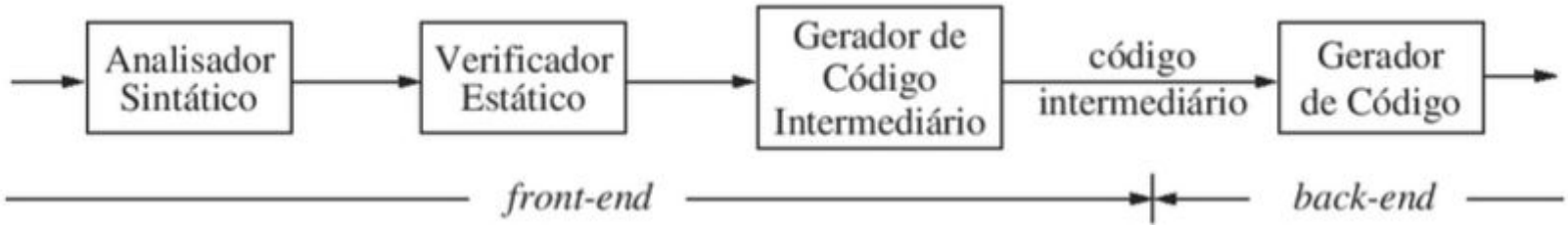
Geração de código intermediário

Equipe:
Daniel Lucas Santana Santos
Valber Francisco dos Santos

Introdução

No modelo de análise e síntese de um compilador, o front-end analisa um programa fonte e cria uma representação intermediária para o back-end gerar o código objeto.



1) Variante das árvores de sintaxe

As árvores de sintaxe representam em seus nós, construções do programa fonte e filhos dos nós representam componentes da construção.

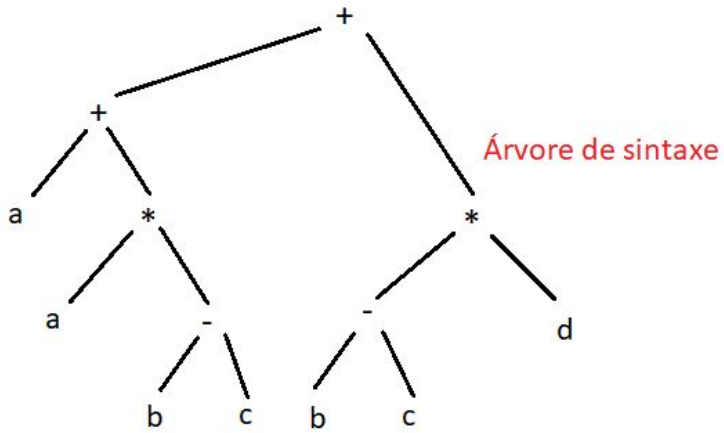
Um grafo acíclico dirigido ou DAG (Directed Acyclic Graph), pode ser construído de forma semelhante a uma árvore de sintaxe, mas sempre analisa a existência de uma subexpressão comum.

1.1) DAG - Grafos acíclicos dirigidos

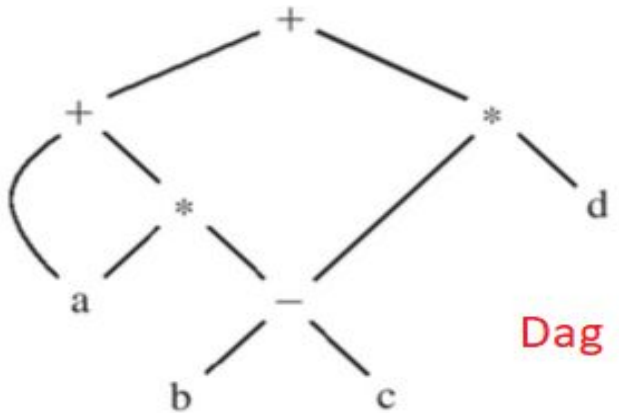
Um DAG assim como uma árvore de sintaxe, possui folhas que são os operandos e os nós interiores são os operadores. O que difere um DAG de uma árvore de sintaxe é que um nó pode ter mais de um pai se o nó representar uma subexpressão que já existe.

Exemplo:

Para a expressão a + a * (b - c) + (b - c) * d temos,



Árvore de sintaxe



Dag

1.1) DAG - Grafos acíclicos dirigidos

Para construir um DAG, pode-se usar também a definição dirigida por sintaxe que é utilizada para produzir as árvores de sintaxe:

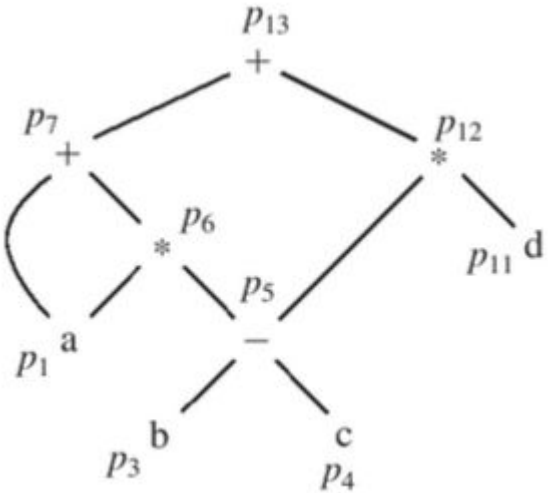
PRODUÇÃO	REGRAS SEMÂNTICAS
1) $E \rightarrow E_1 + T$	$E.node =$ **new** $Node(\text{'+'}, E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node =$ **new** $Node(\text{'}-\text{'}, E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow$ **id**	$T.node =$ **new** $Leaf$ (**id**, **id**.$entry$)
6) $T \rightarrow$ **num**	$T.node =$ **new** $Leaf$ (**num**, **num**.val)

1.1) DAG - Grafos acíclicos dirigidos

Sabendo disso e voltando ao exemplo anterior, agora é possível analisar o DAG de forma mais detalhada:

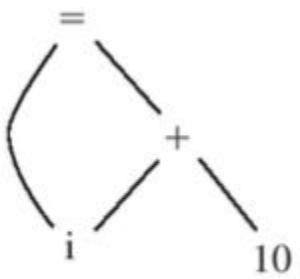
Dado a + a * (b - c) + (b - c) * d

$p_1 = Leaf(\textbf{id}, entry\text{-}a)$
$p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
$p_3 = Leaf(\textbf{id}, entry\text{-}b)$
$p_4 = Leaf(\textbf{id}, entry\text{-}c)$
$p_5 = Node(\text{'-'}, p_3, p_4)$
$p_6 = Node(\text{'*'}, p_1, p_5)$
$p_7 = Node(\text{'+'}, p_1, p_6)$
$p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
$p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
$p_{10} = Node(\text{'-'}, p_3, p_4) = p_5$
$p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
$p_{12} = Node(\text{'*'}, p_5, p_{11})$
$p_{13} = Node(\text{'+'}, p_7, p_{12})$

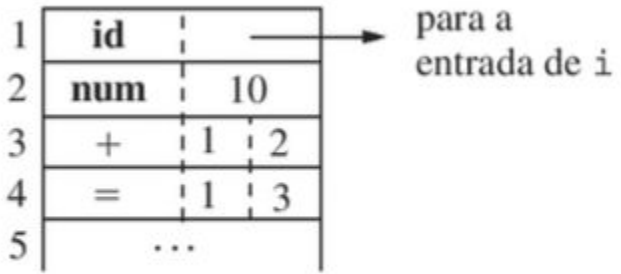


1.2) Método de código numérico para construção de DAG

Os nós de uma árvore de sintaxe ou de um DAG são armazenados em um arranjo de registros em que cada linha representa os dados do respectivo nó.



(a) DAG



(b) Arranjo

1.2) Método de código numérico para construção de DAG

Para a construir um nó, pode-se utilizar o seguinte método:

Entrada: uma tupla <op,L,R>

Saída: código numérico do nó no arranjo com a tupla <op,L,R>

Método: fazer uma busca no arranjo um nó M que contenha a tupla <op,L,R>. Se o nó M existir, retorne o código numérico associado a ele. Caso não exista, crie um nó N com a tupla <op,L,R> e retorne seu código numérico.

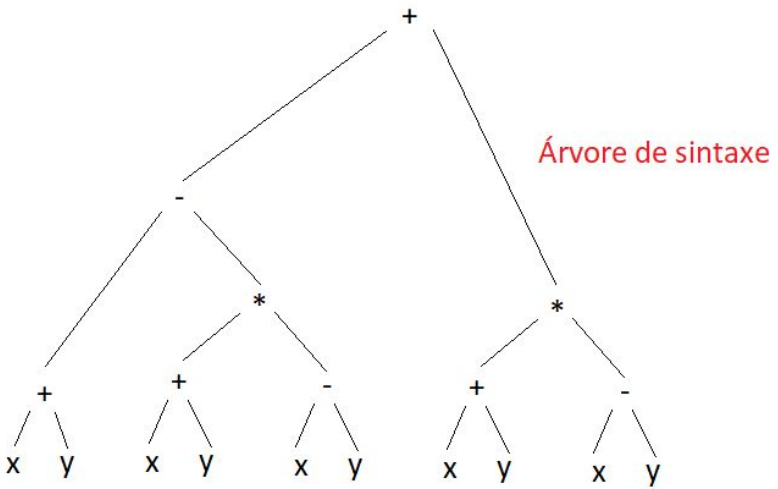
Obs.: A busca pelo nó no DAG inteiro é bem custoso, uma das melhores soluções para esse problema é armazenar os nós numa tabela hash, dada uma função hash que utilize a tupla <op, L, R>.

Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))

Exercício:

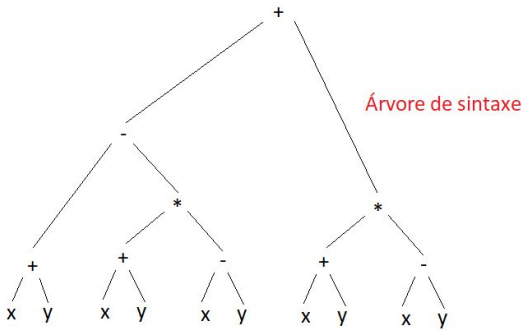
Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



Árvore de sintaxe

Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



Árvore de sintaxe

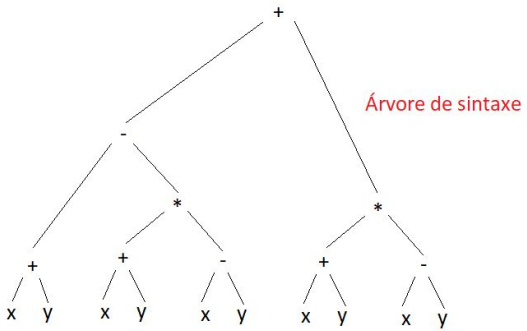
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)

((x+y) - ((x+y) * (x-y))) + ((x+y) * (x-y))

x y

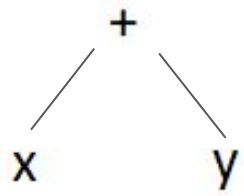
Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



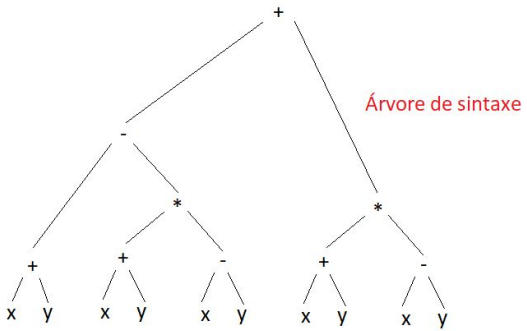
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)

((x+y) - (p3 * (x-y))) + ((x+y) * (x-y))



Exercício:

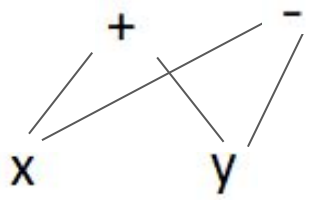
Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



Árvore de sintaxe

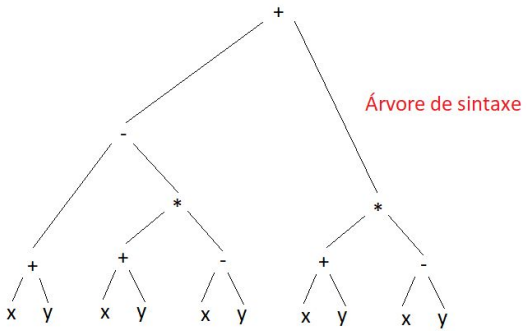
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)

((x+y) - (p3 * p4)) + ((x+y) * (x-y))



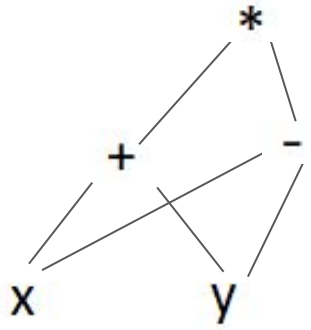
Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



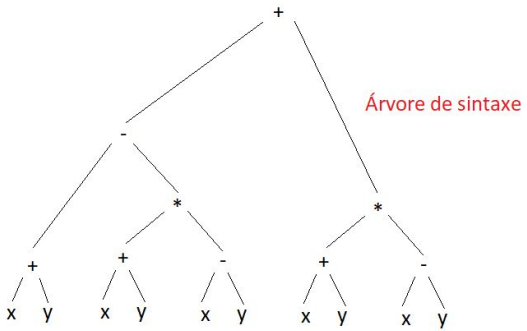
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)

((x+y) - p5) + ((x+y) * (x-y))

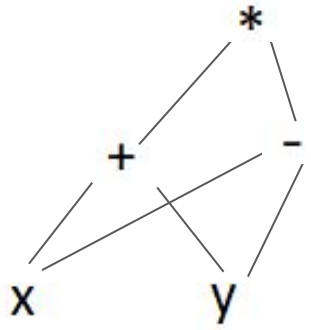


Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



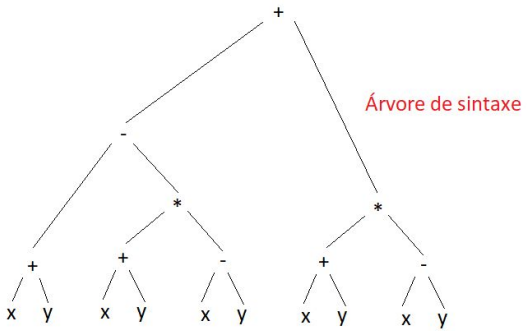
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)
p6 = node('+', p1, p2) = p3



(p3 - p5) + ((x+y) * (x-y))

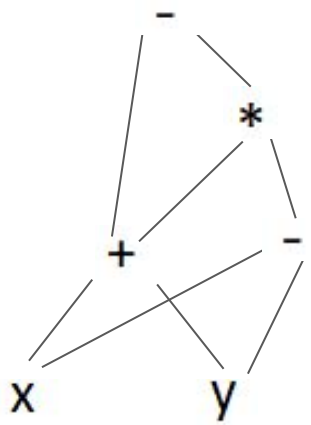
Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



Árvore de sintaxe

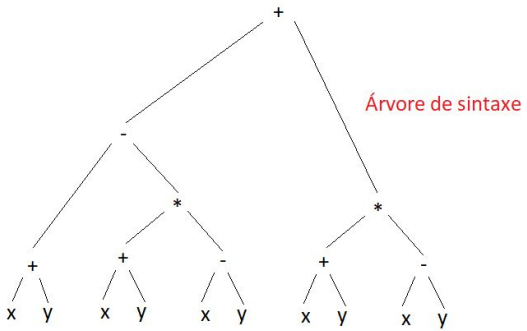
p7 + ((x+y) * (x-y))



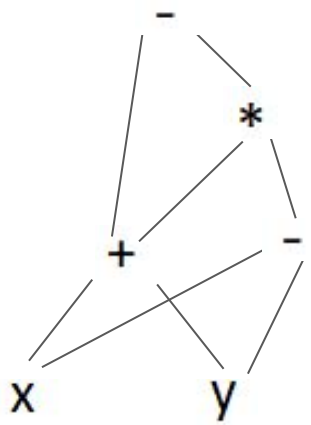
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)
p6 = node('+', p1, p2) = p3
p7 = node('-', p3, p5)

Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



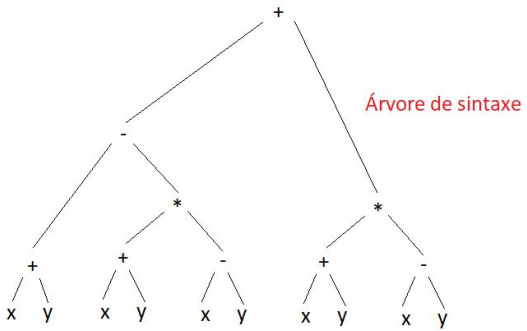
p7 + (p3 * (x-y))



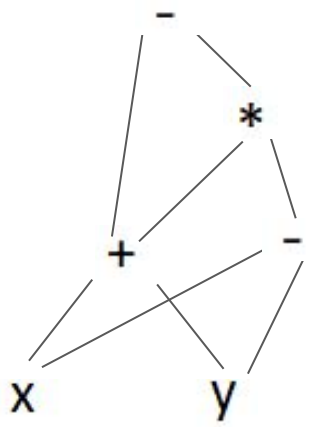
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)
p6 = node('+', p1, p2) = p3
p7 = node('-', p3, p5)
p8 = node('+', p1, p2) = p3

Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



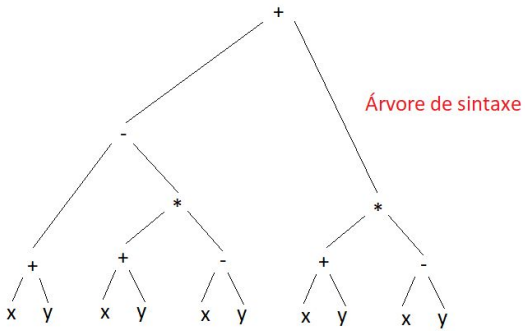
p7 + (p3 * p4)



p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)
p6 = node('+', p1, p2) = p3
p7 = node('-', p3, p5)
p8 = node('+', p1, p2) = p3
p9 = node('-', p1, p2) = p4

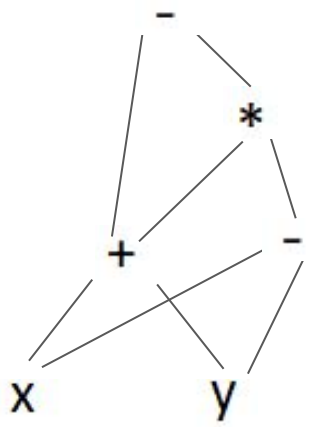
Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



Árvore de sintaxe

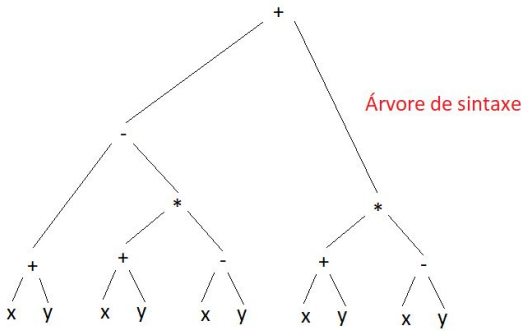
p7 + p5



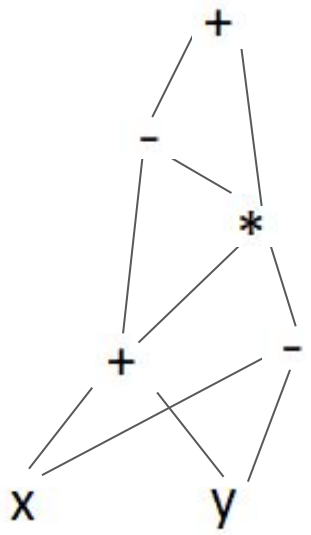
p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)
p6 = node('+', p1, p2) = p3
p7 = node('-', p3, p5)
p8 = node('+', p1, p2) = p3
p9 = node('-', p1, p2) = p4
p10 = node('*', p3, p4) = p5

Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))



p11



p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)
p6 = node('+', p1, p2) = p3
p7 = node('-', p3, p5)
p8 = node('+', p1, p2) = p3
p9 = node('-', p1, p2) = p4
p10 = node('*', p3, p4) = p5
p11 = node('+', p7, p5)

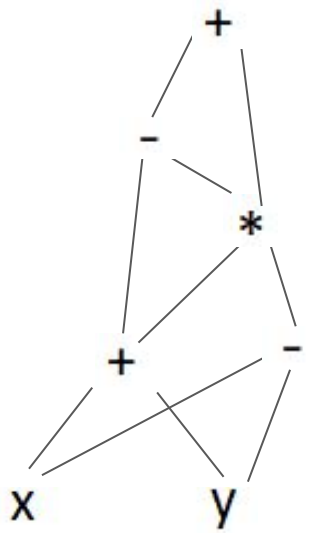
Exercício:

Construa uma DAG para a expressão ((x+y) - ((x+y) * (x-y))) + ((x + y) * (x - y))

Arranjo:

1	id	x	
2	id	y	
3	+	1	2
4	-	1	2
5	*	3	4
6	-	3	5
7	+	6	5

p11



p1 = leaf(id, entry-x)
p2 = leaf(id, entry-y)
p3 = node('+', p1, p2)
p4 = node('-', p1, p2)
p5 = node('*', p3, p4)
p6 = node('+', p1, p2) = p3
p7 = node('-', p3, p5)
p8 = node('+', p1, p2) = p3
p9 = node('-', p1, p2) = p4
p10 = node('*', p3, p4) = p5
p11 = node('+', p7, p5)

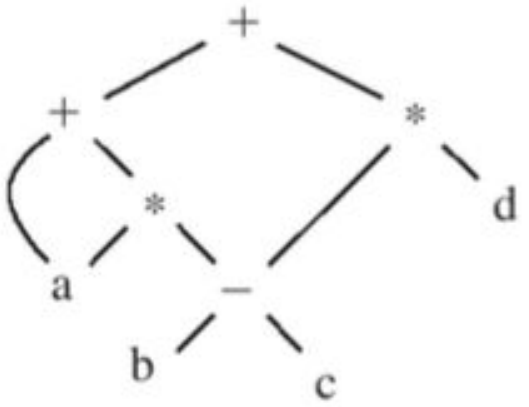
2) Código de três endereços

Nenhuma expressão aritmética com vários operandos é permitida. Assim, uma expressão descrita como **x+y*z** deve ser traduzida para a sequência de instruções de três endereços:

- $t^1 = y * z$
- $t^2 = x + t^1$

onde t^1 e t^2 são nomes **temporários** gerados pelo compilador. Onde tal simplificação torna o código de três endereços desejável para a **otimização** e permite facilmente que seja rearranjado.

Exemplo:



(a) DAG

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

(b) Código de três endereços

2.1) Endereços e Instruções

- **Endereço pode ser**:

  > **Um nome**: nomes do programa fonte aparecem como endereços, sendo substituído por um ponteiro para sua entrada na tabela de símbolos na qual contém suas informações.
  > **Uma constante**: um compilador trata muitos tipos de variáveis e constantes.
  > **Um temporário**: interessante para otimização, podendo ser combinados, se possivel, quando registradores são alocados a variaveis.

2.1) Endereços e Instruções

Lista das formas mais comuns de instruções de três endereços:

1. **x = y op z**, onde **op** é um **operador aritmético** binário ou lógico, e **x**, **y** e **z** são **endereços**.
2. **x = y op z**, onde **op** é um **operador** unário. Tal operador inclui menos unário, negação lógica, operadores de deslocamento e conversão.
3. Cópia da forma **x = y**, onde **x** recebe o valor de **y**.
4. Desvio incondicional **goto** L.
5. Desvios condicionais da forma **if x** goto **L** e **ifFalse x goto L**.
6. Desvios condicionais tais como **if x** relop **y** goto **L**, que aplicam um operador condicional (<, ==, >= etc.) a **x** com **y** e executam **L** se **x relop y.**

2.1) Endereços e Instruções

7. **param =** parâmetros**, call =** chamadas de procedimento e função**, return y =** onde y é o valor a ser retornado (opcional).

```
param x1
param x2
  ...
param xn
call p,n
```

8. Instruções indexadas de cópia da forma x = y[i] (atribui a x o valor contido no endereço i unidades de memória além do endereço de y) e x[i] = y (atribui ao conteúdo do endereço i unidades além do x o valor de y).

9. Atribuições de endereço e ponteiro da forma x = &y (define valor-r de x para ser o endereço [valor-l] de y), x= *y e *x = y.

Exemplo:

Duas formas de atribuir rótulos às instruções de três endereços.

```
do i = i+1; while (a[i] < v);
```

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

(a) Rótulos simbólicos

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Posição numérica

2.2) Quádruplas

Uma quádrupla, representação de instruções em uma estrutura de dados, (ou quad) possui quatro campos chamados de **op, arg¹, arg² e result**.

> **op** contém um código interno para o operador. Por exemplo, a instrução de três endereços **x = y + z** é representada pela colocação de **+** em **op**, **y = arg¹**, **z = arg²**, e **x = result**. Algumas **exceções** a essas regras:

1. Instruções com operadores unários como x = `minus` *y* ou *x* = *y* não usam arg_2. Observe que, para um comando de cópia como *x* = *y*, *op* é =, enquanto para a maioria das outras operações, o operador de atribuição é implícito.
2. Operadores como `param` não utilizam arg_2 nem *result*.
3. Comandos condicionais e incondicionais colocam o rótulo de destino em *result*.

Exemplo:

Código de três endereços em representação quádrupla.

```
a = b * - c + b * - c ;
```

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Código de três endereços

	op	arg_1	arg_2	*result.*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quádruplas

obs: minus é utilizado para distinguir o operador unário menos, como em - c e b - c

2.3) Triplas

Tem apenas três campos: op, arg^1, arg^2.

Usando triplas, uma operação x op y é referida por sua **posição**, em vez de um nome **temporário**. Assim, utilizando o exemplo anterior, ao invés de $\mathbf{t^1}$ representaremos com a posição **(0)**. Os números entre parênteses representam **ponteiros** para a própria estrutura da tripla, representados como códigos numéricos em seções anteriores.

Exemplo:

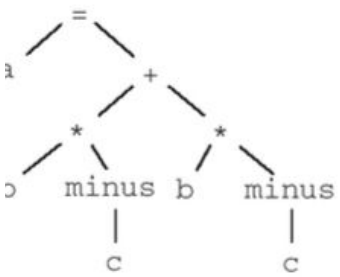
```
a = b * - c + b * - c ;
```

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Código de três endereços

	op	arg_1	arg_2	result.
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quádruplas



(a) Árvore de sintaxe

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triplas

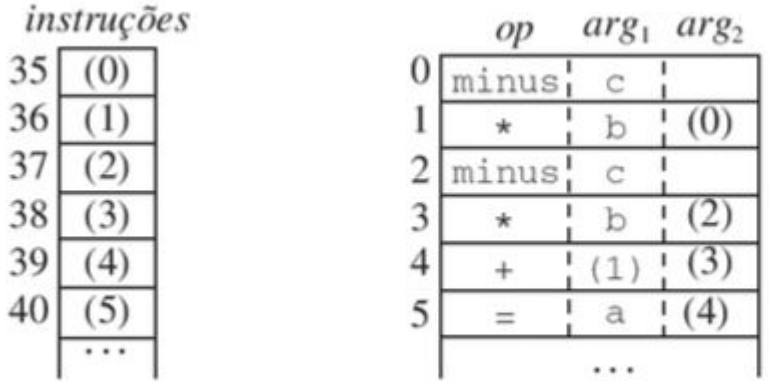
Triplas x Quádruplas

Uma vantagem das quádruplas em relação às triplas pode ser vista em um compilador otimizado, em que as instruções normalmente mudam de lugar.

As triplas são sempre referenciadas pela posição, assim movimentar uma instrução de lugar pode complicar o andamento do processo. Já com as quádruplas tal operação é mais simples pelo fato do uso dos temporários.

Triplas Indiretas

Tenta resolver o problema destacado anteriormente implementando uma lista de ponteiros para as triplas, em vez de uma lista das próprias triplas. Por exemplo:



Um compilador otimizado pode movimentar uma instrução reordenando a lista instruction sem afetar as próprias triplas.

2.4) Forma de atribuição única estática (SSA)

SSA (do ingles Static Single-Assignment), é uma representação intermediária que facilita certas otimizações.

Diferenças entre SSA e código de três endereços:

1. Todas as atribuições em SSA são para variáveis de nomes distintos (atribuição única).

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Código de três endereços.

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

(b) Forma de atribuição única estática.

Exemplo Prático:

- Convencional

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

- SSA

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1, x2 );
```

SSA usa uma convenção nacional, chamada de função Φ para combinar as definições de x.

3) Tipos e Declarações

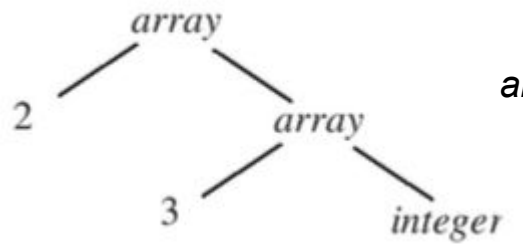
- Verificação de tipo: garante que os operandos são do tipo esperado pelo operador.

- Aplicações de tradução: através do tipo, determina a forma de armazenamento que será necessário no momento da execução.

3.1) Expressões de tipo

Expressões de tipo são os tipos básicos(inteiros, floats, char, boolean, void), assim como também são os tipos criados a partir de construtores de tipo, como por exemplo, o *struct* da *linguagem C.*

Para um dado arranjo int[2][3], podemos expressá-lo da seguinte forma



array(2, array(3, int))

3.1) Expressões de tipo

Definições para expressões de tipo:

1. Os tipos básicos são expressões de tipo.
2. Um nome de tipo é uma expressão de tipo.
3. Uma expressão de tipo pode ser formada por um construtor de tipo array a um número e uma expressão de tipo.
4. Uma expressão de tipo pode ser formada por um construtor do tipo record.
5. Uma expressão de tipo pode ser formada por um construtor do tipo -> para tipos de função, dada uma função s -> t.
6. Se s e t forem um expressão de tipo, o produto cartesiano s x t também é uma expressão de tipo.
7. Expressões de tipo podem conter variáveis cujo os valores são expressões de tipo

3.2) Equivalência de tipo

Quando representadas por grafos, duas expressões de tipo tem tipos equivalentes se e somente se uma dessas condições forem cumpridas:

- Os tipos são do mesmo tipo básico.
- Os tipos são formados pelo mesmo construtor.
- O nome de um tipo denota o outro.

3.3) Declarações

D: sequência de declarações

T: gera tipos básicos de B, de arranjo ou registro.

B: gera tipos básicos int e float.

C: gera uma cadeia de zero ou mais inteiros.

$$D \rightarrow T\ \mathbf{id}\ ;\ D \mid \epsilon$$

$$T \rightarrow B\ C \mid \mathbf{record}\ '\{'\ D\ '\}'$$

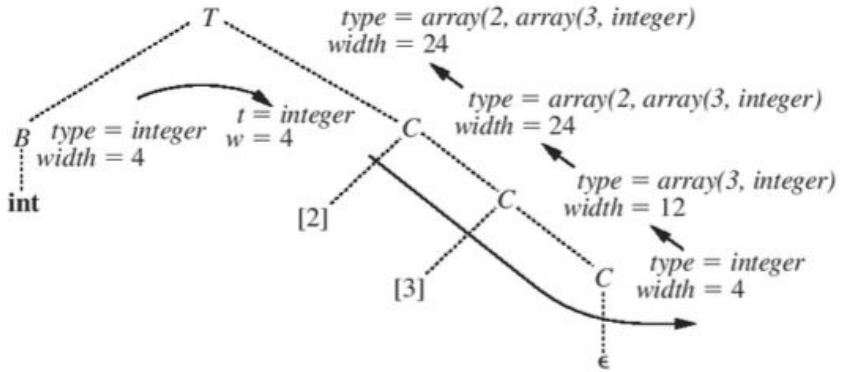
$$B \rightarrow \mathbf{int} \mid \mathbf{float}$$

$$C \rightarrow \epsilon \mid [\ \mathbf{num}\ ]\ C$$

3.4) Layout de armazenamento

Esquema de tradução:

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
$\quad C$	
$B \rightarrow \textbf{int}$	$\{ B.type = integer; B.width = 4; \}$
$B \rightarrow \textbf{float}$	$\{ B.type = float; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$\{ array(\textbf{num}.value, C_1.type);$
	$\quad C.width = \textbf{num}.value \times C_1.width; \}$



3.5) Sequência de declarações

Para efetuar uma sequência de declarações, uma das alternativas é gravar os deslocamentos das declarações anteriores em uma variável offset e somar com o tamanho do tipo da próxima declaração

$P \rightarrow$ { *offset* = 0; }

D

$D \rightarrow T$ **id** ; { *top.put*(**id**.*lexeme*, *T.type*, *offset*);

offset = *offset* + *T.width*; }

D_1

$D \rightarrow \epsilon$

3.6) Campos em registros e classes

Para determinar tipos e endereços para os campos em um registro ou classe, é necessário que se evite que dois casos ocorram:

- Os nomes dos campos de um registro devem ser distintos.

- O deslocamento ou endereço relativo para um nome de campo é relativo a área de dados para esse registro.

3.6) Campos em registros e classes

Exemplo:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

Não existe conflito!

x =/= p.x =/= q.x, logo uma atribuição do tipo x = p.x + q.x é possível.

3.6) Campos em registros e classes

Gramática:

$$
\begin{array}{lll}
T \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env();\\
 & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & D\ '\}' & \{\ T.type = record(top);\ T.\text{width} = offset; \\
 & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

4) Tradução de expressões

- 4.1 Operações no interior de expressões
  > code = codigo de três endereços.
  > addr = endereço que conterá o valor.

```
a = b + - c;
t1 = minus c
t2 = b + t1
a = t2
```

comando de atribuição

código de três endereços

PRODUÇÃO	REGRAS SEMÂNTICAS
$1S \rightarrow \mathbf{id} = E\,;$	$S.code = E.code \,\|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \,\|\| E_2.code \,\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ -\ E_1$	$E.addr = \mathbf{new}\ Temp\ ()$ $E.code = E_1.code \,\|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

- **gen**: monta uma instrução e a retorna. Por conveniência é utilizada para representar instrução de três endereços também;
- || representa concatenação**;**
- **new temp**(): cria nomes temporários;
- **E-> id**, quando a expressão é um único identificador, então ele contém o valor da expressão;
- **top.get()** recupera a entrada quando ela é aplicada à representação da cadeia id.lexeme dessa instância de id (atribui endereço).

4.2) Tradução Incremental

Ao invés de construir E.code, pode-se gerar apenas as novas instruções de três endereços. Nesta abordagem **gen** não só constrói uma instrução de três endereços, mas anexa essa instrução ao conjunto de instruções geradas até o momento.

$S \rightarrow \mathbf{id} = E\ ;$ { *gen* (*top.get*(**id**.*lexeme*) ' = ' *E.addr*); }

$E \rightarrow E_1 + E_2$ { *E.addr* = **new** *Temp* ();
gen(*E.addr* ' = ' E_1.*addr* ' + ' E_2.*addr*); }

| $-\mathrm{E}_1$ { *E.addr* = **new** *Temp* ();
gen(E.*addr* ' = ' ' **minus** ' E_1.*addr*); }

| (E_1) { *E.addr* = E_1.*addr* ; }

| **id** { *E.addr* = *top.get*(**id**.*lexeme*); }

obs: o atributo **code** não é utilzado, uma vez que existe uma única sequencia de instruçoes que é criada por chamadas sucessivas a gen.

4.2) Tradução Incremental x Op. Interior

$S \rightarrow$ **id** $= E$;	{ *gen* (*top.get*(**id**.*lexeme*) ' = ' *E.addr*); }
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp* (); *gen*(*E.addr* ' = ' E_1.*addr* ' + ' E_2.*addr*)
| $- E_1$	{ *E.addr* = **new** *Temp* (); *gen*(E.*addr* ' = ' ' **minus** ' E_1.*addr*); }
| (E_1)	{ *E.addr* = E_1.*addr* ; }
| **id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }

PRODUÇÃO	REGRAS SEMÂNTICAS
$1S \rightarrow$ **id** $= E$;	*S.code* = *E.code* ‖ *gen*(*top.get*(**id**.*lexeme*) ' = ' *E.addr*)
$E \rightarrow E_1 + E_2$	*E.addr* = **new** *Temp*() *E.code* = E_1.*code* ‖ E_2.*code* ‖ *gen*(*E.addr* ' = ' E_1.*addr* ' + ' E_2.*addr*)
| $- E_1$	*E.addr* = **new** *Temp* () *E.code* = E_1.*code* ‖ *gen*(E.*addr* ' = ' ' **minus** ' E_1.*addr*)
| (E_1)	*E.addr* = E_1.*addr* *E.*code = E_1.*code*
| **id**	*E.addr* = *top.get*(**id**.*lexeme*) *E.code* = ' '

4.3) Endereçando elementos de vetor

Elementos de um arranjo podem ser acessados rapidamente se forem armazenados em blocos de endereço consecutivos.

- Cálculo do i-ésimo $base + i \times w$ do arranjo:
  base = endereço relativo A[0] e w é a largura de cada endereço.

- Para duas $base + i_1 \times w_1 + i_2 \times w_2 + ... + i_k \times w_k$ nensões:
  w^1= largura de uma linha e w^2= largura de um elemento em uma linha.

4.4) Tradução de referências a vetor

- Vamos calcular os endereços partindo da fórmula abaixo, com base nas larguras. O exemplo ao lado gera o código de três endereços para expressões com referências a vetores.

$$base + i_1 \times w_1 + i_2 \times w_2 + ... + i_k \times w_k$$

$S \rightarrow$ **id** $= E$;	{ *gen*(*top.get*(**id**.*lexeme*) ' = ' *E.addr*); }
| $L = E$;	{ *gen*(*L.addr.base* '[' *L.addr* ']' ' = ' *E.addr*); }
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' E_1*.addr* ' + ' E_2*.addr*); }
| **id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
| *L*	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' *L.array.base* '[' *L.addr* ']'); }
L $\rightarrow$ **id** [*E*]	{ *L.array* = *top.get*(**id**.*lexeme*); L.type = L.array.type.*elem*; *L.addr* = **new** *Temp*(); *gen*(*L.addr* ' = ' *E.addr* '*' *L.type.width*); }
| L_1 [*E*]	{ *L.array* = L_1*.array*; *L.type* = L_1*.type.elem*; *t* = **new** *Temp*(); *L.addr* = **new** *Temp*(); *gen*(*t* ' = ' *E.addr* '*' *L.type.width*); } *gen*(*L.addr* ' = ' L_1*. addr* ' + ' *t*); }

4.4) Tradução de referências a vetor

1. S -> id = E; Representa uma atribuição para uma variável que não é array.
2. S -> L = E; Gera uma instrução de cópia indexada para atribuir o valor denotado pela expressão E à localização denotada pela referência ao array L.
3. E -> E¹ + E² e E -> id são as mesmas de antes.
4. E -> L copia o valor local para um novo temporário.
5. L.array fornece o nome.
6. L.array.base fornece o endereço base.
7. L.addr contem o deslocamento.

$$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$$

$S \rightarrow \mathbf{id} = E\ ;$	{ *gen*(*top.get*(**id**.*lexeme*) ' = ' *E.addr*); }
$\mid L = E\ ;$	{ *gen*(*L.addr.base* '[' *L.addr* ']' ' = ' *E.addr*); }
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' E_1*.addr* ' + ' E_2*.addr*); }
$\mid$ **id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
$\mid L$	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' *L.array.base* '[' *L.addr* ']'); }
L → **id** [*E*]	{ *L.array* = *top.get*(**id**.*lexeme*); L.type = L.array.type.*elem*; *L.addr* = **new** *Temp*(); *gen*(*L.addr* ' = ' *E.addr* '*' *L.type.width*); }
$\mid L_1$ [*E*]	{ *L.array* = L_1*.array*; *L.type* = L_1*.type.elem*; *t* = **new** *Temp*(); *L.addr* = **new** *Temp*(); *gen*(*t* ' = ' *E.addr* '*' *L.type.width*); } *gen*(*L.addr* ' = ' L_1*. addr* ' + ' *t*); }

O não-terminal L possui três atributos sintetizados:

1. *L.addr* denota um temporário que é usado enquanto calcula o deslocamento da referência ao arranjo, somando os termos $i_j \times w_j$.
2. *L.array* é um apontador para a entrada da tabela de símbolos para o nome do arranjo. O endereço de base do arranjo, digamos, *L.array.base*, é usado para determinar o valor-*l* corrente de uma referência ao arranjo depois que todas as expressões de índice forem analisadas.
3. *L.type* é o tipo do subarranjo gerado por L. Para qualquer tipo t, assumimos que sua largura é dada por *t.width*. Usamos os tipos como atributos, em vez de larguras, já que os tipos são necessários para a verificação de tipo. Para qualquer tipo arranjo t, suponha que *t.elem* dê o tipo do elemento.

$$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$$

$S \rightarrow$ **id** $= E$;	{ *gen*(*top.get*(**id**.*lexeme*) ' = ' *E.addr*); }
| $L = E$;	{ *gen*(*L.addr.base* '[' *L.addr* ']' ' = ' *E.addr*); }
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' E_1*.addr* ' + ' E_2*.addr*); }
| **id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
| L	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' *L.array.base* '[' *L.addr* ']'); }
L $\rightarrow$ **id** [E]	{ *L.array* = *top.get*(**id**.*lexeme*); *L*.type = *L*.array.type*.elem*; *L.addr* = **new** *Temp*(); *gen*(*L.addr* ' = ' *E.addr* '*' *L.type.width*); }
| L_1 [E]	{ *L.array* = L_1*.array*; *L.type* = L_1*.type.elem*; *t* = **new** *Temp*(); *L.addr* = **new** *Temp*(); *gen*(*t* ' = ' *E.addr* '*' *L.type.width*); } *gen*(*L.addr* ' = ' L_1*. addr* ' + ' *t*); }

4.4) Tradução de referências a vetor

O não-terminal L possui três atributos sintetizados:

1. *L.addr* denota um temporário que é usado enquanto calcula o deslocamento da referência ao arranjo, somando os termos $i_j \times w_j$ em (6.4).
2. *L.array* é um apontador para a entrada da tabela de símbolos para o nome do arranjo. O endereço de base do arranjo, digamos, *L.array.base*, é usado para determinar o valor-*l* corrente de uma referência ao arranjo depois que todas as expressões de índice forem analisadas.
3. *L.type* é o tipo do subarranjo gerado por *L*. Para qualquer tipo *t*, assumimos que sua largura é dada por *t.width*. Usamos os tipos como atributos, em vez de larguras, já que os tipos são necessários para a verificação de tipo. Para qualquer tipo arranjo *t*, suponha que *t.elem* dê o tipo do elemento.

$S \rightarrow$ **id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) ' = ' *E.addr*); }
| L = E ;	{ *gen*(*L.addr.base* '[' *L.addr* ']' ' = ' *E.addr*); }
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' E_1*.addr* ' + ' E_2*.addr*); }
| **id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
| L	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' *L.array.base* '[' *L.addr* ']'); }
L → **id** [E]	{ *L.array* = *top.get*(**id**.*lexeme*); *L*.type = *L*.array.type.*elem*; *L.addr* = **new** *Temp*(); *gen*(*L.addr* ' = ' *E.addr* '*' *L.type.width*); }
| L_1 [E]	{ *L.array* = L_1*.array*; *L.type* = L_1*.type.elem*; *t* = **new** *Temp*(); *L.addr* = **new** *Temp*(); *gen*(*t* ' = ' *E.addr* '*' *L.type.width*); } *gen*(*L.addr* ' = ' L_1. *addr* ' + ' *t*); }

4.4) Tradução de referências a vetor

- Vamos calcular os endereços com base na fórmula abaixo, com base nas larguras. O exemplo abaixo gera o código de três endereços para expressões com referências a vetores.

$$base + i_1 \times w_1 + i_2 \times w_2 + ... + i_k \times w_k$$

$S \rightarrow$ **id** $= E$;	{ *gen*(*top.get*(**id**.*lexeme*) ' = ' *E.addr*); }
| $L = E$;	{ *gen*(*L.addr.base* '[' *L.addr* ']' ' = ' *E.addr*); }
$E \rightarrow E_1 + E_2$	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' E_1*.addr* ' + ' E_2*.addr*); }
| **id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
| L	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* ' = ' *L.array.base* '[' *L.addr* ']'); }
L $\rightarrow$ **id** [E]	{ *L.array* = *top.get*(**id**.*lexeme*); L.type = L.array.type.*elem*; *L.addr* = **new** *Temp*(); *gen*(*L.addr* ' = ' *E.addr* '*' *L.type.width*); }
| L_1 [E]	{ *L.array* = L_1*.array*; *L.type* = L_1*.type.elem*; *t* = **new** *Temp*(); *L.addr* = **new** *Temp*(); *gen*(*t* ' = ' *E.addr* '*' *L.type.width*); } *gen*(*L.addr* ' = ' L_1*. addr* ' + ' *t*); }

5) Verificação de tipo

- Verifica o tipo de uma expressão a partir de sub-expressões e valida as expressões com base em regras lógicas chamadas de sistema de tipo.
- Uma implementação de uma linguagem é fortemente tipada se um compilador garantir que os programas que ela aceita executarão sem erros de tipo.

  Ex: Java é compilado para bytecode independente da máquina, os quais incluem informações detalhadas dos tipos relativos às operações, sendo verificado antes da permissão de execução.

5.1) Regras para verificação de tipo

1. **Síntese**: constrói o tipo de uma expressão a partir do tipo das sub-expressões. Exige que nomes precisam ser declaradas antes do uso.

   ex: **E^1 + E^2** é definido a partir do tipo das sub-expressões **E^1** e **E^2**.

2. **Inferência**: determina o tipo através do modo de que a expressão é usada.

   ex: **null** é uma função que testa se uma lista está vazia, portanto a partir do uso **null(x)** podemos dizer que **x** deve ser uma **lista**. O tipo de **x** é desconhecido, o que precisamos saber é que **x** é uma **lista** de **algum tipo**.

obs: a **inferência de tipo** é necessária para linguagens do tipo **ML**, que **verificam tipos**, mas não exigem que **nomes** sejam declarados.

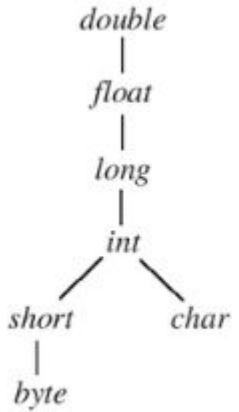
5.2) Conversões de tipo

Considere:

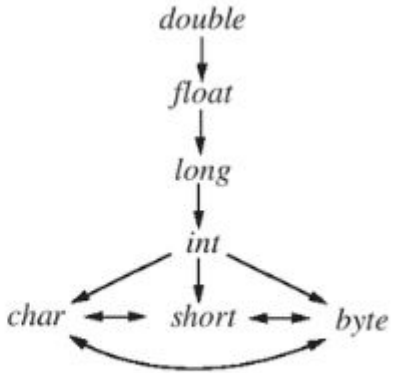
int i; float x; x+i;

```
t1 = (float) 2
t2 = t1 * 3.14
```

if (E_1.*type* = *integer* **and** E_2.*type* = *integer*) E.*type* = *integer*;
else if (E_1.*type* = *float* **and** E_2.*type* = *integer*) ...



(a) Conversões de alargamento



(b) Conversões de estreitamento

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w )    return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else  error;
}
```

5.3) Sobrecarga de funções e operadores

- Um símbolo sobrecarregado possui diferentes significados, dependendo de seu contexto.

- A sobrecarga é resolvida quando um único significado é determinado para cada ocorrência de um nome.

ex: Podemos escolher entre as duas versões da função examinando seus argumentos.

```
void err() {... }
void err(String s) {... }
```

if f pode ter tipo $s_i \rightarrow t_i$, para $1 \leq i \leq n$, onde $s_i \neq s_j$ para $i \neq j$
and x tem tipo s_k, para algum $1 \leq k \leq n$
then expressão $f(x)$ tem tipo t_k

5.4) Inferência de tipo e funções polimórficas

- Os elementos de uma mesma lista tem de ter todos os elementos de um mesmo tipo, porém, length pode ser aplicada em listas de qualquer tipo.

$$length([\text{“sun”}, \text{“mon”}, \text{“tue”}]) + length([10,9,8,7])$$

Usando o símbolo ∀ (lido como ‘para qualquer tipo’) e o construtor de tipo *list*, o tipo de *length* pode ser escrito como

$$\forall \alpha.\ list(\alpha) \rightarrow integer \qquad (6.12)$$

- Cada vez que uma função polimórfica for aplicada, suas variáveis de tipo ligadas podem denotar um tipo diferente.
- Durante a verificação de tipo, em cada uso de um tipo polimórfico, substituímos as variáveis ligadas por novas e removemos os quantificadores universais.

Referências

- AHO, Alfred V. .. [et al.]. Compiladores: princípios, técnicas e ferramentas - 2ª edição.

Geração de código intermediário

Equipe:
Daniel Lucas Santana Santos
Valber Francisco dos Santos