para uma máquina real.

A fase de tradução converte o código fonte para um código objeto, que pode ser:

- um ASSEMBLY de uma determinada máquina.
- um pseudocódigo de uma máquina hipotética:
  - interpretado posteriormente.
  - pode ser executado em qualquer máquina.

Síntese

Gerador de código

Os principais requisitos impostos a geradores de código objeto são:

- O código gerado deve ser correto e de alta qualidade;
- O código gerado deve fazer uso efetivo dos recursos da máquina; e
- O código gerado deve executar eficientemente.

O problema de gerar código ótimo é insolúvel como tantos outros.

Na prática, devemos usar heurísticas que geram um “bom” código.

A última etapa do compilador propriamente dito é a geração do código em linguagem simbólica.

Uma vez que esse código seja gerado, outro programa (o montador) será responsável por traduzir o código para formato de linguagem de máquina.

Síntese

Gerador de código

A abordagem mais simples da etapa de geração de código objeto é:

- Para cada instrução (do código intermediário) ter um gabarito com a correspondente sequência de instruções em linguagem simbólica do processador-alvo.

- Por exemplo: `le := ld1 + ld2`

A sequência de instruções em linguagem simbólica que corresponde a essa instrução depende da arquitetura do processador para o qual o programa é gerado.

Síntese

Gerador de código

Diferentes processadores podem ter distintos formatos de instruções.

Classificação pelo número de endereços na instrução:

- 3: dois operandos e o resultado;
- 2: dois operandos (resultado sobrescreve primeiro operando);
- 1: apenas segundo operando, primeiro operando implícito (registrador acumulador), resultado sobrescreve primeiro operando;
- 0: operandos e resultado numa pilha, sem endereço explícitos;

Síntese

Gerador de código

Tradução para a linguagem simbólica:

- Tradução ocorre segundo gabaritos definidos de acordo com o tipo de máquina, exemplo: x := y + z;

3-end
```
ADD y, z, x
```

2-end
```
MOVE Ri, y
ADD Ri, z
MOVE x, Ri
```

1-end
```
LOAD y
ADD z
STORE x
```

0-end
```
PUSH y
PUSH z
ADD
POP x
```

Síntese

Gerador de código

O resultado da compilação é um arquivo em linguagem simbólica.

Montagem

- Processo em que o programa em linguagem simbólica é transformado em formato binário, em código de máquina.
- O programa responsável por essa transformação é o montador.

Montadores

- Traduzem código em linguagem simbólica p/ linguagem de máquina.

Universidade de Santa Cruz do Sul – UNISC
Departamento de informática

COMPILADORES

Obrigado!!

Prof. Geovane Griesang
geovanegriesang@unisc.br

Créditos desta aula ao professor Dr. José Lucas Rangel: http://www-di.inf.puc-rio.br/~rangel/