Aula 04 – Análise Semântica - Sistemas de Tipos

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**

Introdução

- O terceiro módulo do compilador realiza a análise semântica.
- Esta análise pode ser subdividida em várias etapas.
- **Estes *slides***: discussão sobre as principais ações de verificação de tipos realizadas por um compilador na fase de análise semântica.
- **Objetivos**: apresentar as teorias e aspectos práticos que envolvem essa fase.

Referências

Chapter 6 – Semantic Analysis
K. C. Louden
Chapter 4 – Context-Sensitive Analysis
K. D. Cooper
Chapter 7 – Semantic Analysis
D. Thain

Introdução

- Após a análise léxica (*scanning*) e a análise sintática (*parsing*) vem a fase de análise semântica (ou elaboração semântica).
- Propósito: computar informações adicionais necessárias para o processo de compilação que não podem ser obtidas pelos algoritmos de *parsing*.
- Possíveis ações desta etapa:
  - Construção de uma tabela de símbolos/literais (Aula 03).
  - Verificação de tipos de expressões/declarações.
  - Inferência de tipos.
  - Teste de limites de vetores, etc, etc.

Introdução – Análise Semântica

- A análise semântica requer:
  - Uma descrição das análises a serem realizadas.
  - Uma implementação destas análises usando algoritmos apropriados.
- Analogia com *parsing*: gramática em BNF seria a descrição e o algoritmo de *parsing* seria a implementação.
- Infelizmente não há um método consolidado como BNF para descrição da semântica de LPs.
- Um método originalmente proposto para descrever a análise semântica estática é a gramática de atributos (Knuth, 1968).

Introdução – Gramática de Atributos

- A construção de uma gramática de atributos requer:
  - Identificar atributos das entidades da LP que devem ser computados.
  - Escrever equações de atributos (também chamadas de regras semânticas) que expressam como a computação desses atributos está relacionada com as regras da gramática da LP.
- Gramáticas de atributos são mais úteis para LPs que seguem o princípio da semântica dirigida pela sintaxe, que determina que a semântica de um programa deve ter uma forte relação com a sua sintaxe.
- Todas as LPs atuais seguem esse princípio.

$\Rightarrow$ Para entendermos a implementação da verificação de tipos do compilador, devemos estudar primeiro a teoria de gramática de atributos.

Parte I

Gramáticas de Atributos

Atributos

- Atributo – qualquer propriedade de um construto de LPs:
  - O tipo de dados de uma variável.
  - O valor de uma expressão.
  - A localização de uma variável na memória.
  - O código objeto de um procedimento.
  - A quantidade de dígitos significativos em um número.
- Amarração (*binding*) de um atributo: processo de computar um atributo e associar o valor computado ao construto em questão da LP.
- Tempo de amarração: momento durante a compilação e/ou execução em que ocorre a amarração de um atributo.
- Dependendo do seu tempo de amarração, atributos são divididos em:
  - Estáticos: amarrados antes da execução do programa.
  - Dinâmicos: amarrados durante a execução do programa.

Atributos

Considerando os exemplos de atributos apresentados no slide anterior, discutimos para cada um o tempo de amarração e importância no processo de compilação.

- **Verificador de tipos** (*type checker*)
  - Em uma LP com tipos estáticos como C ou Pascal, é um componente importante da análise semântica.
  - Em uma LP com tipos dinâmicos como Python, o compilador deve gerar código para computar tipos e realizar a verificação dinâmica de tipos durante a execução.
- **Valores de expressões**
  - Geralmente são dinâmicos e computados durante a execução.
  - No entanto, expressões constantes pode ser avaliadas durante a compilação (*constant folding*).

Atributos

- **Alocação de uma variável**
  - Tempo de amarração varia conforme a LP e as propriedades da variável.
  - Totalmente estática: FORTRAN 77.
  - Totalmente dinâmica: LISP.
  - Mistura dos dois: C.
- **Código objeto de um procedimento**
  - Um atributo estático, computado pelo gerador de código do compilador.
- **Quantidade de dígitos significativos em um número**
  - Geralmente não é tratado explicitamente durante a compilação.

Gramática de Atributos

- Na semântica dirigida pela sintaxe, atributos são associados diretamente aos símbolos (*tokens* e não-terminais) da gramática.
- $X.a$: valor do atributo a associado ao símbolo X.
- Dada uma coleção de atributos $a_1, \ldots, a_k$, para cada regra $X_0 \to X_1 \ldots X_n$, o valor dos atributos $X_i.a_j$ está relacionado aos atributos dos demais símbolos da regra.
- Uma gramática de atributos é uma coleção de equações de atributos (regras semânticas) da forma:

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

  onde f_{ij} é uma função com $(n+1) \cdot k$ parâmetros.
- Embora as funções f_{ij} pareçam bastante complexas na teoria, na prática elas são bastante simples.
- Gramáticas de atributos são tipicamente apresentadas em forma de tabela.

Gramática de Atributos – Exemplo 1

Gramática de números sem sinal:

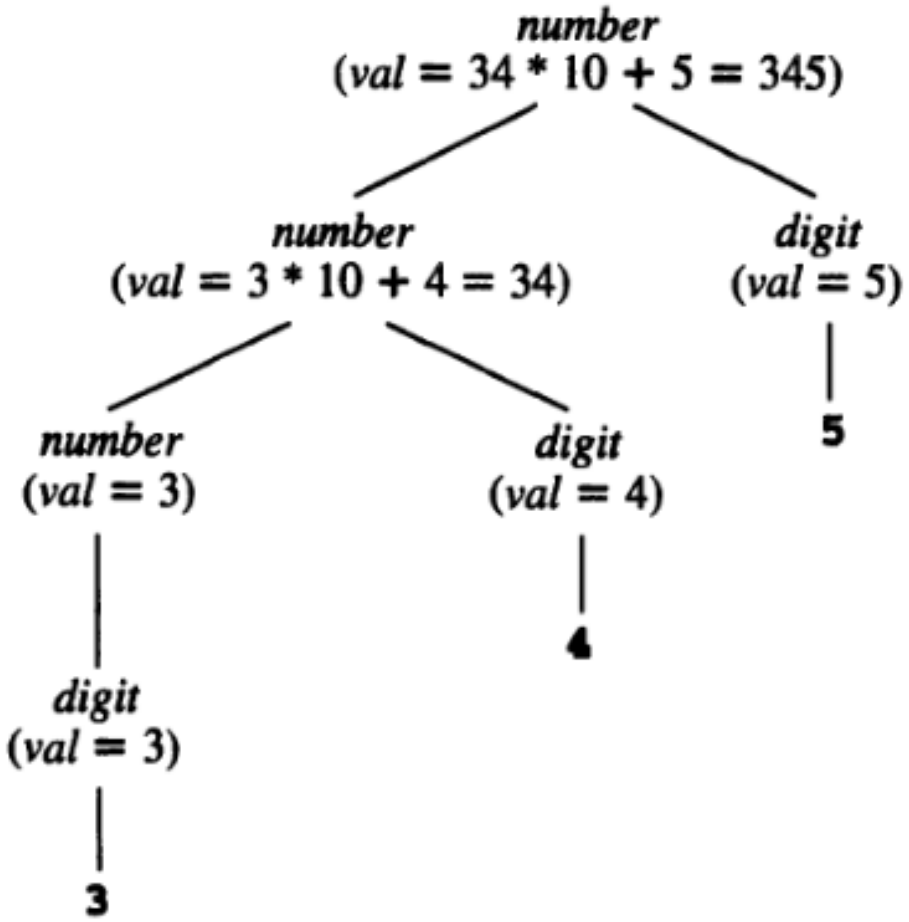
$$number \rightarrow number\ digit \mid digit$$
$$digit \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

O valor do número é dado pelo atributo *val*.

Grammar Rule	Semantic Rules
$number_1 \rightarrow number_2\ digit$	$number_1.val = number_2.val * 10 + digit.val$
$number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow \mathbf{0}$	$digit.val = 0$
$digit \rightarrow \mathbf{1}$	$digit.val = 1$
$digit \rightarrow \mathbf{2}$	$digit.val = 2$
$digit \rightarrow \mathbf{3}$	$digit.val = 3$
$digit \rightarrow \mathbf{4}$	$digit.val = 4$
$digit \rightarrow \mathbf{5}$	$digit.val = 5$
$digit \rightarrow \mathbf{6}$	$digit.val = 6$
$digit \rightarrow \mathbf{7}$	$digit.val = 7$
$digit \rightarrow \mathbf{8}$	$digit.val = 8$
$digit \rightarrow \mathbf{9}$	$digit.val = 9$

Gramática de Atributos – Exemplo 1

O significado das regras semânticas para uma dada entrada (e.g., `345`) pode ser visualizado na *parse tree*.



Gramática de Atributos – Exemplo 2

Gramática de expressões aritméticas:

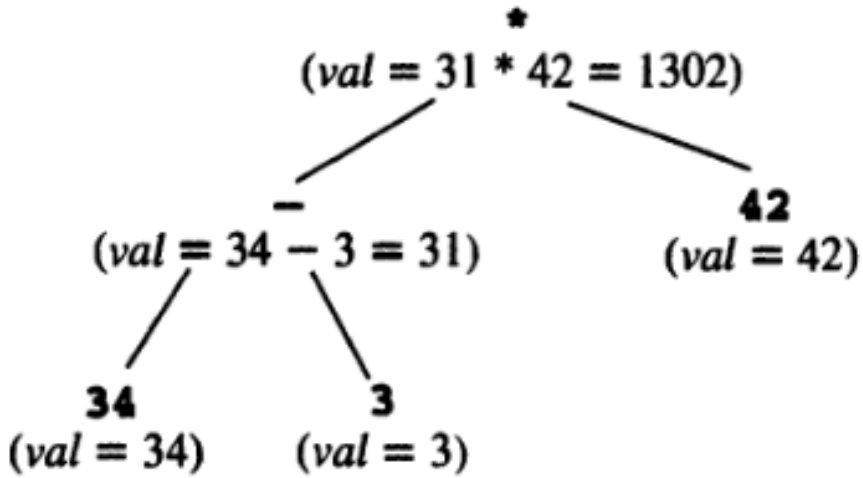
$$exp \rightarrow exp + exp \mid exp - exp \mid exp * exp \mid (exp) \mid \textbf{number}$$

O valor da expressão é dado pelo atributo *val*.

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 + exp_3$	$exp_1.val = exp_2.val + exp_3.val$
$exp_1 \rightarrow exp_2 - exp_3$	$exp_1.val = exp_2.val - exp_3.val$
$exp_1 \rightarrow exp_2 * exp_3$	$exp_1.val = exp_2.val * exp_3.val$
$exp_1 \rightarrow (exp_2)$	$exp_1.val = exp_2.val$
$exp \rightarrow \textbf{number}$	$exp.val = \textbf{number}.val$

Gramática de Atributos – Exemplo 2

O significado das regras semânticas para uma dada entrada (e.g., `(34-3)*42`) pode ser visualizado na AST.



Gramática de Atributos – Exemplo 3

Uma gramática de atributos também pode ser utilizada para construir a AST do programa de entrada de forma *bottom-up*.

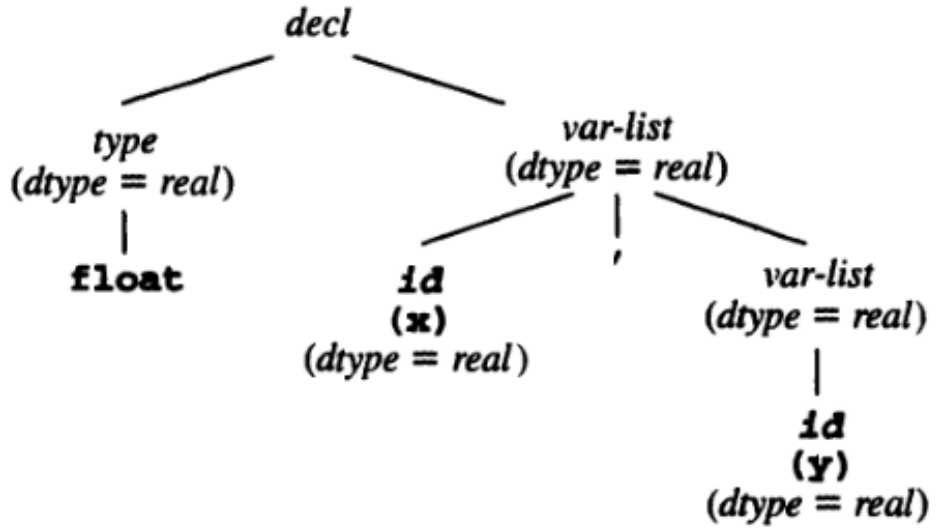
Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ + $term$	$exp_1.tree =$ $mkOpNode(+, exp_2.tree, term.tree)$
$exp_1 \rightarrow exp_2$ - $term$	$exp_1.tree =$ $mkOpNode(-, exp_2.tree, term.tree)$
$exp \rightarrow term$	$exp.tree = term.tree$
$term_1 \rightarrow term_2$ * $factor$	$term_1.tree =$ $mkOpNode(*, term_2.tree, factor.tree)$
$term \rightarrow factor$	$term.tree = factor.tree$
$factor \rightarrow$ (exp)	$factor.tree = exp.tree$
$factor \rightarrow$ **number**	$factor.tree =$ $mkNumNode(\textbf{number}.lexval)$

Gramática de Atributos – Exemplo 4

Nos exemplos anteriores, o cálculo dos atributos era feito das folhas para a raiz mas isso não é obrigatório pela teoria.

decl → *type var-list*
type → **int** | **float**
var-list → **id,** *var-list* | **id**

Grammar Rule	Semantic Rules
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*
var-list$_1$ → **id,** *var-list*$_2$	**id** .*dtype* = *var-list*$_1$.*dtype*
	var-list$_2$.*dtype* = *var-list*$_1$.*dtype*
var-list → **id**	**id** .*dtype* = *var-list.dtype*



AST para `float x,y`:

Gramática de Atributos

- Uma questão fundamental para a especificação de atributos usando gramáticas de atributos é a apresentada a seguir.
- Como podemos ter certeza que uma gramática é consistente e completa, i.e., que define os atributos de forma única?
- Resposta simples: não podemos (até agora).
- Essa pergunta é similar a determinar se uma gramática livre de contexto (CFG) é ambígua.
- Na prática: algoritmos de *parsing* determinam se uma CFG é adequada.
- O mesmo pode ser feito para gramáticas de atributos.

Algoritmos para Computação de Atributos

- Objetivo: estudar como uma gramática de atributos pode ser usada como base para um compilador computar e usar os atributos definidos pelas regras semânticas.
- Ideia básica é transformar as equações de atributos em regras de computação.
- Uma equação como

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

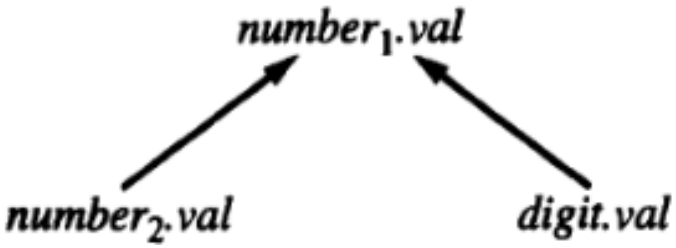
  é vista como uma atribuição do valor da expressão funcional do lado direito para o atributo $X_i.a_j$.
- As equações indicam restrições sobre a ordem de computação dos atributos.
- Para calcular $X_i.a_j$ é necessário que todos os atributos do lado direito já estejam computados.

Algoritmos para Computação de Atributos

- O problema de implementar um algoritmo para o cálculo de atributos consiste primariamente em achar uma ordem válida para a avaliação e atribuição dos atributos.
- Obviamente, uma gramática de atributos com dependência circular não pode ser implementada.
- As restrições de ordem de uma gramática de atributos são representadas por grafos direcionados chamados grafos de dependências.
- *Exemplo*: a regra semântica

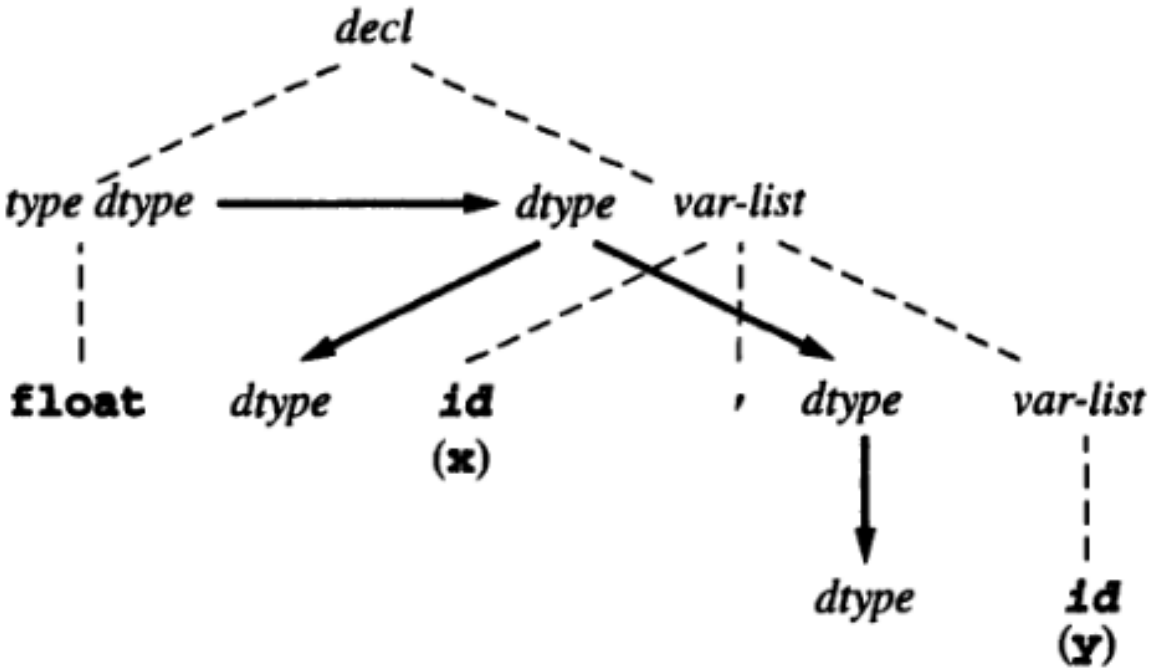
$$number_1.val = number_2.val * 10 + digit.val$$

gera o grafo de dependências abaixo.



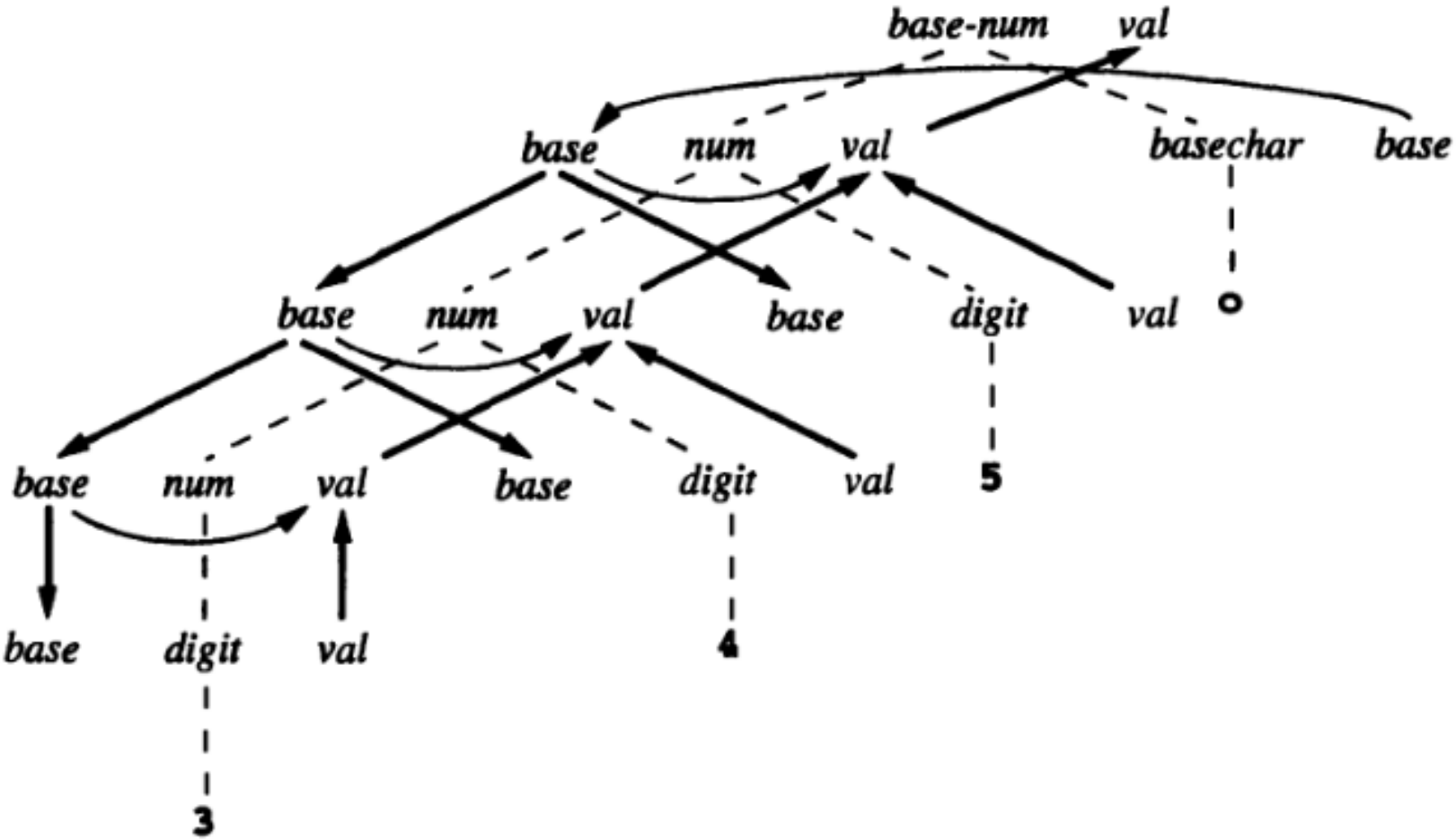
Algoritmos para Computação de Atributos

O grafo de dependências pode ser sobreposto à *parse tree*, como ilustrado abaixo para a entrada `float x,y`.



Algoritmos para Computação de Atributos

Um grafo de dependências pode ser bastante complexo.

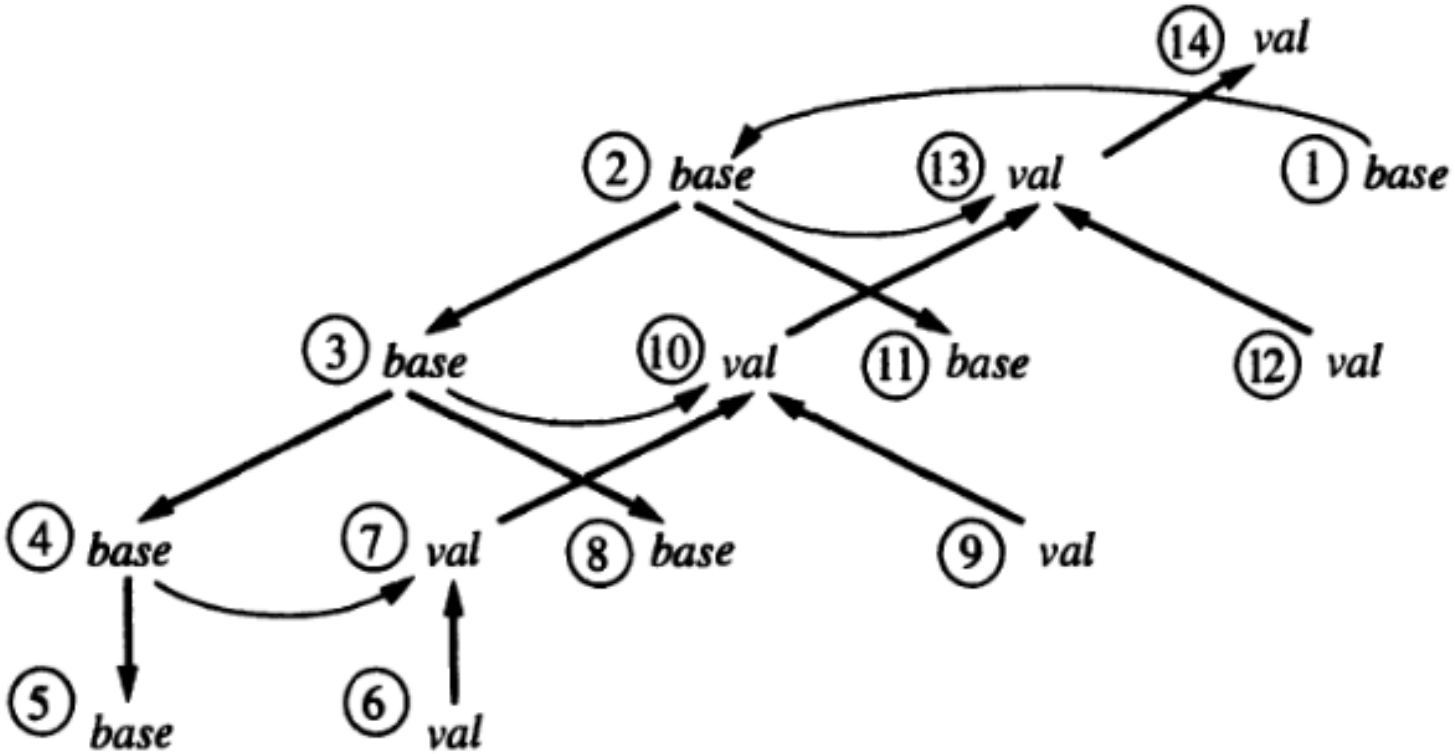


Algoritmos para Computação de Atributos

- Qualquer algoritmo deve computar os atributos de um dado nó antes de computar os atributos sucessores.
- Uma ordem de caminhamento do grafo de dependências que satisfaz a restrição acima é chamada de ordenação topológica.
- Uma condição necessária e suficiente para a existência de uma ordenação topológica é que o grafo seja acíclico.
- Um grafo direcionado e acíclico é chamado DAG (*directed acyclic graph*).

Algoritmos para Computação de Atributos

O grafo abaixo é um DAG e uma ordenação topológica é indicada pelos números abaixo.



Um mesmo DAG pode ter mais de uma ordenação. *Exemplo*:

12 6 9 1 2 11 3 8 4 5 7 10 13 14 .

Algoritmos para Computação de Atributos

- Um algoritmo para ordenação topológica tem complexidade linear em relação ao tamanho (número de nós e arestas) do grafo de dependências.
- No entanto, para garantir que o algoritmo de ordenação sempre tenha sucesso, é necessário que a gramática de atributos seja não-circular.
- Uma gramática é dita não-circular se todo e qualquer grafo de dependências possível for acíclico.
- Essa condição é necessária pois, caso contrário, o compilador poderia falhar para algumas entradas e funcionar para outras.
- Infelizmente, verificar se uma gramática é não-circular é um problema de complexidade exponencial.
- Por conta dessa complexidade, o método descrito acima (chamado de *parse tree method*) não é usado na prática.

Algoritmos para Computação de Atributos

- A alternativa ao método anterior é chamada de *rule-based method*, que é adotada por praticamente todos os compiladores.
- Requer que o desenvolvedor do compilador fixe uma ordem para a avaliação dos atributos durante a construção do compilador.
- Gramáticas de atributos que permitem a fixação de uma ordem como acima são chamadas de fortemente não-circulares.
- Gramáticas fortemente não-circulares formam uma classe menos geral que a classe de todas as gramáticas não-circulares.
- Na prática, todas as gramáticas de atributos usadas em LPs são fortemente não-circulares.

Atributos Sintetizados e Herdados

- Atributos podem ser classificados pelas suas dependências em dois tipos:
  - Sintetizado: quando todas as dependências do atributo forem de filhos para o pai na *parse tree*.
  - Herdado: caso contrário.
- Gramática S-atribuída: uma gramática em que todos os atributos são sintetizados.
- Os valores dos atributos de uma gramática S-atribuída podem ser computados por um caminhamento em pós-ordem (*bottom-up*) da *parse tree* ou AST.
- Trivial de implementar.

Atributos Sintetizados e Herdados

A figura abaixo mostra os dois tipos básicos de dependências de atributos herdados.

(a) Inheritance from parent to siblings

(b) Inheritance from sibling to sibling

- Atributos herdados podem ser computados por um caminhamento em pré-ordem ou uma combinação de pré-ordem/em-ordem.
- Complexo de implementar, inviável quando o número de atributos é grande.

Computação de Atributos Durante *Parsing*

- Questão: quais atributos pode ser computados durante a fase de *parsing*, sem ser necessário realizar passadas adicionais pelo código?
- Resposta depende do poder e propriedades do método de *parsing* empregado.
- Todos os métodos de *parsing* processam o programa de entrada da esquerda para a direita (LL, LR).
- Isso é equivalente a exigir que os atributos possam ser avaliados por um caminhamento da esquerda para a direita na *parse tree*.
- Não é um problema para os atributos sintetizados: qualquer ordem de caminhamento nos filhos serve.
- Para atributos herdados isso implica que não podem haver dependências “para trás” (da direita para a esquerda).
- Gramáticas que satisfazem a restrição acima são chamadas L-atribuídas.

Computação de Atributos Durante *Parsing*

- Uma gramática é L-atribuída se, para cada atributo herdado a_j e cada regra $X_0 \to X_1 \ldots X_n$, o valor de $X_i.a_j$ depende somente dos atributos dos símbolos $X_0, \ldots, X_{i-1}$.
- Uma gramática S-atribuída é um caso trivial de gramática L-atribuída (não possui atributos herdados).
- Como um *parser* lida com uma gramática L-atribuída:
  - *Parser* de descida recursiva: atributos herdados viram parâmetros e atributos sintetizados viram valor de retorno das funções de *parsing* dos não-terminais.
  - *Parser* LR: a princípio não consegue lidar com gramáticas L-atribuídas, somente S-atribuídas.
- Um *parser* LR mantém uma pilha de valores para armazenar atributos sintetizados, que é manipulada em paralelo com a pilha de *parsing*.

Computação de Atributos Durante *Parsing*

Exemplo de ações semânticas para a expressão `3*4+5`.

	Parsing Stack	Input	Parsing Action	Value Stack	Semantic Action
1	$	3*4+5 $	shift	$	
2	$ n	*4+5 $	reduce $E \rightarrow$ n	$ n	$E.val =$ n$.val$
3	$ E	*4+5 $	shift	$ 3	
4	$ E *	4+5 $	shift	$ 3 *	
5	$ E * n	+5 $	reduce $E \rightarrow$ n	$ 3 * n	$E.val =$ n$.val$
6	$ E * E	+5 $	reduce $E \rightarrow E * E$	$ 3 * 4	$E_1.val = E_2.val * E_3.val$
7	$ E	+5 $	shift	$ 12	
8	$ E +	5 $	shift	$ 12 +	
9	$ E + n	$	reduce $E \rightarrow$ n	$ 12 + n	$E.val =$ n$.val$
10	$ E + E	$	reduce $E \rightarrow E + E$	$ 12 + 5	$E_1.val = E_2.val + E_3.val$
11	$ E	$		$ 17	

No `bison`, a redução do passo 10 fica descrita como abaixo.

```
E : E + E { $$ = $1 + $3; }
```

Computação de Atributos Durante *Parsing*

- É possível lidar com gramáticas L-atribuídas em um *parser* LR mas isso requer a introdução de ϵ-produções.
- Suponha uma regra $A \rightarrow B\ C$ aonde um atributo de C depende de um atributo de B, por exemplo, pela equação $C.i = f(B.s)$.
- O valor de $f(B.s)$ pode ser armazenado em uma variável antes do reconhecimento de C, introduzindo-se uma ϵ-produção entre B e C.

Grammar Rule	Semantic Rules
$A \rightarrow B\ D\ C$	
$B \rightarrow \ldots$	{ compute $B.s$ }
$D \rightarrow \epsilon$	$saved\_i = f(valstack[top])$
$C \rightarrow \ldots$	{ now $saved\_i$ is available }

- No `bison`, a ϵ-produção não precisa ser introduzida de forma explícita.

```
A : B { saved_i = f($1); } C ;
```

Parte II

Verificação de Tipos

Uma Introdução aos Sistemas de Tipos

- Tipo: uma coleção de dados que especifica propriedades comuns a todos os elementos da coleção.
- Exemplos de tipos comuns em LPs: inteiro, caractere, lista, etc.
- Tipos podem ser especificados por pertinência em um conjunto ou intervalo:
  - `int`: qualquer número inteiro i na faixa $-2^{31} \leq i < 2^{31}$.
  - `colors`: tipo enumerado definido pelo conjunto {`red, green, blue`}.
- Tipos podem ser especificados por regras.
  Ex.: `struct` em C.
- O conjunto de tipos de uma LP, juntamente com regras que usam tipos para especificar o comportamento do programa, formam o sistema de tipos (*type system – TS*).

Propósito do Sistemas de Tipos

- Projetos de LPs introduzem sistemas de tipos para permitir a especificação de comportamento dos programas de forma mais precisa.
- CFG define a sintaxe (forma) válida dos programas.
- TS define a forma e comportamento válidos dos programas.
- Analisar um programa usando um TS produz informações que não podem ser obtidas puramente pelas análises léxicas e sintáticas.
- Em um compilador, as informações do TS são usadas para três propósitos distintos:
  - Garantir segurança em tempo de execução.
  - Aumentar a expressividade da LP.
  - Melhorar a eficiência do programa gerado.

TS para Garantir a Segurança na Execução

- Um TS deve garantir que programas são bem comportados, i.e., o compilador + sistema de execução devem identificar programas mal-formados antes da execução de alguma operação que causa um erro.
- Nem sempre isso é possível. Por quê?
- Porque determinar se um programa é mal-formado é um problema indecidível.
- Mesmo assim, geralmente quanto mais elaborado o TS mais problemas podem ser detectados sem causar erros.
- Requer que o compilador infira um tipo para cada expressão do programa.
- Inferência de tipos: processo de determinar um tipo para cada nome e expressão no código.

Tipos de Dados e Verificação de Tipos

- Inferência de tipos e verificação de tipos são duas das principais tarefas de um compilador.
- Ambas são fortemente relacionadas e portanto geralmente são realizadas juntas.
- Informações sobre tipos de dados podem ser estáticas ou dinâmicas, dependendo da LP.
- Quando todas as informações de tipos são estáticas, o ambiente de execução fica bem mais simples.
- Informações estáticas sobre tipos são o principal mecanismo para se verificar se um programa está correto sem executá-lo.
- Também servem para determinar a quantidade necessária de memória para alocação das variáveis.

Tipos de Dados e Verificação de Tipos

- Um tipo de dados (*data type*) é um conjunto de valores mais as operações que podem ser realizadas sobre eles.
- Tais conjuntos podem ser descritos por uma expressão de tipo, que pode ter dois formatos.
  - Um nome de tipo: `integer`.
  - Uma expressão estruturada: `array [1..10] of real`.
- Informações de tipo podem ser:
  - **Explícitas**: `var x:array [1..10] of real`.
  - **Implícitas**: `const str = "Hello"`. (Vetor de `char[6]`).
- Informações de tipos são mantidas na tabela de símbolos e consultadas pelo verificador de tipos.

Tipos de Tipos

- Tipos básicos (base, primitivos, etc):
  - Números (inteiros e reais)
  - Booleanos
  - Caracteres
- Tipos compostos (construídos, estruturados, etc):
  - Vetores
  - *Strings*
  - Tipos enumerados
  - *Structs* e variantes
  - Ponteiros
  - Funções (se forem elementos de primeira classe na LP).
- Tipos polimórficos:
  - Parametrizados: só definem um tipo “de fato” quando instanciados.

Relações Entre Tipos

- Equivalência de tipos pode ser por nome ou estrutura.
- *Exemplo*: tipos abaixo não são equivalentes por nome mas são por estrutura.

```
struct Tree {
  struct Tree *left;
  struct Tree *right;
  int value;
};
```

```
struct STree {
  struct STree *l;
  struct STree *r;
  int number;
};
```

- Sub-tipos: critério?
- Valores do sub-tipo $\subseteq$ valores do super-tipo.
- Sub-tipos podem ser inferidos (pela sua estrutura) ou declarados (por um nome).
- Herança $\neq$ sub-tipagem!
- Uma LP não precisa ser OO para ter sub-tipos.
- *Exemplo*: definição dos números naturais em Ada:

```
subtype Natural is Integer range 0 .. Integer'Last;
```

Relações Entre Tipos

- *Type casting*: Tipo real (dinâmico) pode ser um sub-tipo de um tipo conhecido (estático).
- Tipo estático "modificado" pelo *cast*.
- *Up-casting* (implícito) e *down-casting* (explícito).

```
String s = "Hi";          // Example in Java:
Object o = s;             // Up-casting
String t = (String) o;    // Down-casting
```

- *Type coercion*: Força a alteração de um valor de um tipo para um valor de outro tipo.
- Usado principalmente para tipos primitivos.
- *Widening* (implícito) e *narrowing* (explícito).

```
int a = 1;     double b = a;     int c = (int) b;
```

- *Coerced types* não são sub-tipos ou equivalentes.
- *Coercion* $\neq$ *casting*! Apesar da notação de ambos coincidir na maioria das LPs.

Verificação e Inferência de Tipos

- Na *parse tree*, todo nó é um símbolo da gramática.
- Os símbolos que correspondem a expressões são tipados.
- Como estabelecer os tipos dos nós da *parse tree*?
  1. Tipo pode ser sintetizado do símbolo e seus filhos.
  2. Tipo pode ser herdado de outros nós da árvore.
- *Exemplo do caso 1*: tipo de `x+2` em Java?
- O tipo é `int` se `x` é `int`, `String` se `x` é `String`, inválido se `x` é `Object`.
- *Exemplo do caso 2*: tipo de `x,y,z` em `if (x & y) { z = 3; }`?
- `x` e `y` são Booleanos porque `x & y` é Booleano (tipo esperado pelo `if`).
- `z` é `int` porque `z = 3` é `int` (tipo da expressão sintetizado do tipo de `3`).
- Caso 2 é muito importante em LPs com tipagem implícita.

Verificação e Inferência de Tipos

- A verificação e inferência de tipos pode ser especificada por uma gramática de atributos.
- Erros que podem ocorrer:
  - Algum filho do nó tem um tipo "errado".
    Ex.: operação de subtração com filhos do tipo *string*.
  - Tipo herdado não corresponde ao tipo sintetizado.
- Exemplo de uma gramática que infere os tipos de expressões aritméticas.

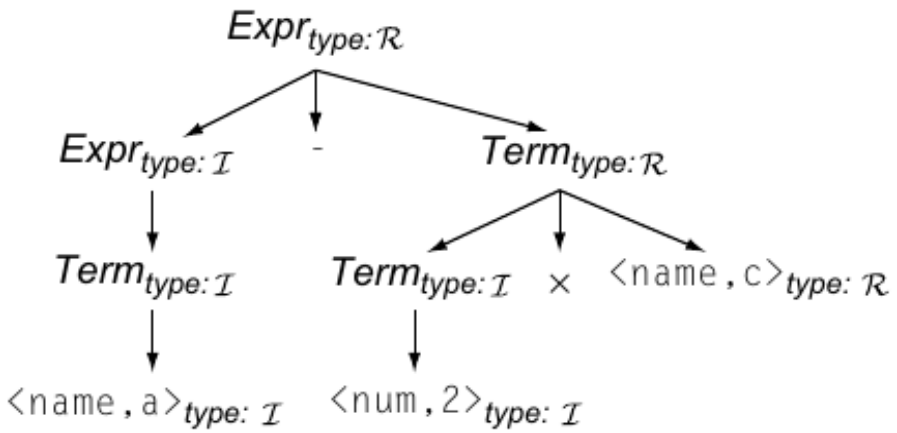
Production	Attribution Rules
$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.type \leftarrow \mathcal{F}_+(Expr_1.type, Term.type)$
$\mid Expr_1 - Term$	$Expr_0.type \leftarrow \mathcal{F}_-(Expr_1.type, Term.type)$
$\mid Term$	$Expr_0.type \leftarrow Term.type$
$Term_0 \rightarrow Term_1 \ Factor$	$Term_0.type \leftarrow \mathcal{F}_\times(Term_1.type, Factor.type)$
$\mid Term_1 \ Factor$	$Term_0.type \leftarrow \mathcal{F}_\div(Term_1.type, Factor.type)$
$\mid Factor$	$Term_0.type \leftarrow Factor.type$
$Factor \rightarrow (Expr)$	$Factor.type \leftarrow Expr.type$
$\mid$ num	num.type *is already defined*
$\mid$ name	name.type *is already defined*

Verificação e Inferência de Tipos

As funções $\mathcal{F}_+$, etc, correspondem às regras definidas por uma tabela como abaixo (exemplo da semântica de Fortran 77).

+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	*illegal*
complex	complex	complex	*illegal*	complex

Exemplo da *parse tree* para `a-2×c`, assumindo que `a` é do tipo inteiro ($\mathcal{I}$) e `c` é do tipo real ($\mathcal{R}$).



Verificação e Inferência de Tipos

Exemplo da inferência de tipos utilizando as ações semânticas no formato do Bison.

Production	Syntax-Directed Actions
$Expr \rightarrow Expr - Term$	{ $\$\$ \leftarrow \mathcal{F}_{+}(\$1, \$3)$ }
$\mid Expr - Term$	{ $\$\$ \leftarrow \mathcal{F}_{-}(\$1, \$3)$ }
$\mid Term$	{ $\$\$ \leftarrow \$1$ }
$Term \rightarrow Term \times Factor$	{ $\$\$ \leftarrow \mathcal{F}_{\times}(\$1, \$3)$ }
$\mid Term \div Factor$	{ $\$\$ \leftarrow \mathcal{F}_{\div}(\$1, \$3)$ }
$\mid Factor$	{ $\$\$ \leftarrow \$1$ }
$Factor \rightarrow$ ($Expr$)	{ $\$\$ \leftarrow \$2$ }
$\mid$ num	{ $\$\$ \leftarrow$ *type of the* num }
$\mid$ name	{ $\$\$ \leftarrow$ *type of the* name }

Verificação e Inferência de Tipos

Exemplo que mostra como a construção da AST e a inferência de tipos podem ser feitas ao mesmo tempo (veja Lab. 05).

Production			Syntax-Directed Actions
Expr	→	*Expr* + *Term*	{ $$ ← $MakeNode_2$ (plus, $1, $3); $$.type ← $\mathcal{F}_+$($1.type, $3.type) }
	|	*Expr* − *Term*	{ $$ ← $MakeNode_2$(minus, $1, $3); $$.type ← $\mathcal{F}_-$($1.type, $3.type) }
	|	*Term*	{ $$ ← $1 }
Term	→	*Term* × *Factor*	{ $$ ← $MakeNode_2$(times, $1, $3); $$.type ← $\mathcal{F}_\times$($1.type, $3.type) }
	|	*Term* ÷ *Factor*	{ $$ ← $MakeNode_2$(divide, $1, $3); $$.type ← $\mathcal{F}_\div$($1.type, $3.type) }
	|	*Factor*	{ $$ ← $1 }
Factor	→	(*Expr*)	{ $$ ← $2 }
	|	num	{ $$ ← $MakeNode_0$(number); $$.text ← *scanned text*; $$.type ← *type of the number* }
	|	name	{ $$ ← $MakeNode_0$(identifier); $$.text ← *scanned text*; $$.type ← *type of the identifier* }

Aspectos Interprocedurais de Inferência de Tipos

- A inferência de tipos para expressões depende de todas as funções que formam o programa.
- Mesmo a mais simples das LPs permite chamadas de funções em expressões.
- O compilador deve verificar todas as chamadas para garantir que os tipos dos parâmetros reais são compatíveis com os tipos dos parâmetros formais.
- Mesma verificação deve ser feita para o(s) valor(es) de retorno da função.
- Compilador precisa da assinatura da função. Exemplo: função `strlen` do C tem a seguinte assinatura:

$$\texttt{strlen} : \texttt{char*} \rightarrow \texttt{unsigned int}$$

- Casos de polimorfismo paramétrico como abaixo complicam mais a tarefa.

$$\texttt{filter} : (\alpha \rightarrow \texttt{bool}) \times \text{list of } \alpha \rightarrow \text{list of } \alpha$$

TS para Aumentar a Expressividade da LP

- O uso de um TS permite a inclusão de características na LP que são impossíveis de especificar na CFG.
- Exemplo: sobrecarga de operadores. Significado (semântica) da operação depende dos tipos dos operandos.
- Exemplo de uma linguagem não-tipada: BCPL.
  - Só possui o tipo célula que é uma sequência de bits.
  - Interpretação dos bits é determinada pelo operador aplicado à célula.
  - BCPL usa `+` para soma de inteiros e `#+` para soma de números de ponto-flutuante.
  - Ambas expressões `a+b` e `a#+b` são válidas, e nenhuma realiza conversão dos operandos.

TS para Aumentar a Expressividade da LP

- Por outro lado, mesmo as mais antigas LPs tipadas (FORTRAN) usam sobrecarga para especificar comportamento mais complexo.
- Outras amenidades permitidas pelo TS.
  - Em C: informação de tipo determina o efeito de incrementar um ponteiro.
  - Em LPs OO: despacho dinâmico de métodos depende do tipo do objeto invocador do método.
  - *Poor man's parametric polymorphism.*

TS para Geração de Código Eficiente

- Um TS bem projetado provê informações detalhadas de todas as expressões do programa.
- Essas informações podem ser usadas para gerar código mais eficiente.
- Exemplo: geração de código para soma em FORTRAN77.

Type of			Code
a	b	a + b	
integer	integer	integer	iADD r_a, r_b ⇒ r_{a+b}
integer	real	real	i2f f_a ⇒ r_{a_f} fADD r_{a_f}, r_b ⇒ r_{a_f+b}
integer	double	double	i2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
real	real	real	fADD r_a, r_b ⇒ r_{a+b}
real	double	double	r2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
double	double	double	dADD r_a, r_b ⇒ r_{a+b}

TS para Geração de Código Eficiente

- Em uma LP aonde os tipos não podem ser totalmente determinados de forma estática, o compilador precisa gerar código para verificação dinâmica.
- Isso garante a segurança do programa em tempo de execução, mas adiciona um *overhead* significativo.
- Verificação dinâmica de tipos requer que toda variável tenha um campo *tag*, o tipo naquele momento.
- Gasta mais memória, performance piora.
- Exemplo de código de verificação dinâmica de tipos no próximo slide.

TS para Geração de Código Eficiente

```
// partial code for "a+b ⇒ c"
if (tag(a) = integer) then
     if (tag(b) = integer) then
          value(c) = value(a) + value(b);
          tag(c) = integer;
     else if (tag(b) = real) then
          temp = ConvertToReal(a);
          value(c) = temp + value(b);
          tag(c) = real;
     else if (tag(b) = ...) then
          // handle all other types ...
     else
          signal runtime type fault
else if (tag(a) = real) then
     if (tag(b) = integer) then
          temp = ConvertToReal(b);
          value(c) = value(a) + temp;
          tag(c) = real;
     else if (tag(b) = real) then
          value(c) = value(a) + value(b);
          tag(c) = real;
     else if (tag(b) = ...) then
          // handle all other types ...
     else
          signal runtime type fault
else if (tag(a) = ...) then
     // handle all other types ...
else
     signal illegal tag value;
```

Verificação estática de tipos elimina o código acima. Sempre preferível do ponto de vista de eficiência mas nem sempre possível (depende da LP).

Aula 04 – Análise Semântica - Sistemas de Tipos

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**