ANÁLISE SENSÍVEL AO CONTEXTO

ÉDER PORFÍRIO

1. CONTEXTUALIZAÇÃO

- Um programa de entrada gramaticalmente correto ainda pode conter sérios erros que impediriam a compilação;

- Para detectá-los, um compilador realiza outro nível de verificação que envolve a consideração de cada comando em seu contexto real;

- Essas verificações encontram erros e tipo e de concordância;

2. INTRODUÇÃO

- A tarefa final de um compilador é traduzir um programa de entrada para um formato que possa ser executado diretamente na máquina alvo;

- Para essa finalidade, ele precisa de conhecimento sobre o programa de entrada que vai bem além da sintaxe;

- O compilador precisa acumular uma grande base de conhecimento sobre a computação detalhada que está codificada no programa de entrada;

2. INTRODUÇÃO

- **É necessário saber:**
  - Quais valores são representados;
  - Onde residem;
  - Como fluem de um nome para o outro;
  - Entender a estrutura da computação;
  - Analisar como o programa interage com arquivos e dispositivos externos;

- Tudo isso pode ser verificado no código fonte usando conhecimento contextual.

3. ROTEIRO CONCEITUAL

Para acumular conhecimento contextual o compilador precisa analisar o código de uma forma que não seja pela sintaxe;

Precisaremos de abstrações que representem algum aspecto do código:

- Sistema de tipos;
- Mapa de armazenamento;
- Grafo de fluxo de controle.

Precisamos entender o espaço de nomes no programa:

- Os tipos de dados representados;
- Os tipos de dados que podem ser associados a cada nome da expressão;
- Mapeamento do nome;
- Fluxo de controle.

4. VISÃO GERAL

Considere um único nome (x) usado no programa sendo compilado. Antes de criar o código executável é preciso responder a muitas perguntas:

1. Que tipo de valor está associado a x?
2. Qual é o tamanho de x?
3. Se x é um procedimento, que argumentos ele utiliza?
4. Por quanto tempo o valor de x pode ser preservado?
5. Quem é responsável por alocar espaço para x (e inicializa-lo)?

4. VISÃO GERAL

O compilador precisa obter as respostas as perguntas a partir do programa fonte e das regras da linguagem-fonte;

Em algumas linguagens, a maior parte dessas perguntas pode ser respondida examinando-se a declaração de x;

Se a linguagem não possuir declarações, o compilador precisa obter essas informações analisando o programa;

4. VISÃO GERAL

Muitas (se não todas) estão fora da sintaxe.

Exemplo:

$\mathrm{x} \leftarrow y$ e $\mathrm{x} \leftarrow z$

Se x e y são inteiro e z é um string o compilador pode ter que emitir um código diferente para cada uma das sentenças.

A análise do significado está no âmbito da análise sensível ao contexto.

5. INTRODUÇÃO AOS SISTEMAS DE TIPOS

A maior parte das linguagens de programação associa uma coleção de propriedades a cada valor de dados;

Chamaremos esta coleção de propriedades de *tipo* do valor;

O tipo especifica um conjunto de propriedades mantidas em comum para todos os valores desse tipo;

SISTEMA DE TIPOS: conjunto de tipos + regras que utilizam tipos para especificar o comportamento do programa.

5.1 FINALIDADE DOS SISTEMAS DE TIPOS

O sistema de tipos cria um segundo vocabulário para descrever a forma e o comportamento dos programas válidos;

Em um compilador o sistema de tipos é usado para três finalidades específicas:

1. A garantia da **segurança** em tempo de execução;
2. A melhoria da **expressividade**;
3. A geração de um código melhor (**eficiência**);

5.2 VERIFICAÇÃO DE TIPO

Para evitar o overhead da verificação de tipo, compilador precisa:

- Atribuir um tipo a cada nome e a cada expressão;
- Verificar esses tipos para garantes que estão sendo usados em um contexto válido;

- Atividade de garantir que operandos e operadores são de tipos compatíveis;
- Tipo compatível:
  - Legal
  - É permitido dentro das regras da LP - convertido (coerção) - Ex: IntToFloat

5.2 VERIFICAÇÃO DE TIPO

- **TIPOS ESTÁTICOS** → Vinculação feita de maneira estática → verificação na compilação

- **TIPOS DINÂMICOS** → Vinculação feita de maneira dinâmica → verificação em tempo de execução

5.3 TIPAGEM FORTE

- FORTEMENTE TIPADA → Erros de tipos são sempre detectados;
- Requer que tipos de dados sejam determinados em tempo de compilação ou em tempo de execução;
- IMPORTANTE: Habilidade de detectar usos incorretos de variáveis (erros de tipos)

5.4 COMPONENTES DE UM SISTEMA DE TIPOS

Um sistema de tipos para uma linguagem moderna típica tem quatro componentes principais:

1. Conjunto de tipos básicos;
2. Regras para construir novos tipos a partir dos existentes;
3. Um método para determinar se dois tipos são compatíveis;
4. Regras para inferir o tipo de cada expressão da linguagem-fonte.

5.4.1 TIPOS DE DADOS PRIMITIVOS

- Tipos de dados não definidos em termos de outros
  - Alguns são reflexo do hardware;
  - Outros requerem suporte externo ao hardware;

- Utilizados com um ou mais construtores de tipos → fornecer tipos estruturados

5.4.1 TIPOS DE DADOS PRIMITIVOS

- Tipos Numéricos
- Tipos Booleanos
- Tipos Caracteres

5.4.1.1 TIPOS NUMÉRICOS

- Desempenham papel central entre os tipos suportados

- Muitas LP’s possuem apenas tipos numéricos

- Principais Tipos:
  - Inteiros
  - Ponto flutuante
  - Decimais

INTEIRO

- Tipo mais comum
  - Não possuem parte decimal;
  - Podem ser positivos e negativos;

- Maioria dos tipos inteiros são suportados diretamente pelo hardware;

- Inicialmente ocupava 2 Bytes.
- Operações Aritméticas → Valores Grandes → Inteiros Longos

INTEIRO

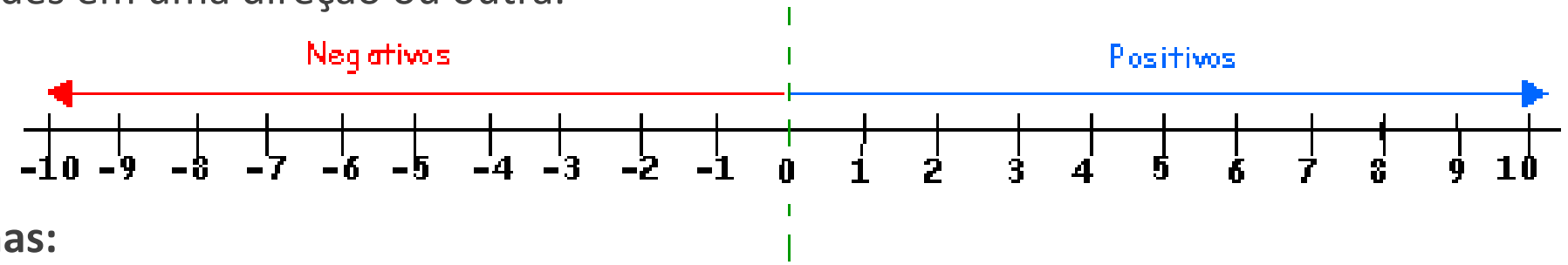
- **INTEIRO COM SINAL**
  - **SINAL MAGNITUDE** → Cadeia de bits, com um dos bits representando o sinal;
  - **COMPLEMENTO DE DOIS** → Aritmética computacional;

SINAL MAGNITUDE

- Representação Básica → um sinal (+ ou -) é colocado antes do valor do número;

- O sinal representa a posição relativa do número (positivo ou negativo) e o valor representa a magnitude;

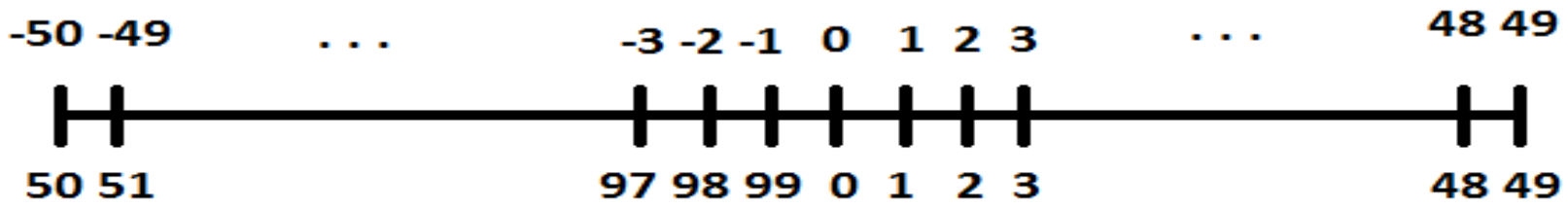
- Realizar adições e subtrações com números inteiros pode ser descrito como descolar certo número de unidades em uma direção ou outra.



- **Problemas:**
  - Bit Adicional
  - Duas representações para o zero

NÚMERO DE TAMANHO FIXO

Se restringimos apenas um número fixo de valores, podemos representar números com valores apenas inteiros, onde a metade deles representa números negativos.



Negativo (I) = 10^K – I, onde k é o número de dígitos .

Ex:

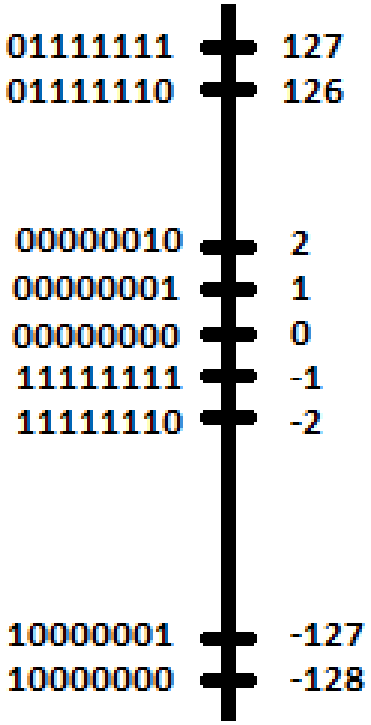
-3 = 10^2 – 3 = 97

-3 = 10^3 – 3 = 997

(Complemento de 10)

COMPLEMENTO DE 2

Assumindo que um número tenha que ser representado por 8 bits



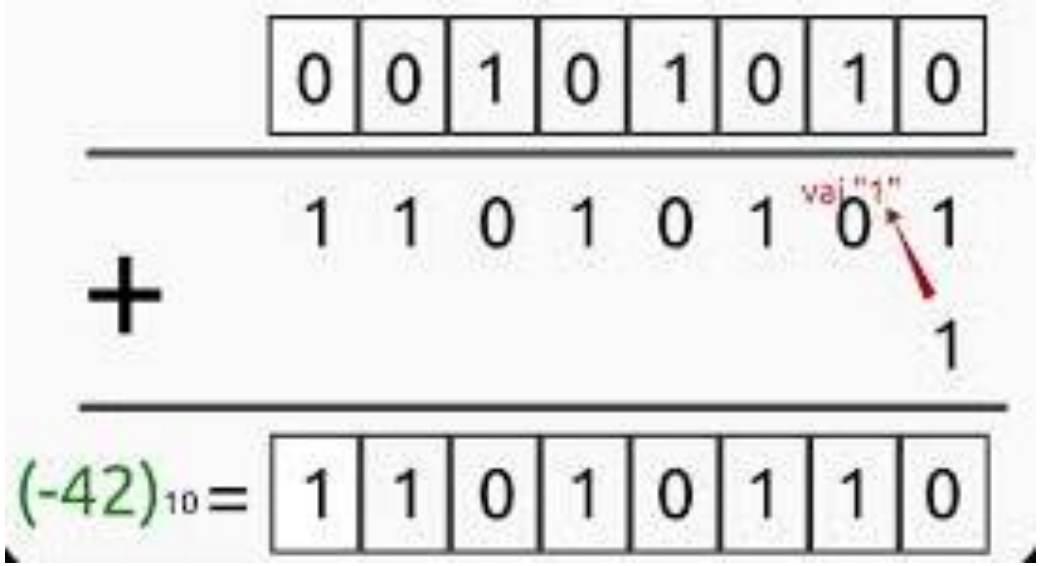
- A formula do complemento de 10 fica substituída por 2?

- Negativo (I) = $2^K - I$

-(2) = 2^8 -2 = 256 -2 = 254

Complemento de 2

Existe uma forma mais simples de calcular o complemento de dois:



TRANSBORDAMENTO (overflow)

- Ocorre quando o valor que calculamos não cabe no número de bits que alocamos para o resultado.

$$01111111 \rightarrow 127$$

$$+\ 00000011 \rightarrow 3$$

$$10000010$$

- Em nosso esquema 10000010 representa -126 não 130.

- Esse é um tipo clássico de problema que encontramos ao mapear o mundo infinito em uma máquina finita.

PONTO FLUTUANTE

- Modelam números reais;

- **SINTAXE:**
  - Frações Exponenciais (notação científica)

- **PRICIPAL REPRESENTAÇÃO:**
  - IEEE Float-Point Standard 754

PROBLEMA:

- Perda de precisão por expressões aritméticas;
- Representação são apenas aproximações: **π – e**

- **TAMANHO:**
  - float
  - double

DECIMAL

- Computadores de grande porte possuem suporte de hardware para decimais
- Funcionamento: Armazenam um número fixo de números decimais – ponto decimal em uma posição fixa
- Vantagem:
  - Capazes de armazenar precisamente valores decimais (dentro de uma faixa)
- Desvantagens:
  - Restrita (nenhum expoente é usado)
  - Precisão em memória é dispendiosa

5.4.1.2 TIPOS BOOLEANOS

- Tipos de dados mais simples;
- Faixa de valores apenas dois elementos: verdadeiro e falso
- Geralmente usados para representar escolhas.
  - Poderiam ser realizadas com outros tipos (inteiros por exemplo)
  - Tipo booleano aumenta a Legibilidade

5.4.1.3 CARACTERES

- São armazenados como codificações numéricas

- Tradicionalmente → ASCII (não mais usada)
- Atualmente → UCS-4 ou UTF-32.
  - Consórcio: Unicod e ISO

Tabela ASCII

Binário	Decimal	Hexa	Glifo
0010 0000	32	20	
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3

Binário	Decimal	Hexa	Glifo
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S

Binário	Decimal	Hexa	Glifo
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s

UNICODE

- ASCII – 256 Caracteres (Não são suficientes para o uso internacional)

- UNICODE – Representar todos os caracteres de todas as línguas usadas no mundo (Codificação ainda em evolução)

- Sistema Unicode
  - Usado por muitas linguagens de programação e sistemas computacionais atuais


UNICODE

5.4.2 TIPOS COMPOSTOS E CONSTRUÍDOS

Os tipos básicos oferecem abstração para os tipos tratados diretamente pelo hardware;

O programas lidam com estruturas mais complexas como: grafos, árvores, tabelas, arrays, listas e pilhas;

A capacidade de construir novos tipos para esses objetos compostos e agregados é um recurso essencial para as LP’s;

Unir esse organização ao sistema de tipos melhora a capacidade do compilador detectar programas malformados;

5.4.2.1 CADEIA DE CARACTERES

- SEQUÊNCIA DE CARACTERES → Usados para rotular entradas/saídas

- Tipo essencial para LP's que desejam manipular caracteres

- Duas questões de projeto:
  1. Cadeia é um tipo especial de vetor ou um tipo primitivo?
  2. As cadeias devem ter tamanho estático ou dinâmico?

5.4.2.2 ARRAYS

- Agregação homogênea de elementos

- Elementos específicos são referenciados por meio de um mecanismo sintático e dois níveis:
  - Primeira parte: Nome do Agregado;
  - Segunda parte: seletor – um ou mais itens (índices)

- Sintaxe Universal:
  - Nome da matriz, seguida por uma lista de índices (envolvidos por parênteses ou colchetes)

Arrays

A declaração em C `int a[100][200];` reserva um espaço 100x200=20.000 inteiros e garante que eles possam ser endereçados usando o nome `a`.

As referências `a[13][20]` e `a[25][32]` acessam locais de memória distintos e independentes.

A propriedade essencial de um *array* é que o programa pode computar nomes para cada um de seus elementos usando números como subscritos.

O suporte para operações com *array* varia bastante de LP para LP.

5.4.2.3 REGISTROS

- Agregado de elementos de dados
  - Usado para modelar coleções de dados que não são do mesmo tipo;
  - Elementos individuais identificados por nomes e acessados por deslocamento a partir do início da estrutura;

Exemplo C:

```
Struct ficha_aluno {
   char nome[50];
   char disciplina [30];
   float AP1;
   float AP2;
   float AP3;
}
```

5.4.2.4 TIPOS ENUMERADOS

Muitas linguagens permitem que os programadores criem um tipo que contém um conjunto específico de valores constantes.

O tipo enumerado, permite que o programador use nomes autodocumentáveis para pequenos conjuntos constantes.

Exemplo de Sintaxe :

```
Enum DiaSemana {Segunda, Terça, Quarta, Quinta, Sexta, Sábado, Domingo};
```

O compilador mapeia cada elemento de um tipo enumerado para um valor distinto.

5.4.2.5 PONTEIROS

São endereços de memória abstratos, que permitem que o programador manipule quaisquer estruturas de dados;

Ponteiros permitem que um programa salve um enderenço e mais tarde examine o objeto que ele endereça;

Ponteiros são criado quando os objetos são criados:

- new em Java
- malloc em C

Algumas linguagens oferecem um operador que retorna o endereço do um objeto. Operador & em C.

5.4.3 EQUIVALÊNCIA DE TIPOS

- Componente crítico de qualquer sistema de tipos
- Definida para dar suporte a verificação de tipos

- Regras de Compatibilidade
  - Ditam os operandos aceitáveis para cada operador ;
  - Especificam possíveis erros de tipos.

- Regras simples e rígidas → escalares pré-definidos (inteiros)
- Regras Complexas → tipos estruturados (matrizes e registros)

Equivalência de Tipos

Historicamente dois tipos de equivalência tem sido testados.

Equivalência de nomes: afirma que dois tipos são equivalentes se e somente se tiverem o mesmo nome;

Equivalência estrutural: declara que dois tipos são equivalentes se e somente se ambos tiverem a mesma estrutura;

Cada política tem seus pontos franco e pontos fortes.

Trabalho – Equipe 5 membros

1. Escolha uma linguagem de programação e descreva os tipos básicos de seu sistema de tipos. Que regras e construções a linguagem permite para construir tipos agregados? Ele fornece algum mecanismo para criar um procedimento que use um número variável de argumentos?

2. Que tipo de informação o compilador deve ter para garantir a segurança de tipos nas chamadas de procedimento? Esboce um esquema.

6. GRAMÁTICA DE ATRIBUTOS

Um formalismo que tem sido proposto para realizar análise sensível ao contexto é a **Gramática de Atributos** ou **Gramática Livre de Contexto Atribuída.**

Consiste em uma gramática livre de contexto aumentada por um conjunto de regras que especificam computações.

Alguns autores chamam de: Definição definida pela sintaxe.

A regra associa o atributo a um símbolo da gramática.

GRAMÁTICA DE ATRIBUTOS

Cada ocorrência de um símbolo da árvore sintática tem uma ocorrência correspondente do atributo.

As regras são funcionais; não definem uma ordem de avaliação específica, e definem o valor de cada atributo de forma única.

Gramática de Atributos - Exemplo

Considere a gramática para geração de números binários com sinal.

$$P = \begin{Bmatrix} Number \rightarrow Sign\ List \\ Sign \rightarrow + \mid - \\ List \rightarrow List\ Bit \mid Bit \\ Bit \rightarrow 0 \mid 1 \end{Bmatrix}$$

$$T = \{+, -, 0, 1\}$$
$$NT = \{Number, Sign, List, Bit\}$$
$$S = \{Number\}$$

Para nossa versão atribuída, os seguintes atributos são necessários:

Símbolo	Atributos
Number	value
Sign	negative
List	position, value
Bit	position, value

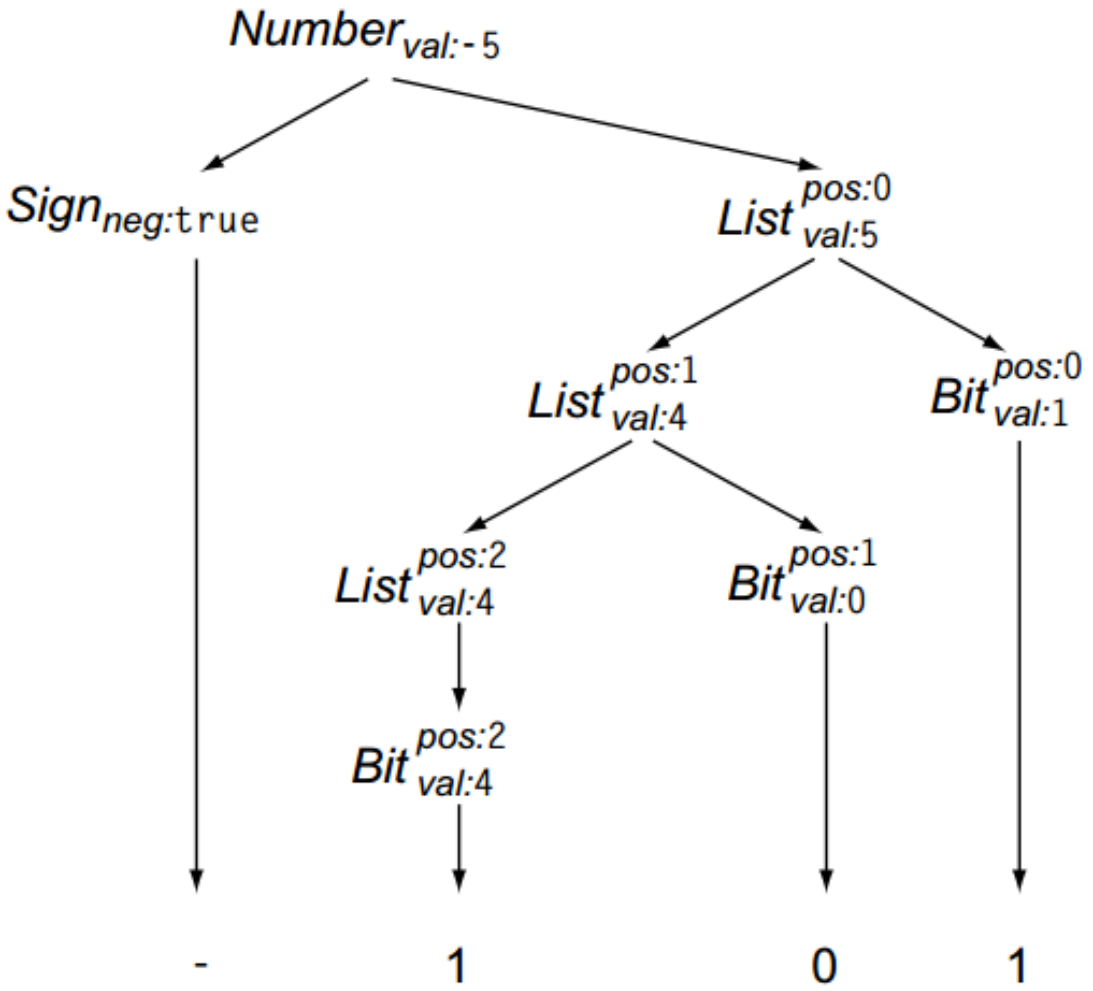
Gramática de Atributos - Exemplo

	Padrão	Regras de Atribuição
1	$Number \rightarrow Sign\ List$	**List.**position ← 0 if **Sign.**negative then **Number.**value ← - **List.**value else **Number.**value ← **List.**value
2	$Sign \rightarrow +$	**Sign.**negative ← false
3	$Sign \rightarrow -$	**Sign.**negative ← true
4	$List \rightarrow Bit$	**Bit.**position ← **List.**position **List.**value ← **Bit.**value
5	$List_0 \rightarrow List_1\ Bit$	**List$_1$.**position ← **List$_0$.**position + 1 **Bit.**position ← **List$_0$.**position **List$_0$.**value ← **List$_1$.**value + **Bit.**value
6	$Bit \rightarrow 0$	**Bit.**value ← 0
7	$Bit \rightarrow 1$	**Bit.**value ← $2^{\text{Bit.position}}$

Árvore de derivação

As regras de atribuição definem *Number*.value como valor decimal para a string de entrada binária.

Por exemplo a sting -101 causa a seguinte atribuição.



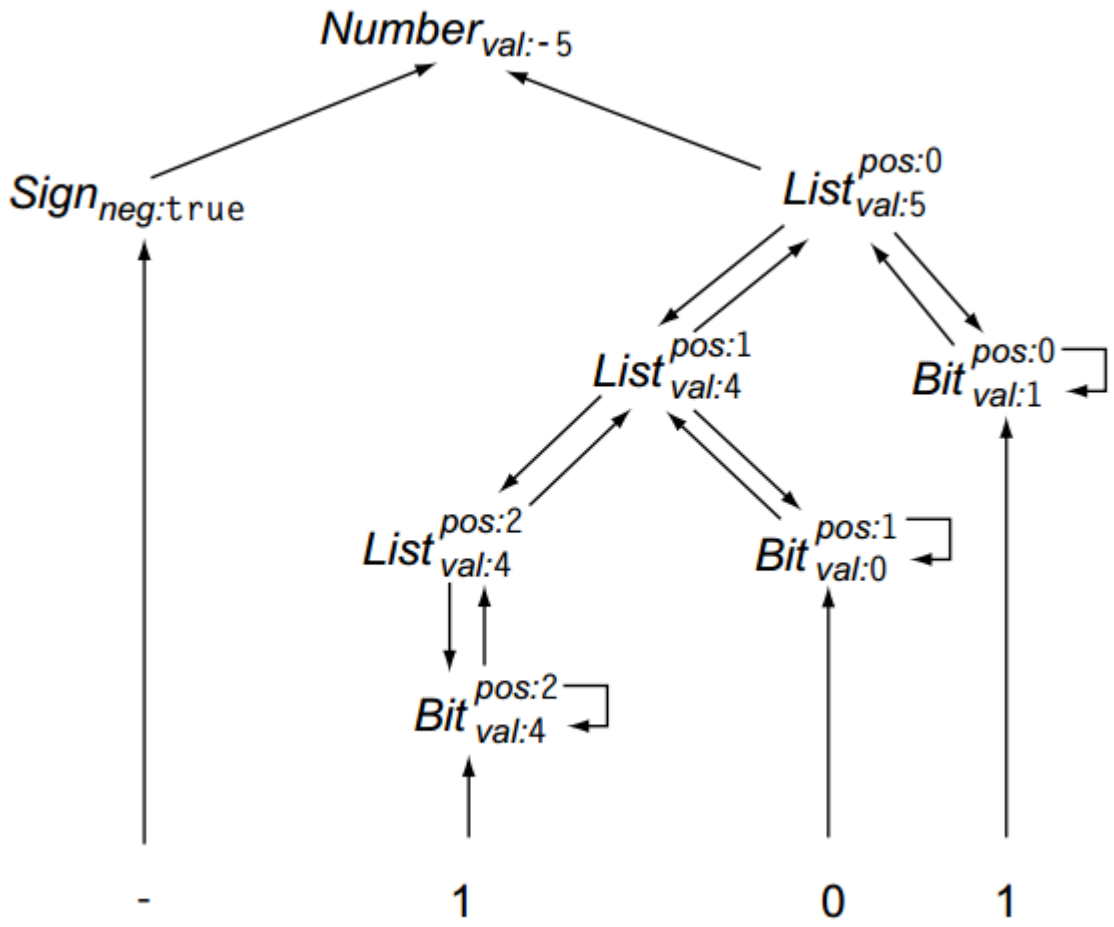
Grafo de Dependência

Cada regra define implicitamente um conjunto de dependências;

O atributo sendo definido depende de cada argumento da regra;

Tomadas sobre a árvore de derivação inteira essas dependências formam um grafo de dependências de atribut

As arestas de no grafo seguem o fluxo de valores para a avaliação de uma regra.



Tipos de Atributos

Atributo Sintetizado:

Atributo definido totalmente em termos do atributo do nó, seus filhos e constantes;

Atributo Herdado:

Atributo definido totalmente em termos de atributos próprios do nó e daqueles de seus irmãos ou seu pai na árvore de derivação (além das constantes)

Avaliação de Atributos

- Qualquer esquema para avaliar atributos precisa respeitar os relacionamentos codificados implicitamente no grafo de dependência de atributo.

- Cada atributo deve ser definido por alguma regra. Se essa regra depender da avaliação de outros atributos, não poderá ser avaliada até que todos os valores tenham sido definidos.

- Se não depender então deve produzir o seu valor a partir de uma constante ou de alguma fonte externa.

Implementação - Gramática de Atributos

Para criar uma gramática de atributos deve-se determinar um conjunto de atributos para cada símbolo da gramática e projetar um conjunto de regras para calcular seus valores.

Para criar uma implementação, é necessário criar um avaliador, o que pode ser feito com:

- *Um programa ad hoc;*
- *Um gerador de avaliador;*

6.1 Métodos de Avaliação

O modelo de gramática de atributos tem uso prático somente se pudermos criar avaliadores que interpretem as regras para avaliar um exemplo d problema automaticamente.

Muitos técnicas de avaliação foram propostas na literatura. Em geral, todas encontram-se em uma dessas três categorias.

1. Métodos Dinâmicos
2. Métodos Desatentos
3. Métodos baseados em regras

6.1.Métodos de Avaliação

1. **Métodos Dinâmicos**
   - Utiliza a estrutura de uma particular árvore de derivação atribuída para determinar a ordem de avaliação
2. **Métodos Desatentos**
   - A ordem de derivação independe da gramática de atributos e da particular árvore de derivação atribuída;
   - Presume-se que o projetista seleciona um método considerado o mais apropriado
3. **Métodos baseados em regras**
   - Baseia-se e uma análise estática da gramática de atributos;

Caminhamento em Profundidade

Uma definição dirigida pela sintaxe não impõe qualquer ordem específica para avaliação dos atributos;

Qualquer ordem de avaliação que compute um atributo a, após computar todos os atributos dos quais a dependa é aceitável.

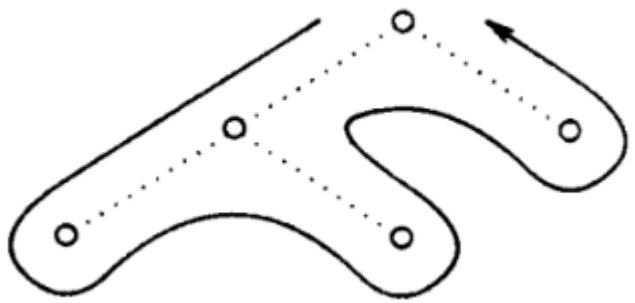
Em, geral, durante o caminhamento da árvore gramatical, alguns atributos terão que ser avaliados quando um nó for atingido pela primeira vez, outros quando todos os seus filhos tiverem sido visitados ou em algum ponto entre as visitas aos filhos daquele nó.

Caminhamento em profundidade

Inicialmente trataremos da avaliação das regras semânticas, numa ordem pré-determinada.

O Caminhamento em profundidade se inicia na raiz e visita recursivamente seus filhos em uma ordem da esquerda para a direita.

```
procedimento visitar(n:nó);
início
        para cada filho m de n, da esquerda para a direita faça
                visitar(m);
        avaliar as regras semânticas do nó n
fim
```



6.2 CIRCULARIDADE

Uma gramática de atributo é circular se puder, para algumas entradas, criar um grafo de dependência cíclico;

As regras de atribuição permitem que uma regra referencia seu próprio resultado de forma direta ou indireta.

Os modelos de avaliação falham quando a grafo de dependência contém um ciclo.

Uma falha desse tipo em um compilador pode gerar sérios problemas.

Circularidade – Técnicas de Tratamento:

1. **Evitamento.**
   - O construtor de compiladores pode restringir a gramática de atributo a uma classe que não fará surgir grafos de dependência circulares.

2. **Avaliação.**
   - O construtor de compiladores pode usar um método de avaliação que concede um valor a cada atributo, mesmo para aqueles envolvidos em ciclos;
   - O avaliador pode percorrer o ciclo e atribuir valores apropriados ou valores default.

6.3 Exemplos Estendidos

Para melhorar o entendimento da gramática de atributos serão apresentados dois exemplos práticos:

1. Inferência de tipos de expressão

2. Estimador de tempo de execução

3. Notação Posfixa

Exemplo 1: Inferência de Tipos

Qualquer compilador que tente gerar código eficiente para uma linguagem tipada, precisa encarar o problema de inferir tipos para cada expressão do programa;

Este problema conta, inerentemente com a informação sensível ao contexto.

O tipo relacionado a um ID ou a um NUM depende da sua identidade e não da categoria sintática;

Exemplo 1: Inferência de Tipos

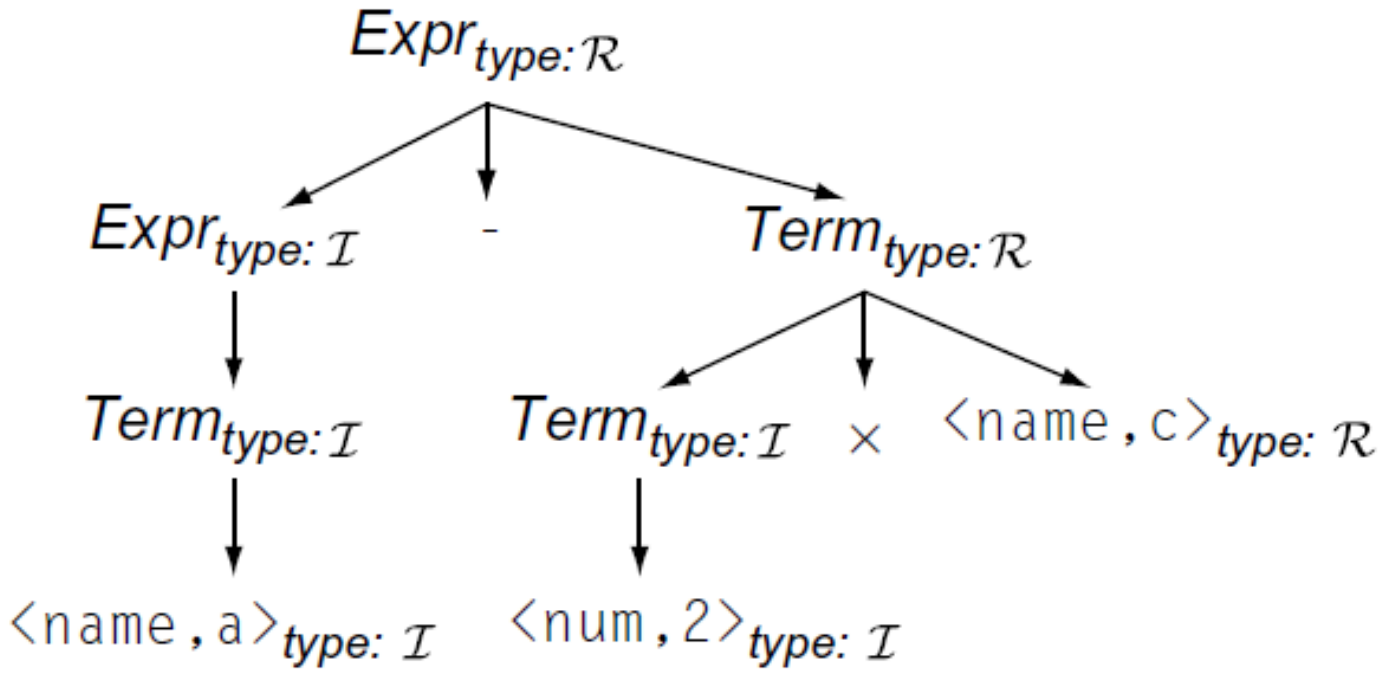
Suponha que as expressões sejam representadas como árvores sintáticas, e que qualquer nó representando NAME ou NUM já tenha um atributo *type.*

Para cada operador precisaremos de uma função que mapeio dois tipos de operando para um tipo de resultado.

+	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 1: Inferência de Tipos

Regras de atribuição – Arvore de derivação para a string: a – 2 x c



Exemplo 1: Inferência de Tipos

O nós folha têm seus atributos type inicializados de modo apropriado. O restante dos atributos é definido pela regras da tabela abaixo:

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*

Exemplo 1: Inferência de Tipos

Uma visão mais aprofundada das regras de atribuição mostra que todos os atributos são sintetizados.

Todos os atributos fluem de um filho para o seu pai na árvore de derivação;

*Essa gramática são chamadas de **Gramáticas S-Atribuídas** e possui um esquema de avaliação simples **baseado em regras**;*

Encaixa-se com a análise sintática bottom-up. Cada regra pode ser avaliada quando o parser reduz o lado direito correspondente.

Exemplo 2: Estimador de Tempo de Execução

Considere o problema de estimar o tempo de execução de uma sequência de instruções de atribuição.

Podemos Gerar uma sequência de atribuições acrescentando três novas produções a nossa gramática clássica:

$$
\begin{aligned}
\textit{Block} &\rightarrow \textit{Block Assign} \\
&\mid \textit{Assign} \\
\textit{Assign} &\rightarrow \texttt{name} = \textit{Expr;}
\end{aligned}
$$

Exemplo 2: Estimador de Tempo de Execução

Production			Attribution Rules
$Block_0$	→	$Block_1$ *Assign*	{ $Block_0$. *cost* ← $Block_1$. *cost* + *Assign* . *cost* }
	|	*Assign*	{ $Block_0$. *cost* ← *Assign* . *cost* }
Assign	→	name = *Expr;*	{ *Assign* . *cost* ← *Cost*(store) + *Expr* . *cost* }
$Expr_0$	→	$Expr_1$ + *Term*	{ $Expr_0$. *cost* ← $Expr_1$. *cost* + *Cost*(add) + *Term* . *cost* }
	|	$Expr_1$ − *Term*	{ $Expr_0$. *cost* ← $Expr_1$. *cost* + *Cost*(sub) + *Term* . *cost* }
	|	*Term*	{ $Expr_0$. *cost* ← *Term* . *cost* }
$Term_0$	→	$Term_1$ × *Factor*	{ $Term_0$. *cost* ← $Term_1$. *cost* + *Cost*(mult) + *Factor* . *cost* }
	|	$Term_1$ ÷ *Factor*	{ $Term_0$. *cost* ← $Term_1$. *cost* + *Cost*(div) + *Factor* . *cost* }
	|	*Factor*	{ $Term_0$. *cost* ← *Factor* . *cost* }
Factor	→	(Expr)	{ *Factor* . *cost* ← *Expr* . *cost* }
	|	num	{ *Factor* . *cost* ← *Cost*(loadI) }
	|	name	{ *Factor* . *cost* ← *Cost*(load) }

Exemplo 2: Estimador de Tempo de Execução

- **Gramática S-atribuída** - gramática utiliza apenas atributos sintetizados;
- A estimativa aparece no atributo cost no nó Bloco mais alto da árvore;
- Os custos são calculados de baixo para cima.
- A função Cost retorna a latência de uma dada operação ILOC.

Exercício Proposto

1. Construa uma árvore de derivação para a expressão que preferir e apresente como é feita a estimação do tempo de execução.

Atribua valores para a função Cost.

Exemplo 3 - Notação Posfixa

Uma notação posfixa para uma expressão E pode ser definida indutivamente como se segue:

REGRAS:

1. Se E é uma variável ou uma constante.
   - NPF = E
2. Se é E é uma expressão da forma E_1 op E_2
   - NPF E_1' E_2'op

   Em que E_1' e E_2' são a notação posfixa de E_1 e E_2 respectivamente
3. Se E for uma expressão da forma (E_1)
   - NPF = E = NPF E_1

Exemplo

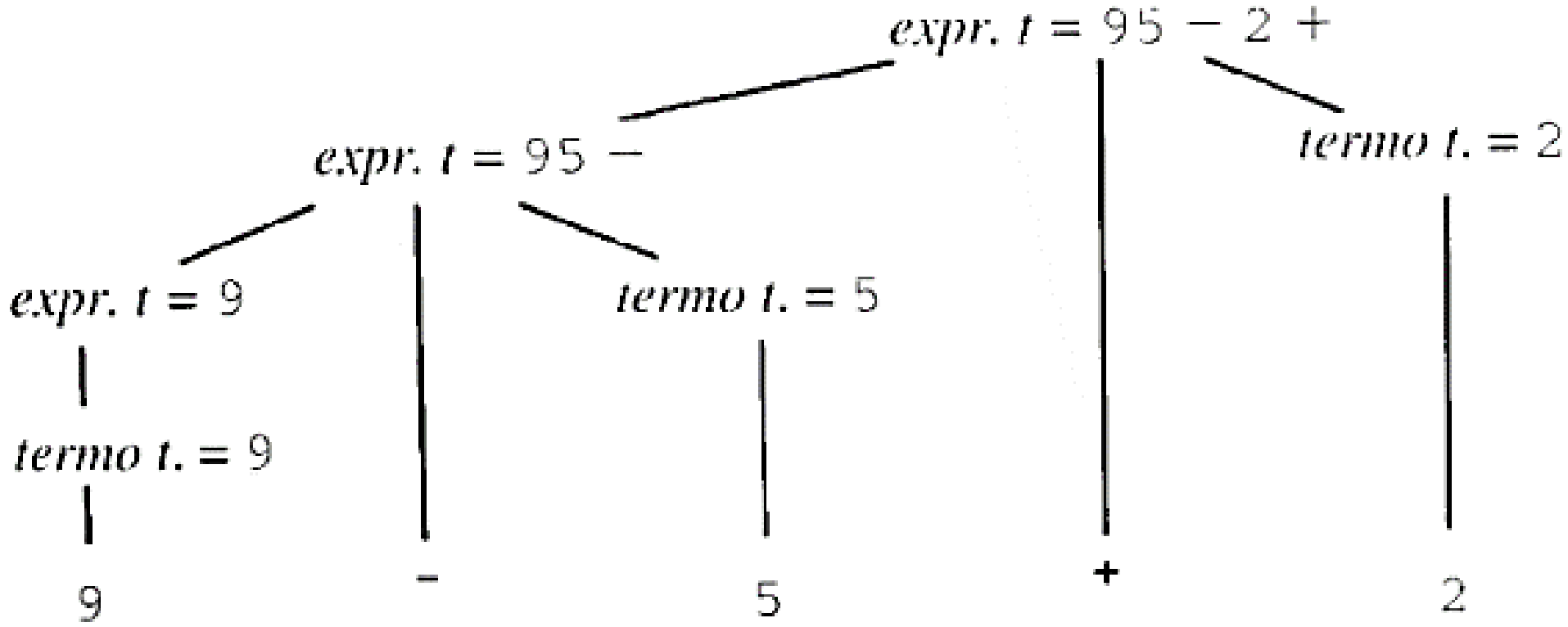
a := b * c + b

a b c * b + **atrib**

Exemplo 3 - Notação Posfixa (Tabela)

PRODUÇÃO	REGRA SEMÂNTICA
$expr \rightarrow expr_1 + termo$	$expr.t := expr_1.t \,\|\|\, termo.t \,\|\, '+'$
$expr \rightarrow expr_1 - termo$	$expr.t := expr_1.t \,\|\|\, termo.t \,\|\, '-'$
$expr \rightarrow termo$	$expr.t := termo.t$
$termo \rightarrow 0$	$termo.t := '0'$
$termo \rightarrow 1$	$termo.t := '1'$
...	...
$termo \rightarrow 9$	$termo.t := '9'$

Tabela – Notação Pósfixa (Árvore)



Problemas das Gramáticas de Atributos

1. *Tratamento de Informações não locais*
2. *Gerenciamento de espaço de armazenamento*
3. *Instanciação da árvore de derivação*
4. *Localização de respostas*

7. Tradução Dirigida pela sintaxe.

Gramática de Atributos serve de base para as técnicas *ad hoc* usadas para análise sensível ao contexto de muitos compiladores;

Sequência de ações que estão associadas as produções da gramática;

7. Tradução Dirigida pela sintaxe.

Ações → Trechos de códigos que são executados no momento da análise sintática;

Cada trecho (ação) está ligado diretamente a uma produção na gramática.

Esquema de Tradução

A posição à qual uma ação deve ser executada é mostrada envolvendo-a entre chaves e escrevendo-a no lado direito da produção.

Exemplo:

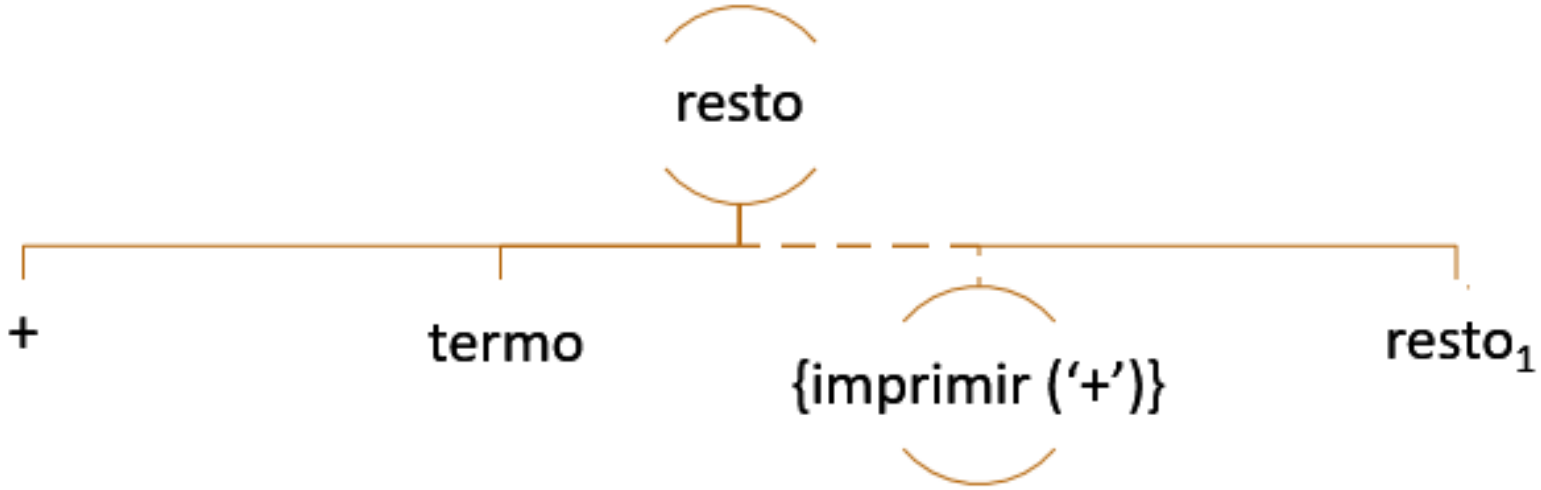
$resto \rightarrow + termo \{imprimir('+')\} resto_1$

Um esquema de tradução gera uma saída para cada sentença x.

A ação $\{imprimir('+')\}$ será realizada depois que a subárvore para termo seja percorrida, mas antes que o filho para $resto_1$ seja visitado.

Exemplo:

$resto \rightarrow + termo \{imprimir(' +')\} resto_1$



Emitindo uma tradução – Notação Posfixa

$$\boldsymbol{exp} \rightarrow \boldsymbol{exp} + \boldsymbol{termo}\ \{\boldsymbol{impirmir}\ (' + ')\}$$

$$\boldsymbol{exp} \rightarrow \boldsymbol{exp} - \boldsymbol{termo}\ \{\boldsymbol{impirmir}\ (' - ')\}$$

$$\boldsymbol{exp} \rightarrow \boldsymbol{termo}$$

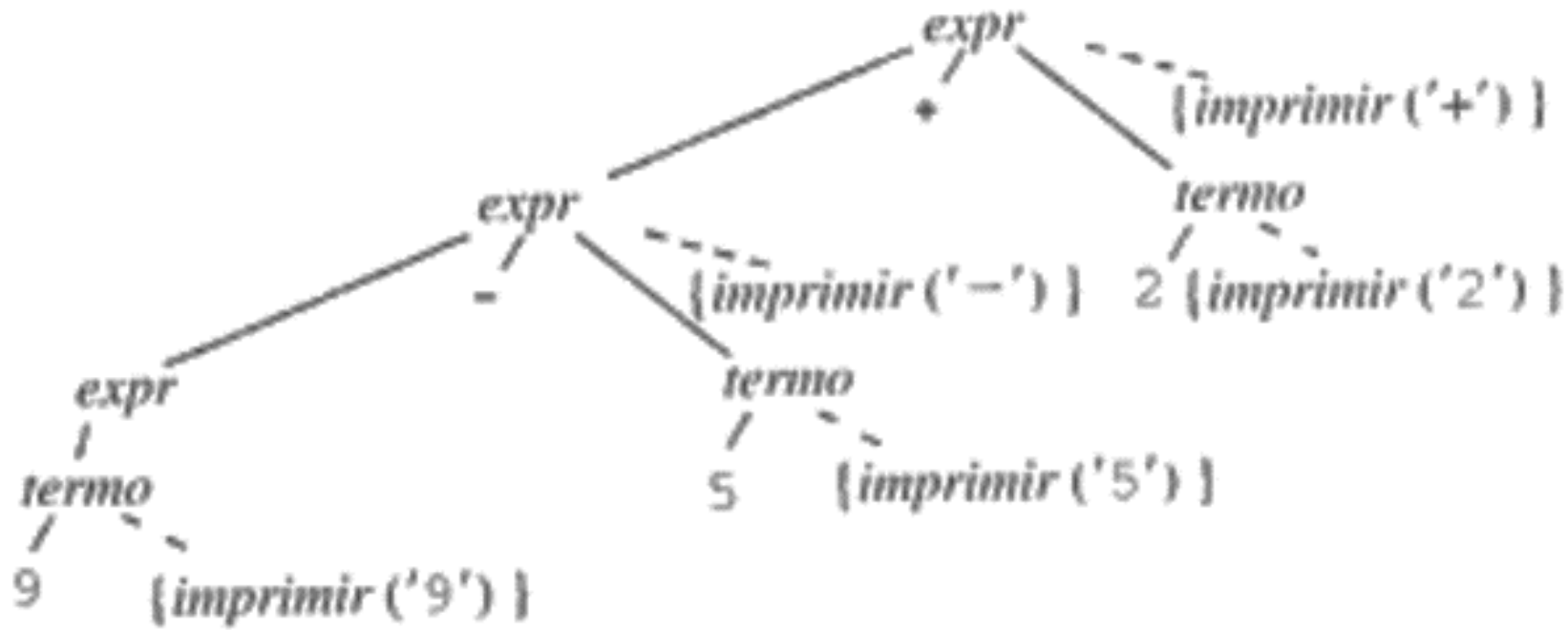
$$\boldsymbol{exp} \rightarrow \mathbf{0}\ \{\boldsymbol{imprimir}\ (\mathbf{'0'})\}$$

$$\boldsymbol{exp} \rightarrow \mathbf{1}\ \{\boldsymbol{imprimir}\ (\mathbf{'1'})\}$$

...

$$\boldsymbol{exp} \rightarrow \mathbf{9}\ \{\boldsymbol{imprimir}\ (\mathbf{'9'})\}$$

Emitindo uma tradução – Notação Posfixa



Tradução ad hoc dirigida pela sintaxe

Parser de Descida Recursiva: *acrescenta-se o código apropriado nas rotinas da análise sintática.*

Parser Bottom-up: *As ações são realizadas toda vez que o parser realiza a* ***ação reduzir***

Exemplo – Número binário com sinal

	Produção			Trecho de código
1	*Number*	→	*Sign List*	*Number*.`val` ← *Sign*.`val` × *List*.`val`
2	*Sign*	→	+	*Sign*.`val` ← *1*
3	*Sign*	→	-	*Sign*.`val` ← *-1*
4	*List*	→	*Bit*	*List*.`val` ← *Bit*.`val`
5	$List_0$	→	$List_1$ *Bit*	$List_0$.`val` ← *2* × $List_1$.`val` + *Bit*.`val`
6	*Bit*	→	0	*Bit*.`val` ← *0*
7	*Bit*	→	1	*Bit*.`val` ← *1*

Exemplo – Número binário com sinal

Cada símbolo da gramática tem um único valor associado → *val*.

O trecho de código para cada regra define o valor associado ao símbolo do lado esquerdo da regra.

- A regra 1 simplesmente multiplica o valor de *Sign* pelo de *List*.
- As regras 2, 3, 6 e 7 definem o valor de *Sign e Bit* de modo apropriado
- A regra 4 simplesmente copia o valor de *Bit* para *List*.
- O trabalho real ocorre na regra 5, que multiplica o valor acumulado dos bits iniciais (em *List*.*val*) por dois e depois soma ao próximo bit.

Exemplo – Número binário com sinal

Até aqui, isto parece ser muito semelhante a uma gramática de atributo.

Há duas simplificações-chave:

1. Os valores fluem em apenas um sentido, das folhas para a raiz;
2. Somente é permitido um único valor por símbolo da gramática.

O esquema apresentado calcula corretamente o valor do número binário com sinal. Deixando esse valor na raiz da árvore.

Tradução ad hoc dirigida pela sintaxe

Essas duas simplificações tornam possível um método de avaliação que funciona bem com um **parser *bottom-up***, como os **parsers LR(1)**.

Como cada trecho de código está associado ao lado direito de uma produção específica, o parser pode chamar a ação toda vez que efetuar uma redução.

Implementação da tradução *ad hoc* dirigida pela sintaxe

Para fazer a tradução *ad hoc* dirigida pela sintaxe funcionar, o parser precisa :

1. Incluir mecanismos para **passar valores**;
2. **Fornecer nomeação** conveniente e coerente;
3. Permitir **ações** que sejam **executadas em outros pontos na análise** sintática.

Comunicação entre ações

Para passar valores entre ações, o *parser* deve:

→ **Alocar espaço de armazenamento** de valores produzidos pelas diversas ações.

Este mecanismo deve tornar possível que uma ação que utiliza um valor o encontre.

Comunicação entre ações

Uma **gramática de atributo** associa os valores (atributos) aos nós na árvore sintática;

- Unir o armazenamento do atributo ao armazenamento dos nós da árvore torna possível encontrar valores de atributo de um modo sistemático.

Na tradução *ad hoc* dirigida pela sintaxe, o parser pode não construir a árvore sintática.

Ao invés disso, o parser pode integrar o armazenamento para valores ao seu próprio mecanismo de rastreamento do estado da análise — sua pilha interna.

Comunicação em ações

Lembre-se: O esqueleto de *parser* LR(1) armazenou dois valores na pilha para cada símbolo da gramática: **o símbolo e o estado** correspondente.

Podemos substituir esses pares ⟨***símbolo, estado***⟩

por triplas ⟨***valor, símbolo, estado***⟩ .

Isto fornece um atributo de valor único por símbolo da gramática *(exatamente do que o esquema simplificado precisa)*.

Comunicação entre ações

Para gerenciar a pilha, o *parser* empilha e desempilha mais valores.

Em uma redução usando $A \rightarrow \beta$, retira $3 \times |\beta|$ itens da pilha, ao invés de $2 \times |\beta|$, e empilha o valor junto com o símbolo e o estado.

Comunicação entre ações

- *Esta técnica armazena os valores em locais facilmente calculados em relação ao topo da pilha.*

- ***Redução*** → *coloca seu resultado na pilha como parte da tripla que representa o lado esquerdo.*

- ***Ação*** → *lê os valores para o lado direito a partir de suas posições relativas na pilha;*
  - *O **i-ésimo símbolo no lado** direito tem **seu valor na i-ésima tripla** a partir do topo da pilha.*

Comunicação entre ações

INFORMAÇÕES ADICIONAIS:

Os valores são restritos a um tamanho fixo; na prática, esta limitação significa que valores mais complexos são passados usando ponteiros para estruturas.

Para economizar espaço de armazenamento, o parser pode omitir da pilha os símbolos reais da gramática. A informação necessária para a análise sintática é codificada no estado.

Nomeação de valores

Para simplificar o uso de valores baseados em pilha, é necessário uma **notação para nomeá-los.**

O Yacc introduziu uma notação concisa para resolver este problema.

O símbolo $$ refere-se ao local do resultado para a produção atual.

- Assim, a atribuição $$ = *0*; empilha o valor inteiro zero como resultado correspondente à redução atual.

Nomeação de valores

Para o lado direito, os símbolos $*1*, $*2*, ..., $*n* referem-se aos locais para o primeiro, segundo até o *n*-ésimo símbolo no lado direito, respectivamente.

Exemplo reescrito:

	Produção		Trecho de código
1	*Number* → *Sign List*		$$ ← $1 × $2
2	*Sign*	→ +	$$ ← 1
3	*Sign*	→ –	$$ ← −1
4	*List*	→ *Bit*	$$ ← $1
5	$List_0$	→ $List_1$ *Bit*	$$ ← 2 × $1 + $2
6	*Bit*	→ 0	$$ ← 0
7	*Bit*	→ 1	$$ ← 1

Ações em outros pontos da análise

- As vezes é necessário realizar uma ação no meio de uma produção ou em uma ação shift.

- Para isso é necessário transformar a gramática de modo que seja realizada uma redução em cada ponto onde uma ação é necessária.

Ações em outros pontos da análise

Para reduzir no meio de uma produção:

- Pode-se quebrar a produção em duas partes em torno do ponto onde a ação deve ser executada.
  - Uma produção de nível mais alto, que sequencia a primeira parte e depois a segunda, é acrescentada. Quando a primeira parte é reduzida, o parser chama a ação.

Para forçar ações sobre shifts:

O construtor de compiladores pode movê-la para o scanner ou acrescentar uma produção para conter a ação.

Ações em outros pontos da análise

EXEMPLO:

Para realizar uma ação sempre que o *parser* desloca o símbolo *Bit*, o construtor de compiladores pode acrescentar uma produção

$$\boldsymbol{ShifedBit} \rightarrow \boldsymbol{Bit}$$

e substituir cada ocorrência de *Bit* por *ShiftedBit*.

Isto acrescenta uma redução extra para cada símbolo terminal.

Exemplo 1 – Inf. de tipos para exp. (Revisão)

O problema da inferência de tipos para expressões encaixa-se bem ao framework de gramática de atributo (desde que assumamos que os nós folha já contenham informações de tipo).

A simplicidade da solução mostrada anteriormente deriva de dois fatos principais.

- Primeiro: o fluxo natural de informações corre de baixo para cima;
- Segundo: os tipos de expressão são definidos em termos da sintaxe da linguagem fonte.

Isto se encaixa bem ao framework da gramática de atributo, que implicitamente exige a presença de uma árvore sintática.

Exemplo 1 – Inf. de tipos para exp. (Revisão)

Podemos reformular este problema em um framework *ad hoc*.

Produção			Ações dirigidas pela sintaxe
Expr	→	*Expr* + *Termo*	{ $$ ← $\mathcal{F}_{+}$($1,$3) }
	|	*Expr* – *Termo*	{ $$ ← $\mathcal{F}_{-}$($1,$3) }
	|	*Termo*	{ $$ ← $1 }
Term	→	*Termo* × *Fator*	{ $$ ← $\mathcal{F}_{\times}$($1,$3) }
	|	*Termo* ÷ *Fator*	{ $$ ← $\mathcal{F}_{\div}$ ($1,$3) }
	|	*Fator*	{ $$ ← $1 }
Factor	→	(*Expr*)	{ $$ ← $2 }
	|	num	{ $$ ← *tipo do* num }
	|	nome	{$$← *tipo do* nome }

Nesse caso, o framework *ad hoc* não oferece uma vantagem real para este problema.

Exemplo 2: *Criação de uma árvore sintática abstrata*

- Os front ends de compiladores precisam criar uma representação intermediária do programa para usar na parte do meio do compilador e no seu back end.

- As árvores sintáticas abstratas são uma forma comum de IR estruturada em árvore.

- A tarefa de criar uma AST assenta-se bem a um esquema de tradução *ad hoc* dirigida pela sintaxe.

Exemplo 2: *Criação de uma árvore sintática abstrata*

Suponha que o compilador tenha uma série de rotinas chamadas *MakeNodei*, para $0 \leq i \leq 3$.

A rotina usa, como primeiro argumento, uma constante que identifica exclusivamente o símbolo da gramática que o novo nó representará.

Os *i* argumentos restantes são os nós que encabeçam cada uma das *i* subárvores.

Exemplo 2: *Criação de uma árvore sintática abstrata*

Assim, *MakeNode*0 (*number*) constrói um nó folha e o marca como representando um num.

De modo semelhante,

$$MakeNode_2\ (Plus,\ MakeNode_0\ (number,)\ makeNode_0\ (number))$$

cria uma AST cuja raiz é um nó para *plus* com dois filhos, cada um deles um nó folha para num.

Exemplo 2: *Criação de uma árvore sintática abstrata*

Para criar uma árvore sintática abstrata, o esquema de tradução *ad hoc* dirigida pela sintaxe segue dois princípios gerais:

1. Para um operador, cria um nó com um filho para cada operando. Assim, 2 + 3 cria um nó binário para + com os nós para 2 e 3 como filhos.

2. Para uma produção "inútil", como *Termo* → *Fator*, reutiliza o resultado da ação *Fator* como seu próprio resultado.

Produção			Ações dirigidas pela sintaxe
Expr	→	*Expr + Termo*	{ $$ ← $MakeNode_2$(add, $1, $3);
			$$.type ← $\mathcal{F}_+$ ($1.type, $3.type) }
	|	*Expr – Termo*	{ $$ ← $MakeNode_2$ (sub, $1, $3);
			$$.type ← $\mathcal{F}_-$ ($1.type,$3.type) }
	|	*Termo*	{ $$ ← $1 }
Termo	→	*Termo × Fator*	{ $$ ← $MakeNode_2$ (mult, $1, $3);
			$$.type ← $\mathcal{F}_\times$ ($1.type, $3.type) }
	|	*Termo ÷ Fator*	{ $$ ← $MakeNode_2$ (div, $1, $3);
			$$.type ← $\mathcal{F}_\div$ ($1.type, $3.type) }
	|	*Fator*	{ $$ ← $1 }
Fator	→	*(Expr)*	{ $$ ← $2 }
	|	num	{ $$ ← $MakeNode_0$ (number);
			$$.text ← *texto retornado pelo scanner;*
			$$.type ← *tipo do número* }
	|	nome	{ $$ ← $MakeNode_0$ (identifier);
			$$.text ← *texto retornado pelo scanner;*
			$$.type ← *tipo do identificador* }

Este esquema evita a criação de nós de árvore que representam variáveis sintáticas, como *Fator*, *Termo* e *Expr*

Exemplo 3 - *Geração de ILOC para expressões*

Considere um framework *ad hoc* que gera ILOC ao invés de uma AST.

Faremos várias suposições para simplificar. :

1. O exemplo limita sua atenção a inteiros;

→ O tratamento de outros tipos aumenta a complexidade, mas não muito a percepção;

2. Considera que todos os valores podem ser mantidos em registradores;

→ tanto, que os valores cabem nos registradores, como que a implementação ILOC fornece mais registradores do que a computação usará.

Exemplo 3 - *Geração de ILOC para expressões*

Para remover o máximo de detalhes, o framework de exemplo utiliza quatro rotinas de suporte.

1. ***Address*** utiliza um nome de variável como seu argumento. Retorna o número de um registrador que contém o valor especificado por nome.

2. *Emit* trata dos detalhes da criação de uma representação concreta para as diversas operações ILOC, e pode formatá-las e imprimi-las em um arquivo.

3. *NextRegister* retorna um novo número de registrador.

4. *Value* utiliza um número como seu argumento e retorna um número de registrador.

Produção			Ações dirigidas pela sintaxe
Expr	→	*Expr* + *Termo*	{ $$ ← NextRegister; Emit(add, $1, $3, $$) }
	|	*Expr* – *Termo*	{ $$ ← NextRegister; Emit(sub, $1, $3, $$) }
	|	*Termo*	{ $$ ← $1 }
Termo	→	*Termo* × *Fator*	{ $$ ← NextRegister; Emit(mult, $1, $3, $$) }
	|	*Termo* ÷ *Fator*	{ $$ ← NextRegister; Emit(div, $1, $3,$$) }
	|	*Fator*	{ $$ ← $1 }
Fator	→	(*Expr*)	{ $$ ← $2 }
	|	num	{ $$ ← Value(texto retornado pelo scanner); }
	|	nome	{ $$ ← Address(texto retornado pelo scanner); }

As ações se comunicam passando os nomes de registradores na pilha de análise.

As ações passam esses nomes para *Emit* conforme a necessidade, a fim de criar as operações que implementam a expressão de entrada.

E AS GRAMÁTICAS SENSÍVEIS AO CONTEXTO?

Pode ser natural considerar o uso de linguagens sensíveis ao contexto para realizar verificações sensíveis ao contexto, como a inferência de tipos.

Afinal, usamos:

Linguagens regulares para realizar análise léxica;

Linguagens livres de contexto para a análise sintática.

Uma progressão natural poderia sugerir o estudo de linguagens sensíveis ao contexto e suas gramáticas.

As gramáticas sensíveis ao contexto podem expressar uma família maior de linguagens do que as gramáticas livres de contexto.

E AS GRAMÁTICAS SENSÍVEIS AO CONTEXTO?

Gramáticas sensíveis ao contexto não são a resposta certa por dois motivos distintos.

Primeiro: o problema de analisar uma gramática sensível ao contexto é P-Espeço completo. Assim, um compilador que usasse tal técnica poderia executar *muito* lentamente.

Segundo: muitas das questões importantes são difíceis, se não impossíveis de codificar em uma gramática sensível ao contexto.