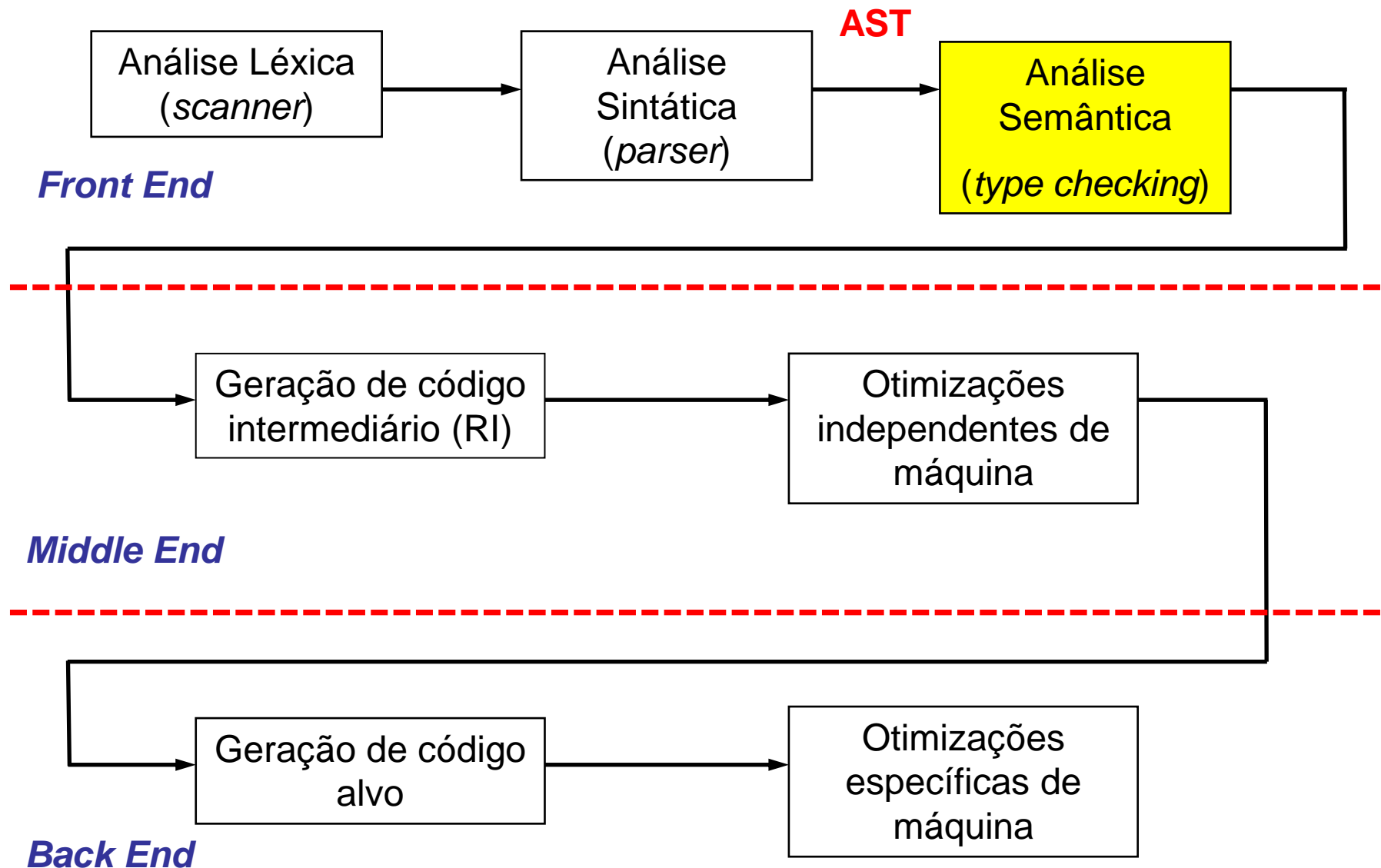

Análise Semântica

Fluxo do Compilador



Front-End do Compilador

Tipo de Erro	Exemplo	Detectado Por:
Léxico	...@...	Analizador Léxico
Sintático	...x*%...	Analizador Sintático
Semântico	... int x; y=x(3);	Analizador Semântico
Corretude	O seu programa	Testador/Usuário

Análise Semântica

- Verificar se todos os identificadores estão declarados
- Verificar redefinição de identificadores
- Verificar tipos
- Verificar se protótipos correspondem às funções declaradas
- Verificar Expressões matemáticas
- Armazenar informações sobre variáveis
- Verificar se os identificadores estão sendo usados da forma correta
- Verificar o escopo das variáveis
- Construir a AST durante a verificação semântica

Análise Semântica

Escopo de Identificadores

Análise Semântica: Escopo

Escopo de um Identificador

O escopo de um identificador é a porção do programa onde este identificador é acessível.

```
int foo(int a)
{
    int b;
    return (a+b) ;
}

int main()
{
    return foo(666) ;
}
```

Análise Semântica: Escopo

Escopo de um Identificador

As linguagens de programação podem ter escopo estático ou dinâmico.

Escopo Estático: O escopo do identificador depende apenas do texto do programa fonte. Ex.: Algol, Pascal, C

Escopo Dinâmico: O escopo do identificador depende da execução do programa. Ex.: Lisp, SNOBOL

Análise Semântica: Escopo

Escopo de um Identificador

As linguagens de programação podem ter escopo estático ou dinâmico.

Escopo Estático: O escopo do identificador depende apenas do texto do programa fonte. Ex.: Algol, Pascal, C

Escopo Dinâmico: O escopo do identificador depende da execução do programa. Ex.: Lisp, SNOBOL

Análise Semântica: Escopo

Escopo de um Identificador

Um mesmo identificador pode se referir a coisas distintas em diferentes partes do programa.

```
int foo(int a)
{
    return (a);
}

int main()
{
    int a = 666;
    return foo(a);
}
```

Análise Semântica: Escopo

Escopo de um Identificador

```
int a;
```

```
int foo(int foo, int a)
{
    return (foo+a);
}
```

O programa está correto?

Análise Semântica: Escopo

Escopo de um Identificador

```
int a;  
char a;
```

```
int foo(int foo, int a)  
{  
    int a;  
    return (foo+a);  
}
```

O programa está correto?


Análise Semântica: Escopo

Escopo de um Identificador

```
int a;  
char a;
```



identificador a redefinido

```
int foo(int foo, int a)  
{  
    int a;  identificador a redefinido  
    return (foo+a);  
}
```

Como controlar o escopo dos identificadores e descobrir redefinições dos mesmos?

Tabelas de Símbolos

Tabela de Símbolos: Para que serve?

Guardar informação para cada identificador que aparece no programa:

```
int a;  
int v[10][15];  
  
void func(int a, char* b);  
  
int foo(int foo, int a)  
{  
    return func(666, "Skynet is Online", v[-1]['a'][77]);  
}  
  
int main()  
{  
    char* c = "AAdeus mundo!" + 1;  
    printf("%s\n", c);  
    return 0;  
}
```

Tabela de Símbolos: O que armazenar?

variáveis: nome, tipo, dimensão, se são locais ou não, se são parâmetros de função, se estão inicializadas.

funções: nome, se possui parâmetros, tipo de retorno.

protótipos: nome, parâmetros, tipo de retorno.

constantes: nome, valores.

Tabela de Símbolos: Como é usada?

Ao declarar uma variável:

```
int y;
```

- `int` é um tipo válido?
- Já existe algum outro identificador neste escopo chamado `y`?

Ao declarar uma Função:

```
void function (int k, char* s)
```

- `void` é um tipo válido ?
- Já existe algum outro identificador neste escopo chamado `function`?
- Se existe algum protótipo para `function`, ele corresponde a declaração da função, em relação ao tipo de retorno e aos parâmetros?

Tabela de Símbolos: Como é usada?

Em constantes:

```
#define a 1+5+c;
```

- Já existe algum outro identificador neste escopo chamado **a**?
- O identificador **c** já está declarado e é uma constante?

Em comandos:

```
x[i+9] = y(10, "teste");
```

- O identificador **x** está declarado?
- O identificador **x** é um **array**? Se sim, ele possui só uma dimensão?
- O identificador **i** está declarado e é do tipo **int**?
- O identificador **y** está declarado?
- O identificador **y** é uma função? Se sim, possui exatamente 2 parâmetros?
- O parâmetros passados para **y** tem o tipo correto?

Tabela de Símbolos: Implementação

Lista dinâmica

NAME	ATTRIBUTES
s o r t	
a	
r e a d a r r a y	
i	

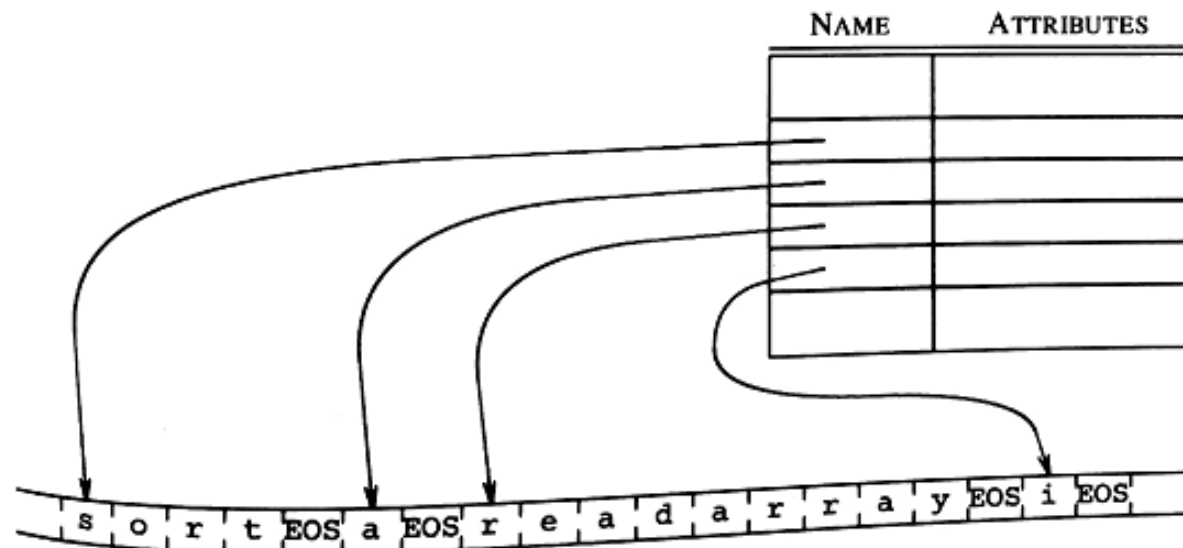


Tabela de Símbolos: Implementação

Lista dinâmica

Não é eficiente!!!

NAME	ATTRIBUTES
s o r t	
a	
r e a d a r r a y	
i	

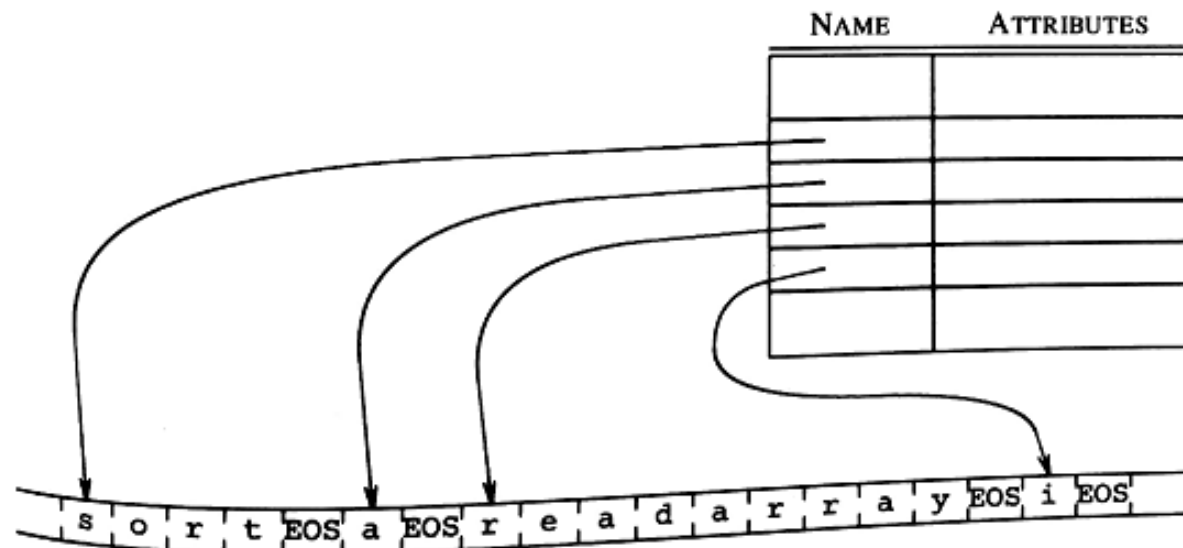
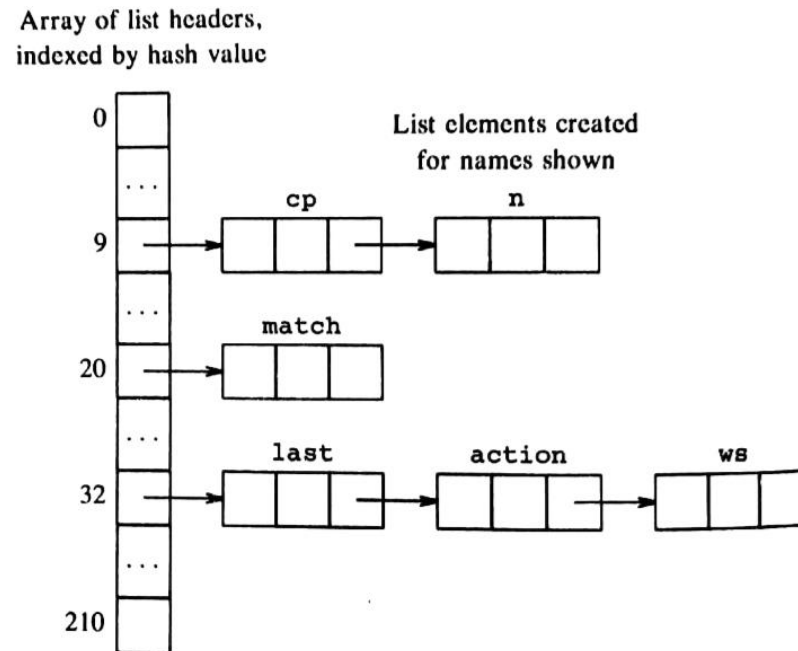


Tabela de Símbolos: Implementação

Tabela Hash



1. Um **array** de tamanho fixo com **m** entradas, que são constituídas por ponteiros, sendo que **m** geralmente é um número primo.
2. Cada entrada, chamada de **bucket**, contém um ponteiro para uma lista ligada de registros com informações sobre os símbolos

Tabela de Símbolos

Como armazenar/buscar informações?

```
int main()  
{  
    int a, A, chuchu, abobrinha;  
    char abacate, c;  
  
    return 0;  
}
```

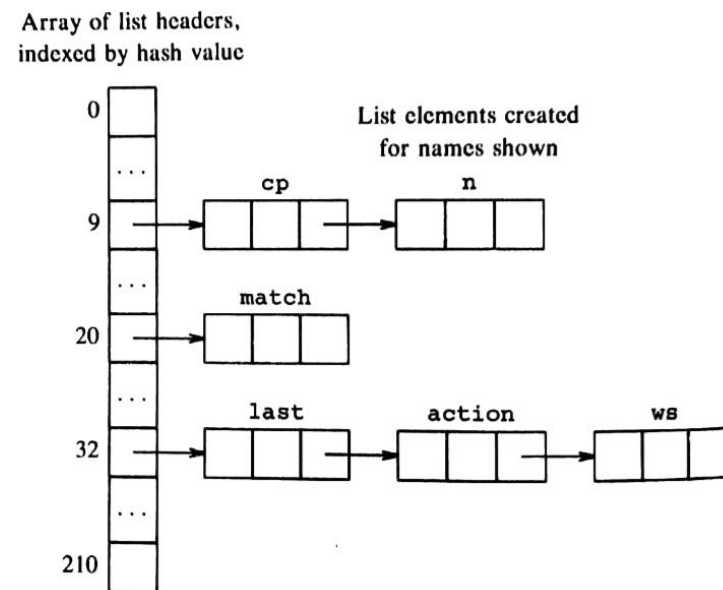


Tabela de Símbolos

Exemplo de Função Hash:

chuchu

Código ASCII dos caracteres:

$$c = 99 \quad h = 104 \quad u = 117$$

$$\text{chuchu} = 99 + 104 + 117 + 99 + 104 + 117 = 640$$

Resto da divisão de 640 por 211 = 7

A posição 7 da tabela hash é utilizada para armazenar o identificador.

Análise Semântica

Criação da AST

Árvore Sintática Abstrata (AST)

Dada uma Gramática Livre de Contexto, uma árvore de derivação é uma árvore rotulada cuja raiz tem sempre o não-terminal inicial como rótulo. Em uma derivação

$$S \Rightarrow a_1 a_2 \dots a_n$$

o nó S tem como filhos n nós rotulados de a_1 até a_n que são símbolos terminais ou não-terminais.

O *parser* também pode construir **A**bstract **S**yntax **T**rees:

- Estas árvores refletem a estrutura sintática do programa
- São a interface entre o *parser* e as próximas fases da compilação
- Já abstraem detalhes irrelevantes para as fases seguintes. Ex.: pontuação.
- À medida que um não terminal é reconhecido, um nó da árvore é criado
- De baixo para cima (em *parsers* LR)
- Cada execução de um dos métodos associados aos não-terminais cria um nó da árvore e “pendura” nele os nós filhos.

Árvore Sintática Abstrata (AST)

Considere a gramática:

$$E \rightarrow \textit{int} \mid (E) \mid E + E$$

E a cadeia:

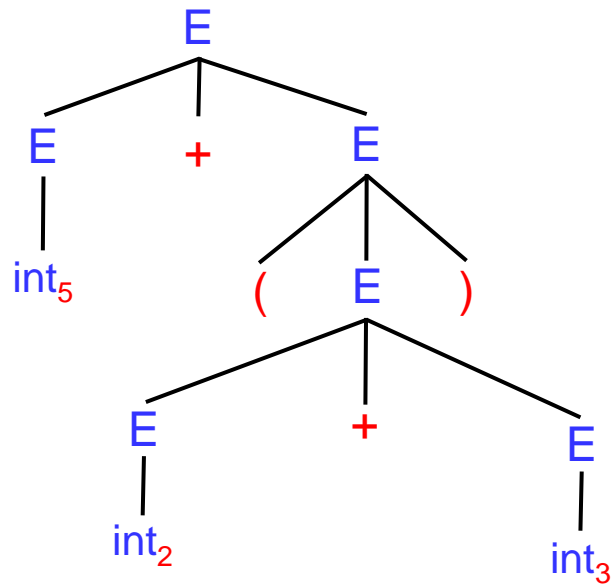
5 + (2 + 3)

O analisador léxico irá produzir a seguinte sequencia de *tokens*:

*int*₅ '+' '(' *int*₂ '+' *int*₃ ')'

Árvore de Derivação

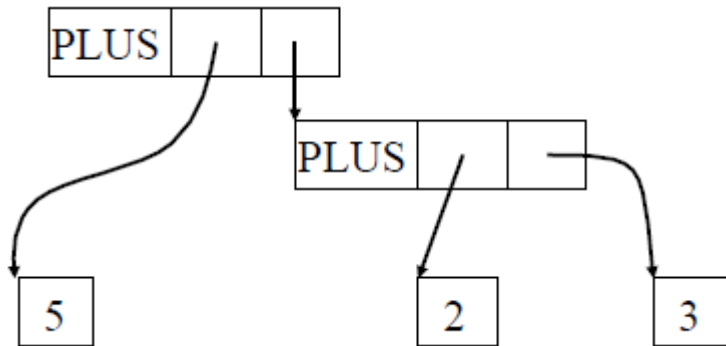
O analisador sintático “constrói” a árvore de derivação:



- Detalha o processo de funcionamento do *parser*
- Captura a estrutura de aninhamento do programa
- Mas contém informações desnecessárias:
 - Parênteses
 - Nós não-terminais

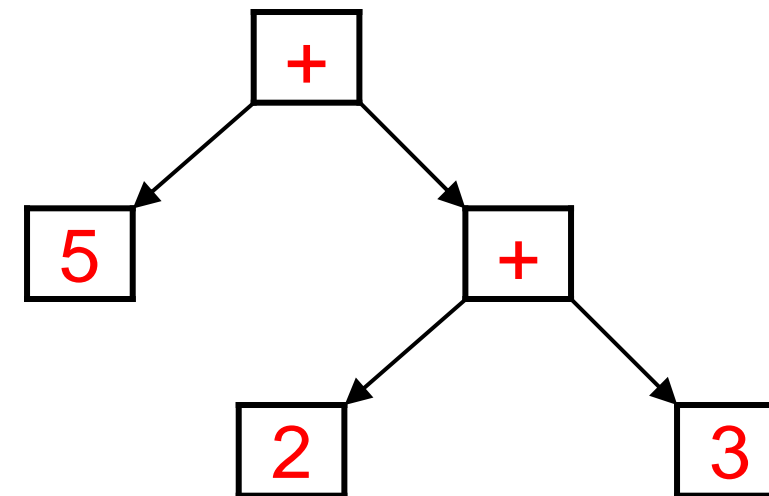
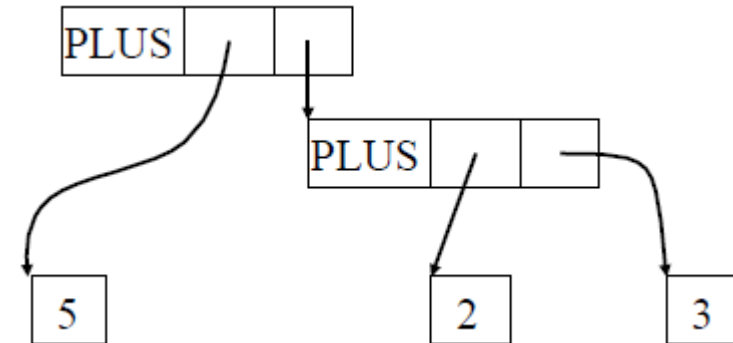
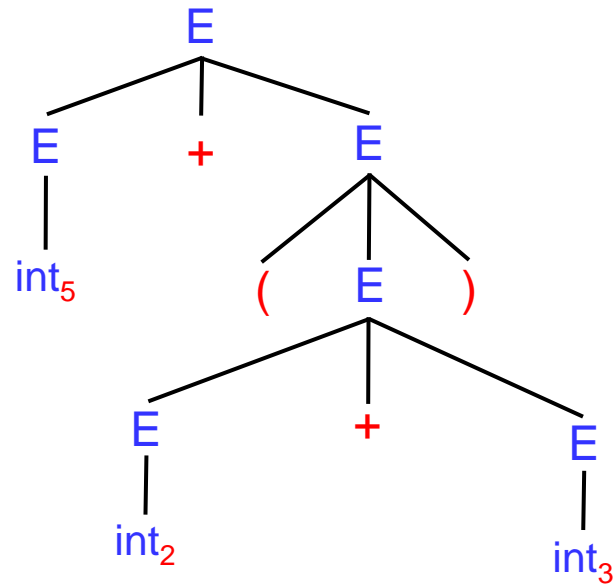
Árvore Sintática Abstrata (AST)

O análise semântica (em conjunto com o parser) constrói, de fato, a árvore sintática abstrata:



- Também captura a estrutura de aninhamento do programa
- É uma abstração da sintaxe da gramática, mais compacta e simples de usar
- É uma importante estrutura de dados do compilador

5 + (2 + 3)



Tradução Dirigida Pela Sintaxe: Ações Semânticas

A **AST** é construída utilizando **Tradução Dirigida pela Sintaxe**, através de ações semânticas.

Nas ações semânticas tem-se:

- Cada símbolo da gramática pode possuir vários atributos associados a ele.
- No caso de símbolos terminais, o valor de tais atributos é determinado pelo analisador léxico.
- Cada produção da gramática possui uma ação associada:

$$X \rightarrow Y_1 Y_2 \dots Y_n \{ \text{action} \}$$

- As ações podem se referir ou computar atributos de símbolos da gramática.

Análise Semântica: Criação da AST

Cada elemento da linguagem fonte, vai possuir uma estrutura na árvore para o representar.

Tal estrutura pode ser, por exemplo, uma `struct` ou uma `classe`.

- Pode haver uma estrutura específica para cada elemento ou várias estruturas genéricas que se adaptam a vários elementos.

Análise Semântica: Criação da AST

Vamos ver alguns exemplos para o C...

Comando IF

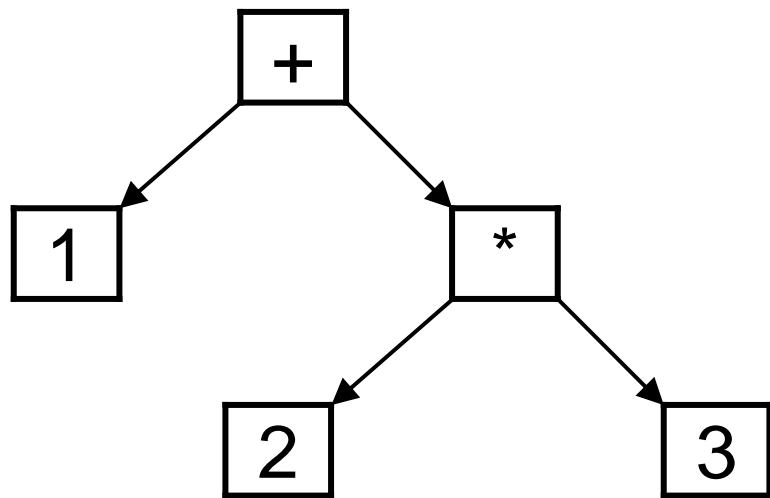
Comando WHILE

Comando Expressão

Partindo inicialmente de uma estrutura específica e indo para uma mais geral...

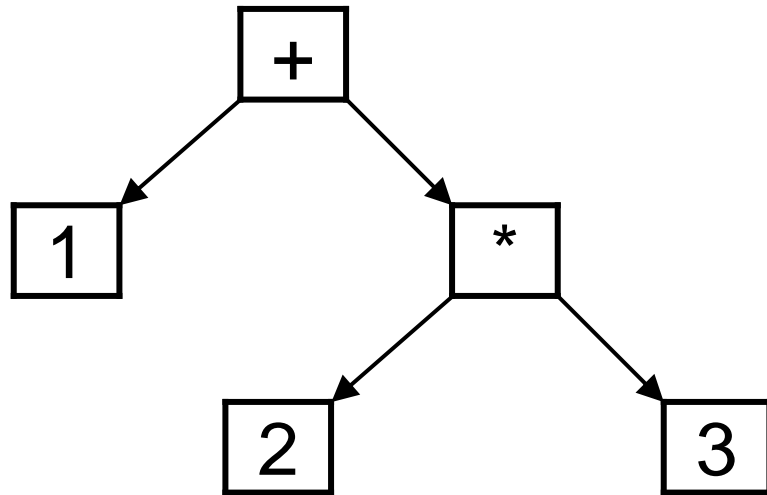
Criação da AST: expressão

Expressão Matemática: $1+2*3$



Criação da AST: expressão

Expressão Matemática: $1+2*3$



```
struct expression. /* expressão */
{
    int exp_type; /* operando/operador */
    //criar variáveis para armazenar:
    // - valores constantes
    // - identificadores
    // - parâmetros para funções
    // - dimensões para array
    // - etc
    struct expression *left_child;
    struct expression *right_child;
};
```

Criação da AST: if

Comando IF

Criação da AST: if

Comando IF

- Expressão matemática
- Lista de comandos para THEN
- Lista de comandos para ELSE

IF
exp →
then →
else →

```
struct cmd /* comando if */
{
    int cmd_type;
    struct expression *exp;
    struct cmd* then;
    struct cmd* else;
    struct cmd* next;
};
```

Criação da AST: while

Comando WHILE

Criação da AST: while

Comando WHILE

- Expressão matemática
- Lista de comandos

WHILE
exp →
comandos →

```
struct cmd /* comando while */
{
    int cmd_type;
    struct expression *exp;
    struct cmd* comandos;
    struct cmd* next;
};
```

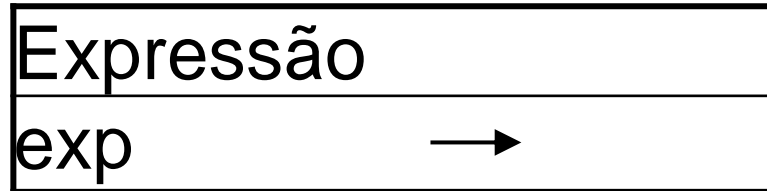
Criação da AST: expressão

Comando Expressão

Criação da AST: expressão

Comando Expressão

- Expressão matemática



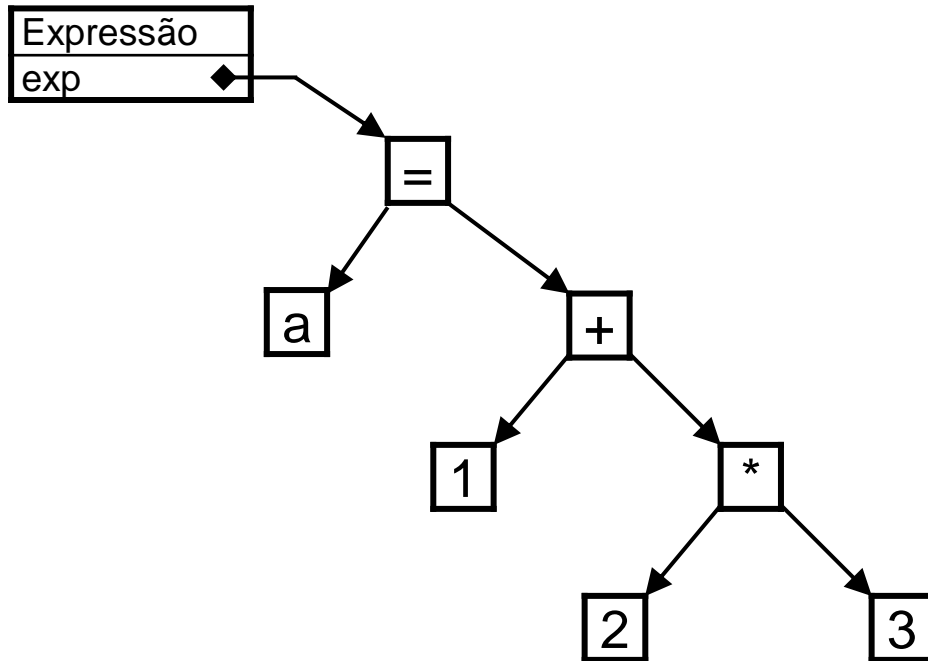
```
struct cmd /* comando expressão */  
{  
    int cmd_type;  
    struct expression *exp;  
    struct cmd* next;  
};
```

Criação da AST: expressão

Comando Expressão: $a = 1 + 2 * 3;$

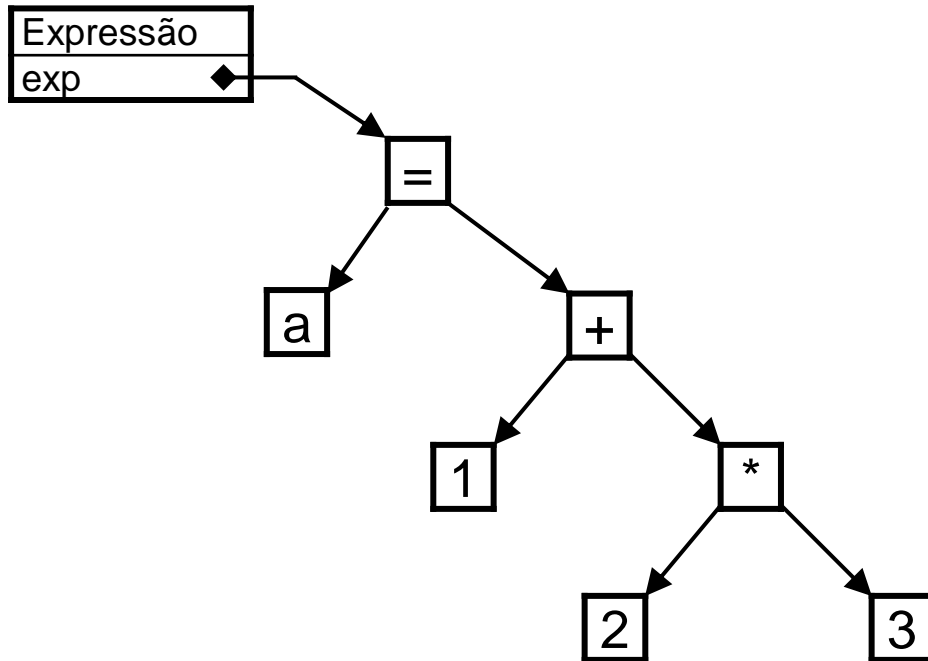
Criação da AST: expressão

Comando Expressão: $a = 1 + 2 * 3;$



Criação da AST: expressão

Comando Expressão: $a = 1 + 2 * 3;$



```
struct cmd /* comando expressão */
{
    int cmd_type;
    struct expression *exp;
    struct cmd* next;
};

struct expression. /* expressão */
{
    int exp_type; /* operando/operador */
    //criar variáveis para armazenar:
    // - valores constantes
    // - identificadores
    // - parâmetros para funções
    // - dimensões para array
    // - etc
    struct expression *left_child;
    struct expression *right_child;
};
```

Criação da AST: Estruturas Genéricas

Estrutura genérica para IF, WHILE e Expressão:

```
struct cmd /* comando genérico */
{
    //tipo de comando
    int cmd_type;

    //comandos expressão/if/while
    struct expression *exp;

    //comando if
    struct cmd* then;
    struct cmd* else;

    //comando while
    struct cmd* comandos;

    //próximo comando
    struct cmd* next;
};
```

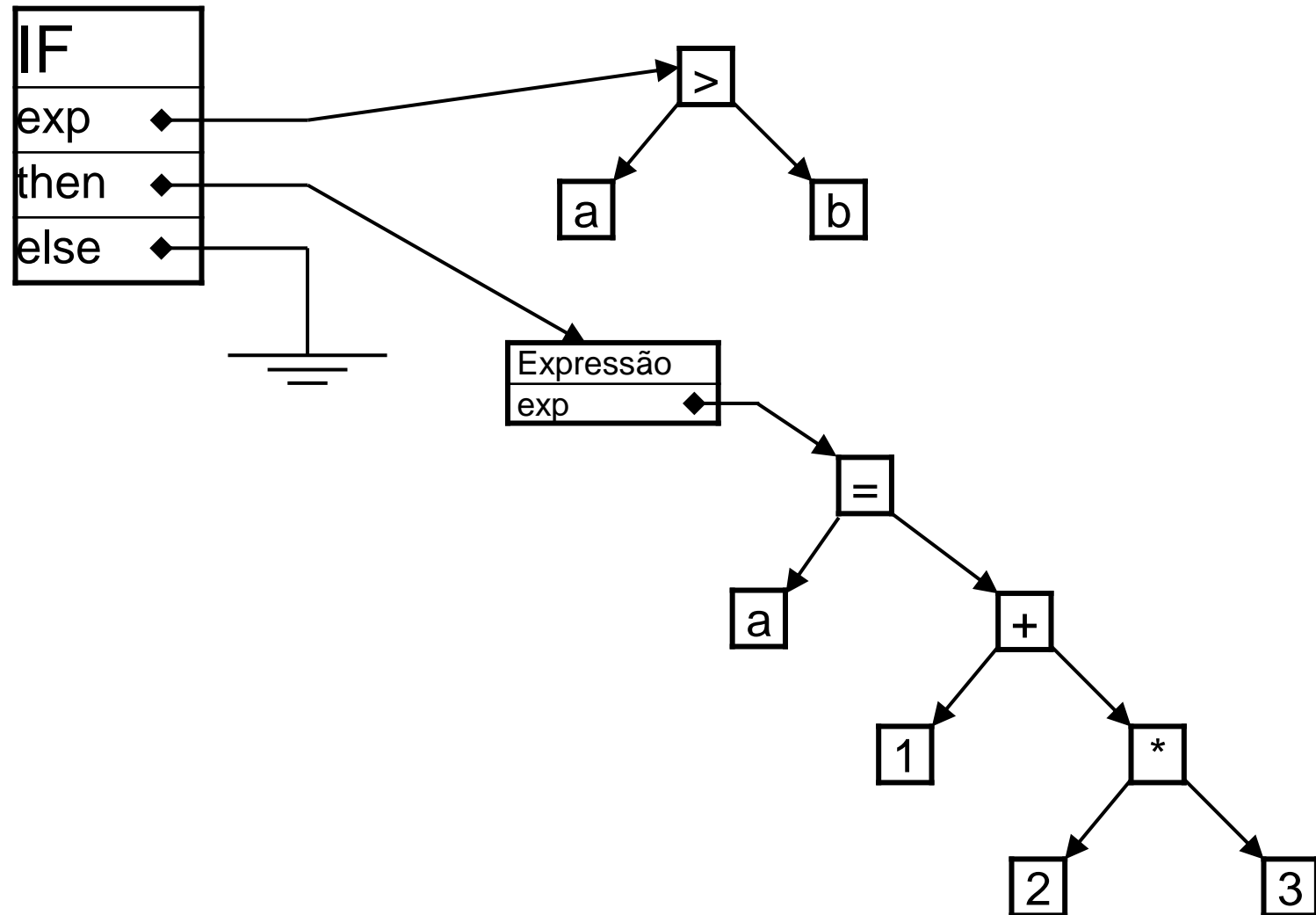
```
struct expression. /* expressão */
{
    int exp_type; /* operando/operador */
    //criar variáveis para armazenar:
    // - valores constantes
    // - identificadores
    // - parâmetros para funções
    // - dimensões para array
    // - etc
    struct expression *left_child;
    struct expression *right_child;
};
```

Criação da AST

```
if (a>b)
{
    a = 1+2*3;
}
```

Criação da AST

```
if (a>b)
{
  a = 1+2*3;
}
```

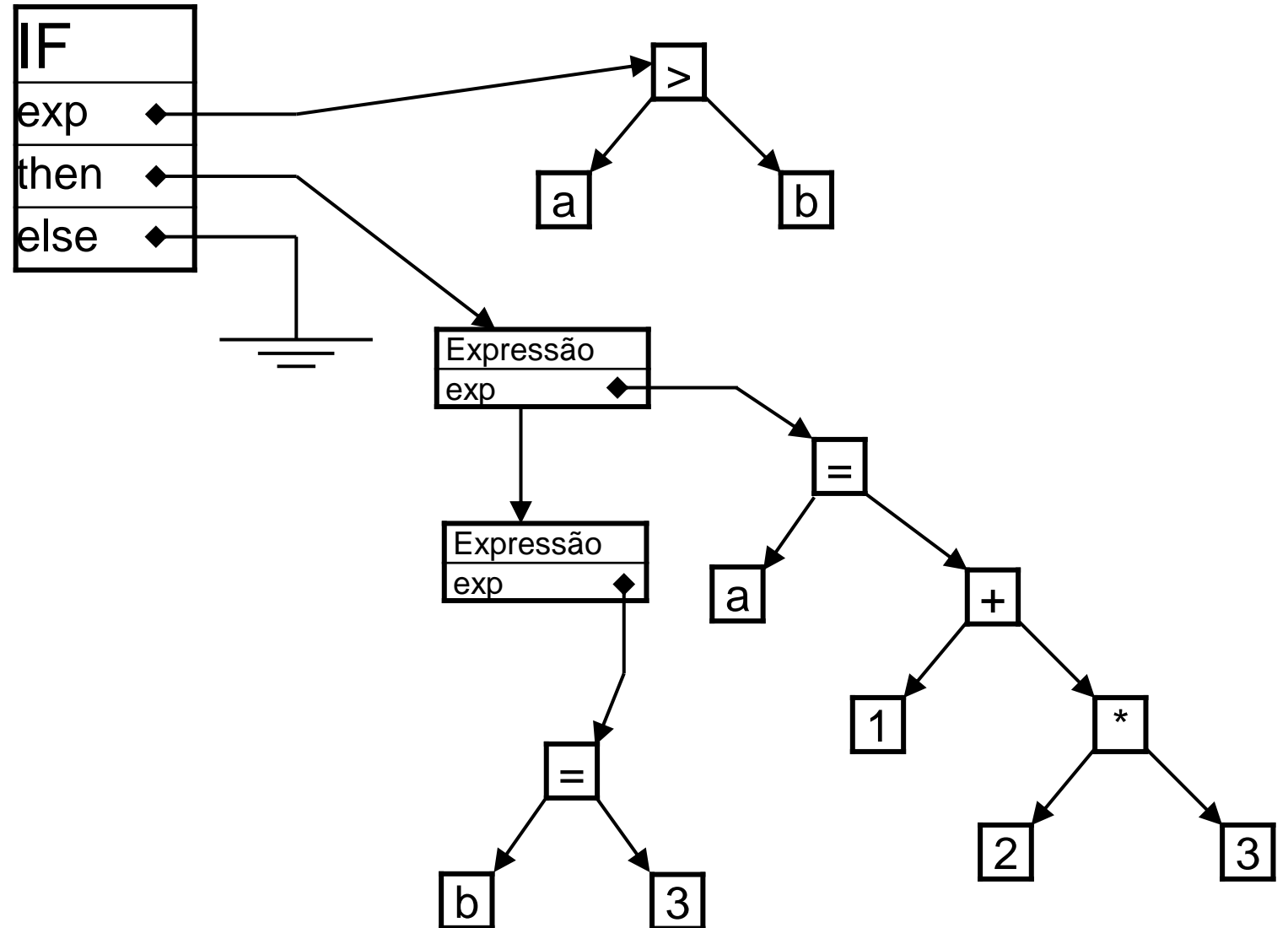


Criação da AST

```
if (a>b)
{
    a = 1+2*3;
    b = 3;
}
```

Criação da AST

```
if (a>b)
{
  a = 1+2*3;
  b = 3;
}
```

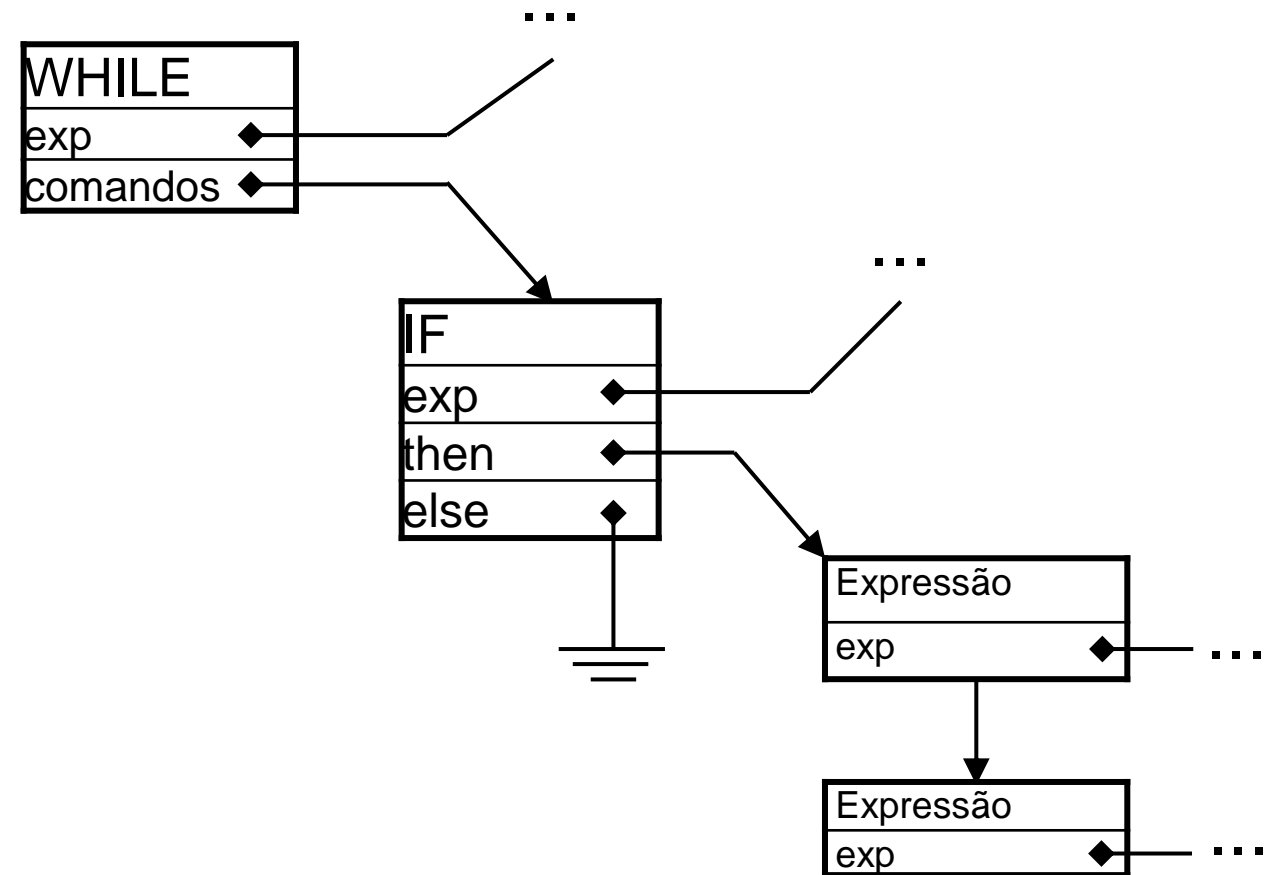


Criação da AST

```
while (a<b)
{
    if (a>b)
    {
        a = 1+2*3;
        b = 3;
    }
}
```

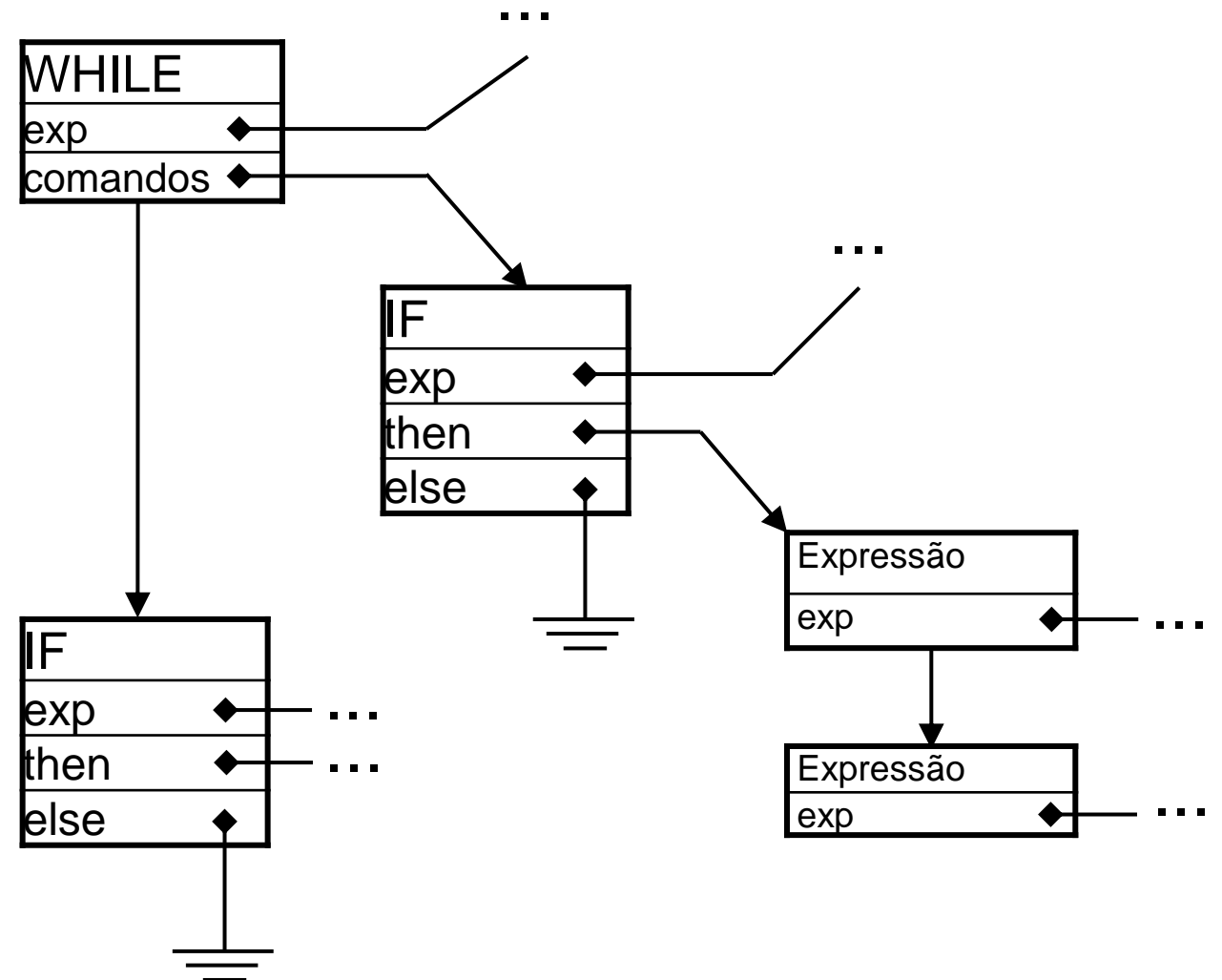

Criação da AST

```
while (a<b)
{
    if (a>b)
    {
        a = 1+2*3;
        b = 3;
    }
}
```



Criação da AST

```
while (a<b)
{
    if (a>b)
    {
        a = 1+2*3;
        b = 3;
    }
    if (b)
    {
        while (a)
        {
            b = b+1;
            a = a-1;
        }
    }
}
```



Criação da AST: Bison

```
lista_de_comandos: comandos comandos_prime
;

comandos_prime: comandos comandos_prime
|
;

comandos: WHILE ( expressao ) '{' lista_de_comandos '}'
| IF ( expressao ) '{' lista_de_comandos '}'
| IF ( expressao ) '{' lista_de_comandos '}' ELSE '{' lista_de_comandos '}'
| expressao ';'
;

expressao: expressao_aditiva
;

expressao_aditiva: expressao_primaria
| expressao_aditiva + expressao_primaria
| expressao_aditiva - expressao_primaria
;

expressao_primaria: IDENTIFICADOR
| numero
| CARACTERE
| ( expressao )
;

numero: NUM_INTEGER
| NUM_HEX
;
```

Criação da AST: Bison – Expressões Matemáticas

```
expressao: expressao_aditiva  
;
```

```
expressao_aditiva: expressao_primaria  
                  | expressao_aditiva + expressao_primaria  
                  | expressao_aditiva - expressao_primaria  
;
```

```
expressao_primaria: IDENTIFICADOR  
                  | numero  
                  | CARACTERE  
                  | ( expressao )  
;
```

```
numero: NUM_INTEGER  
       | NUM_HEXA  
;
```

Criação da AST: Bison – Expressões Matemáticas

```
numero: NUM_INTEGER {struct expression* aux = (struct expression*) malloc(sizeof(struct expression));
    aux->exp_type = ... //valor para indicar que eh um operando
    aux->... //ajusta tudo o que precisa para criar um valor constante, seu tipo, etc
    aux->left_child = NULL; //Não tem filho esquerdo, pois eh noh folha
    aux->right_child = NULL; //Não tem filho direito, pois eh noh folha
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal numero
}

| NUM_HEXA {struct expression* aux = (struct expression*) malloc(sizeof(struct expression));
    aux->exp_type = ... //valor para indicar que eh um operando
    aux->... //ajusta tudo o que precisa para criar um valor constante, seu tipo, etc
    aux->left_child = NULL; //Não tem filho esquerdo, pois eh noh folha
    aux->right_child = NULL; //Não tem filho direito, pois eh noh folha
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal numero
}

;
```

```
struct expression. /* expressão */
{
    int exp_type; /* operando/operador */
    //criar variáveis para armazenar:
    // - valores constantes
    // - identificadores
    // - parâmetros para funções
    // - dimensões para array
    // - etc
    struct expression *left_child;
    struct expression *right_child;
};
```

Criação da AST: Bison – Expressões Matemáticas

```
expressao_primaria: IDENTIFICADOR {struct expression* aux = (struct expression*) malloc(sizeof(struct expression));
    aux->exp_type = ... //valor para indicar que eh um operando
    aux->... //ajusta tudo o que precisa para indicar que o noh tem um IDENTIFICADOR
    aux->left_child = NULL; //Não tem filho esquerdo, pois eh noh folha
    aux->right_child = NULL; //Não tem filho direito, pois eh noh folha
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal expressao_primaria
}

| numero { $$ = $1; //retorna o ponteiro da estrutura recebida pelo não-terminal numero para o não-terminal expressao_primaria }

| CARACTERE {struct expression* aux = (struct expression*) malloc(sizeof(struct expression));
    aux->exp_type = ... //valor para indicar que eh um operando
    aux->... //ajusta tudo o que precisa para criar um valor constante, seu tipo, etc
    aux->left_child = NULL; //Não tem filho esquerdo, pois eh noh folha
    aux->right_child = NULL; //Não tem filho direito, pois eh noh folha
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal expressao_primaria
}

| ( expressao ) {
    $$ = $2; //retorna o ponteiro da estrutura recebida pelo não-terminal expressao para o não-terminal expressao_primaria
}

;
```

```
struct expression. /* expressão */
{
    int exp_type; /* operando/operador */
    //criar variáveis para armazenar:
    // - valores constantes
    // - identificadores
    // - parâmetros para funções
    // - dimensões para array
    // - etc
    struct expression *left_child;
    struct expression *right_child;
};
```

Criação da AST: Bison – Expressões Matemáticas

```
expressao: expressao_aditiva {$$ = $1; /*retorna o ponteiro da estrutura recebida pelo não-terminal expressao_aditiva para o não-terminal expressao */ }
;

expressao_aditiva: expressao_primaria {
    $$ = $1; //retorna o ponteiro da estrutura recebida pelo não-terminal expressao_primaria
               //para o não-terminal expressao_aditiva
}

| expressao_aditiva + expressao_primaria {struct expression* aux = (struct expression*) malloc(sizeof(struct expression));
    aux->exp_type = ... //valor para indicar que eh o operador de SOMA
    aux->left_child = $1; //ponteiro do filho esquerdo no não-terminal expressao_aditiva
    aux->right_child = $3; //ponteiro do filho direito no não-terminal expressao_primaria
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal expressao_aditiva
}

| expressao_aditiva - expressao_primaria {struct expression* aux = (struct expression*) malloc(sizeof(struct expression));
    aux->exp_type = ... //valor para indicar que eh o operador de SUBTRACAO
    aux->left_child = $1; //ponteiro do filho esquerdo no não-terminal expressao_aditiva
    aux->right_child = $3; //ponteiro do filho direito no não-terminal expressao_primaria
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal expressao_aditiva
}
;
```

```
struct expression. /* expressão */
{
    int exp_type; /* operando/operador */
    //criar variáveis para armazenar:
    // - valores constantes
    // - identificadores
    // - parâmetros para funções
    // - dimensões para array
    // - etc
    struct expression *left_child;
    struct expression *right_child;
};
```

Criação da AST: Bison – Comandos

```
lista_de_comandos: comandos comandos_prime
;

comandos_prime: comandos comandos_prime
|
;

comandos: WHILE ( expressao ) '{' lista_de_comandos '}'
| IF ( expressao ) '{' lista_de_comandos '}'
| IF ( expressao ) '{' lista_de_comandos '}' ELSE '{' lista_de_comandos '}'
| expressao ';'
;
```


Criação da AST: Bison – Comandos

```
comandos: WHILE ( expressao ) '{' lista_de_comandos '}'
{
    struct cmd* aux = (struct cmd*) malloc(sizeof(struct cmd));
    aux->cmd_type = ... //valor para indicar o comando WHILE
    aux->expressao = $3; //retorna o ponteiro da estrutura recebida pelo não-terminal expressao, esse ponteiro
                        //contém a estrutura de uma árvore de expressão já completamente montada.
    aux->comandos = $6; //retorna o ponteiro da estrutura recebida pelo não-terminal lista_de_comandos,
                        //esse ponteiro contém uma lista de comandos já completamente montada.
    aux->next = NULL; //neste momento esse ponteiro eh NULL pois a lista é montada em outro lugar.
    aux->... = NULL; //atribui NULL aos demais campos não utilizados.
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal comandos
}
| IF ( expressao ) '{' lista_de_comandos '}' { /* código semelhante a produção apresentada abaixo */ }
| IF ( expressao ) '{' lista_de_comandos '}' ELSE '{' lista_de_comandos '}'
{
    struct cmd* aux = (struct cmd*) malloc(sizeof(struct cmd));
    aux->cmd_type = ... //valor para indicar o comando IF
    aux->expressao = $3; //retorna o ponteiro da estrutura recebida pelo não-terminal expressao, esse
                        //ponteiro contém a estrutura de uma árvore de expressão já completamente montada.
    aux->then = $6; //retorna o ponteiro da estrutura recebida pelo não-terminal lista_de_comandos, esse ponteiro
                    //contém uma lista de comandos já completamente montada para a parte verdadeira do IF
    aux->else = $10; //retorna o ponteiro da estrutura recebida pelo não-terminal lista_de_comandos, esse ponteiro
                    //contém uma lista de comandos já completamente montada para a parte falsa do IF
    aux->next = NULL; //neste momento esse ponteiro eh NULL pois a lista é montada em outro lugar.
    aux->... = NULL; //atribui NULL aos demais campos não utilizados.
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal comandos
}
| expressao ';'
{
    struct cmd* aux = (struct cmd*) malloc(sizeof(struct cmd));
    aux->cmd_type = ... //valor para indicar o comando EXPRESSAO
    aux->expressao = $1; //retorna o ponteiro da estrutura recebida pelo não-terminal expressao, esse
                        //ponteiro contém a estrutura de uma árvore de expressão já completamente montada.
    aux->next = NULL; //neste momento esse ponteiro eh NULL pois a lista é montada em outro lugar.
    aux->... = NULL; //atribui NULL aos demais campos não utilizados.
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal comandos
}
;
```

```
struct cmd /* comando genérico */
{
    //tipo de comando
    int cmd_type;

    //comandos expressão/if/while
    struct expression *exp;

    //comando if
    struct cmd* then;
    struct cmd* else;

    //comando while
    struct cmd* comandos;

    //próximo comando
    struct cmd* next;
};
```

Criação da AST: Bison – Comandos

```
lista_de_comandos: comandos comandos_prime {
    $1->next = $2; //O não-terminal comandos ($1) possui um ponteiro para a estrutura de um único comando,
                  //se houver um outro comando, haverá um ponteiro para o mesmo no não-terminal comandos_prime ($2),
                  //dessa forma, basta fazer $1->next = $2; para criar uma lista ligada.
    $$ = $1; //retorna o ponteiro da lista montada com todos os comandos para o não-terminal lista_de_comandos
}
;
```

```
comandos_prime: comandos comandos_prime {
    $1->next = $2; //O não-terminal comandos ($1) possui um ponteiro para a estrutura de um único comando,
                  //se houver um outro comando, haverá um ponteiro para o mesmo no não-terminal comandos_prime ($2),
                  //dessa forma, basta fazer $1->next = $2; para criar uma lista ligada.
    $$ = $1; //retorna o ponteiro da lista de comandos montada até agora para o não-terminal comandos_prime
}
| { $$ = (struct cmd*) NULL; /* retorna NULL pois não existem mais comandos. */ }
;
```

```
comandos: WHILE ( expressao ) '{' lista_de_comandos '}' { ... }
| IF ( expressao ) '{' lista_de_comandos '}' { ... }
| IF ( expressao ) '{' lista_de_comandos '}' ELSE '{' lista_de_comandos '}' { ... }
| expressao ';' { ... }
;
```

```
struct cmd /* comando genérico */
{
    //tipo de comando
    int cmd_type;

    //comandos expressão/if/while
    struct expression *exp;

    //comando if
    struct cmd* then;
    struct cmd* else;

    //comando while
    struct cmd* comandos;

    //próximo comando
    struct cmd* next;
};
```

Análise Semântica: Criação da AST

Exemplos simplificados para o C...

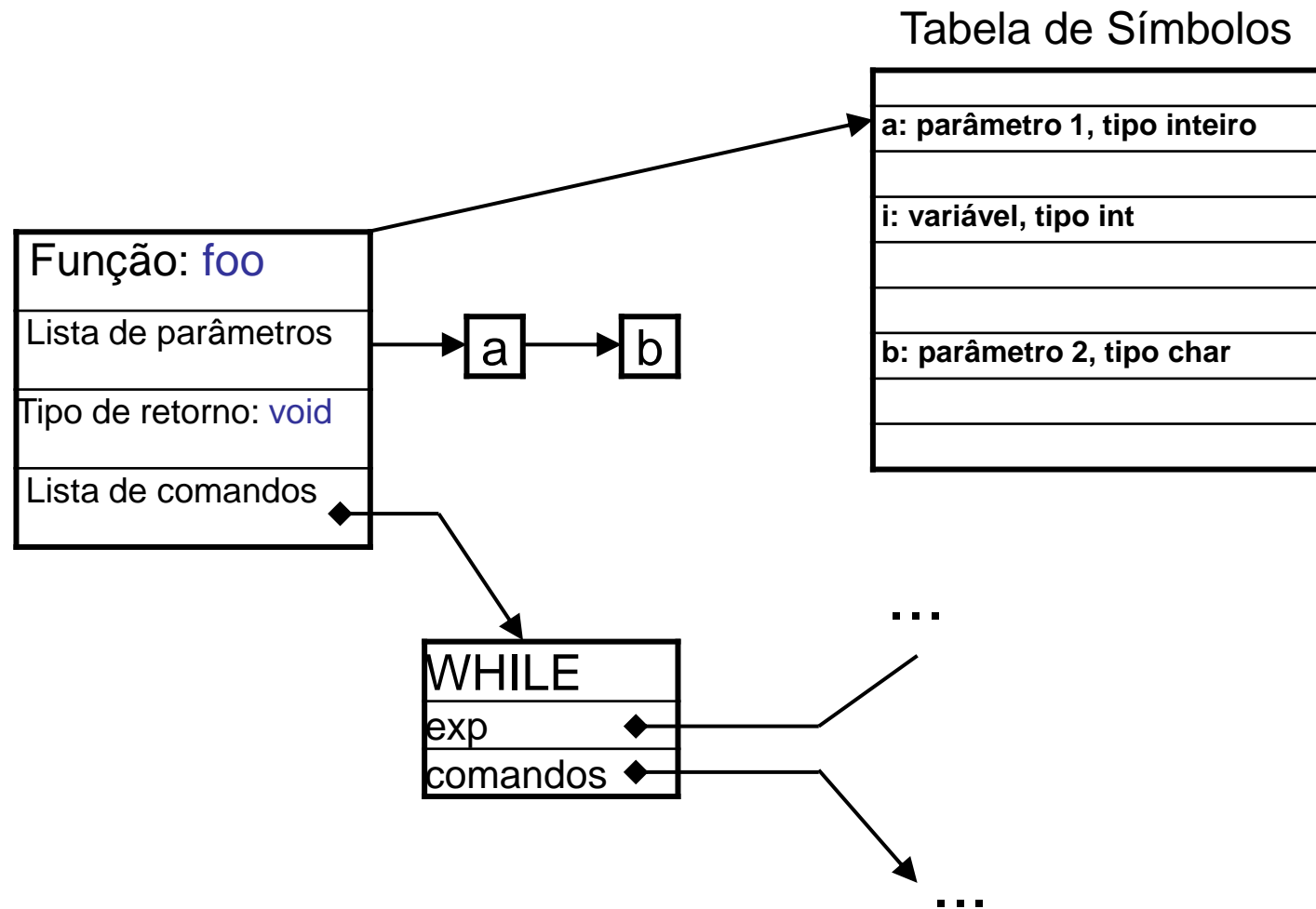
Programa
Funções

Criação da AST: Funções

```
void foo(int a, char b)
{
    int i;
    while(...)
    {...}
}
```

Criação da AST: Funções

```
void foo(int a, char b)
{
    int i;
    while(...)
    {...}
}
```



Criação da AST: Funções

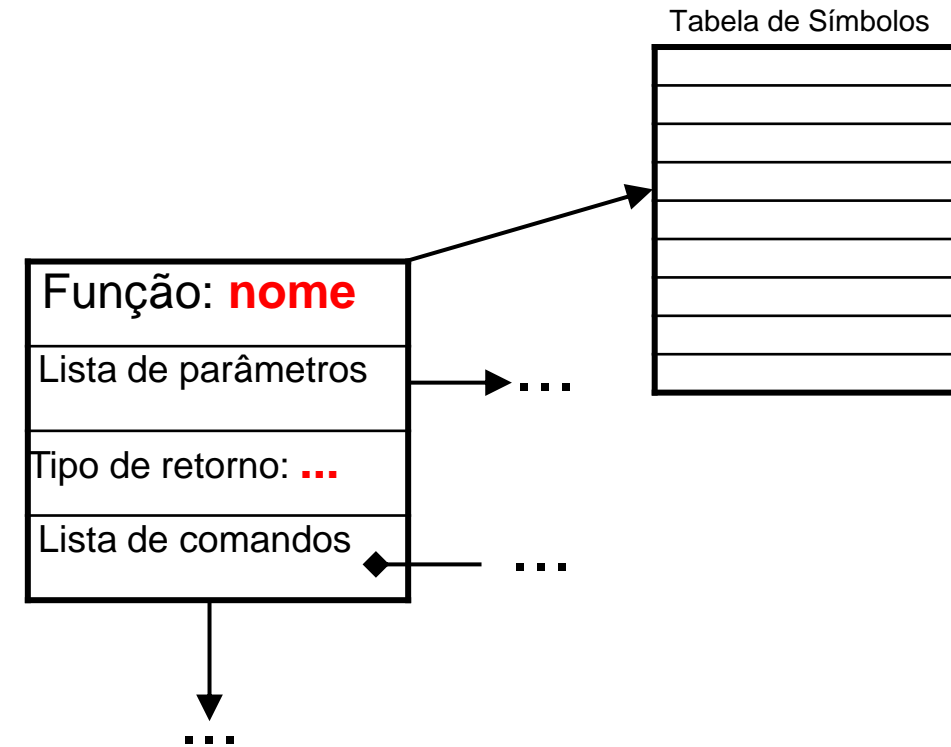
```
struct function /* estrutura para função */
{
    //nome da função
    char *nome;

    //tabela de símbolos local
    SYMBOL_TABLE_TYPE *symbolTable;

    //Criar variáveis para:
    // -ter informações sobre os parâmetros
    // -guardar o tipo de retorno
    // -qualquer outra informação necessária
    // -etc

    //lista de comandos
    struct cmd *lista_de_comandos;

    // próxima função
    struct function *next;
};
```

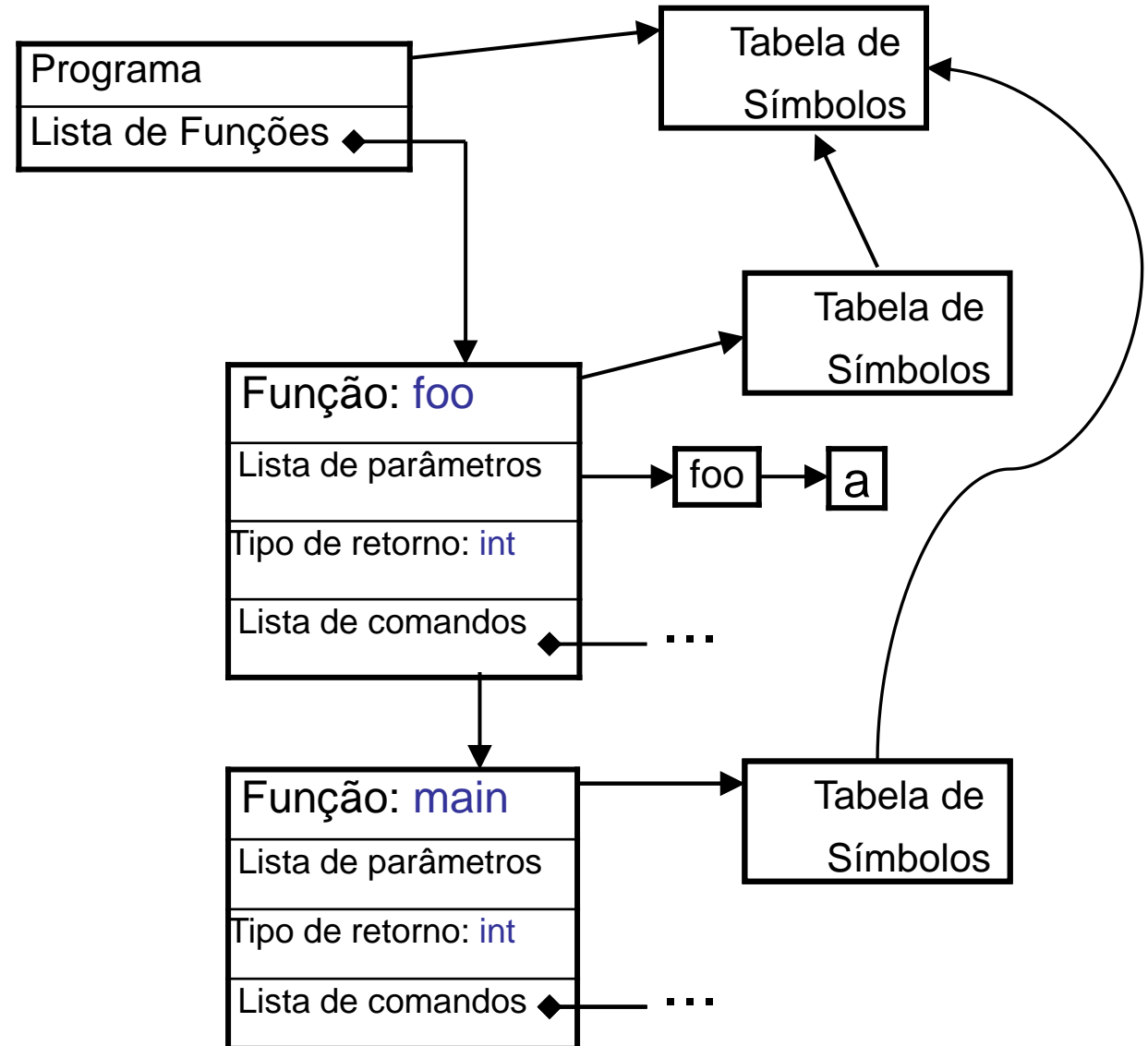


Criação da AST: Programa

```
#define c1 5
#define c2 c1+10
int a,b,c[5][5];

int foo(int foo, int a)
{
    return (foo+a);
}

int main()
{
    return foo(1,2);
}
```

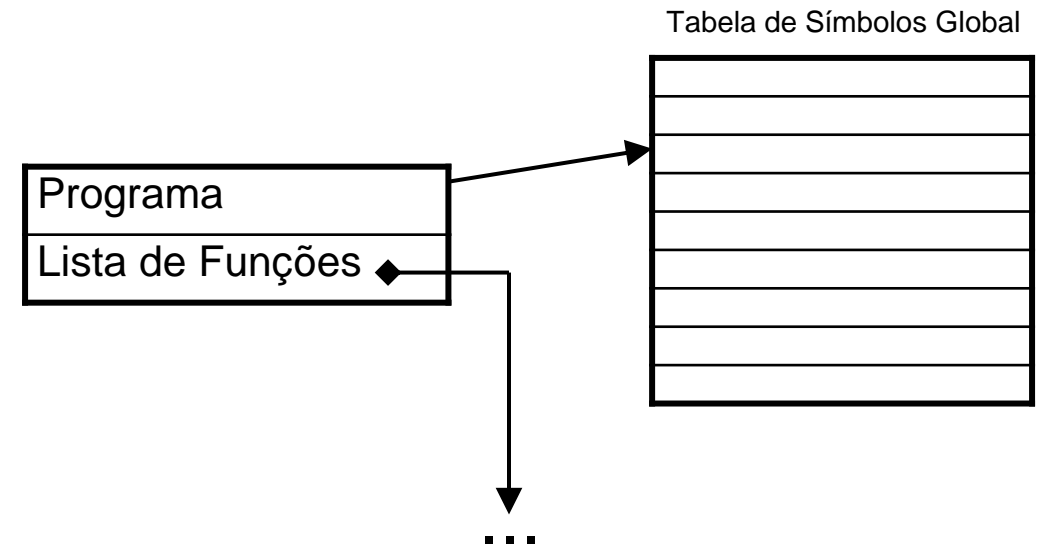


Criação da AST: Programa

```
struct program /* estrutura para programa */
{
    //tabela de símbolos global
    SYMBOL_TABLE_TYPE *globalSymbolTable;

    //Criar variáveis para outras
    //informações que sejam necessárias

    //lista de funções
    struct function *lista_de_funcoes;
};
```



Criação da AST: Bison

```
programa: declaracoes_globais lista_de_funcoes FIM_DE_ARQUIVO
;

lista_de_funcoes : funcao funcao_prime
;

funcao_prime: funcao funcao_prime
            |
;

funcao: tipo IDENTIFICADOR parametros '{' declaracoes_locais lista_de_comandos '}'
;

declaracoes_globais: ...
;
tipo: ...
;
parametros: ...
;
declaracoes_locais: ...
;
```

Criação da AST: Funções

```
funcao: tipo IDENTIFICADOR parametros '{' declaracoes_locais lista_de_comandos '}'  
;
```

Criação da AST: Funções

```
funcao: tipo IDENTIFICADOR parametros '{' declaracoes_locais lista_de_comandos '}'
{
    struct function* aux = (struct function*) malloc(sizeof(struct function));
    aux->nome = (char*) malloc(strlen(IDENTIFICADOR)+1);
    strcpy(aux->nome, IDENTIFICADOR); //guarda o nome da função
    aux->symbolTable = ... //cria e inicializa a tabela de símbolos local

    //as derivações e ações dos não-terminais parametros e declaracoes_locais irão armazenar os símbolos locais na tabela de símbolos
    //recém criada, além de verificar se existem identificadores duplicados, redefinidos, entre outros erros na declaração de símbolos.

    aux->lista_de_comandos = $6; //retorna o ponteiro da estrutura recebida pelo não-terminal
                                //lista_de_comandos, esse ponteiro contém uma lista de comandos
                                //já completamente montada.

    aux->... //ajusta os demais campos:
            // - tabela de símbolos
            // - parâmetros
            // - tipo de retorno
            // - etc

    aux->next = NULL; //neste momento esse ponteiro eh NULL pois a lista é montada em outro lugar.
    $$ = aux; //retorna o ponteiro da estrutura para o não-terminal funcao
}

;

struct function /* estrutura para função */
{
    //nome da função
    char *nome;

    //tabela de símbolos local
    SYMBOL_TABLE_TYPE *symbolTable;

    //Criar variáveis para:
    // -ter informações sobre os parâmetros
    // -guardar o tipo de retorno
    // -qualquer outra informação necessária
    // -etc

    //lista de comandos
    struct cmd *lista_de_comandos;

    // próxima função
    struct function *next;
};
```

Criação da AST: Bison

```
programa: declaracoes_globais lista_de_funcoes FIM_DE_ARQUIVO
;

lista_de_funcoes : funcao funcao_prime
;

funcao_prime: funcao funcao_prime
            |
;

funcao: tipo IDENTIFICADOR parametros '{' declaracoes_locais lista_de_comandos '}' { ... }
;

declaracoes_globais: ...
;
tipo: ...
;
parametros: ...
;
declaracoes_locais: ...
;
```

Criação da AST: Programa

```
programa: declaracoes_globais lista_de_funcoes FIM_DE_ARQUIVO {
    struct program* aux = (struct program*) malloc(sizeof(struct program));
    aux->globalSymbolTable = ... //cria e inicializa a tabela de símbolos global

    //as derivações e ações do não-terminal declaracoes_globais irá armazenar os símbolos globais na tabela de símbolos recém criada
    //além de verificar se existem identificadores duplicados, redefinidos, entre outros erros na declaração de símbolos.

    aux->lista_de_funcoes = $2; //pega o ponteiro da estrutura recebida pelo não-terminal lista_de_funcoes ($2),
                                //o qual contém a lista de todas as funções já completamente montada.
    aux->... //qualquer outra informação necessária é criada/inserida.
    $$ = aux; //retorna o ponteiro da estrutura do programa completamente montada para o não-terminal programa, o qual provavelmente
              //irá retornar esse ponteiro para alguma variável global que irá conter toda a estrutura do programa a qual será
              //utilizada na etapa de type checking.
}

;

lista_de_funcoes : funcao funcao_prime {
    $1->next = $2; //O não-terminal funcao ($1) possui um ponteiro para a estrutura de uma única função,
                  //se houver um outra função, haverá um ponteiro para a mesma no não-terminal funcao_prime ($2),
                  //dessa forma, basta fazer $1->next = $2; para criar uma lista ligada.
    $$ = $1; //retorna o ponteiro da lista de todas as funções para o não-terminal lista_de_funcoes
}

;

funcao_prime: funcao funcao_prime {
    $1->next = $2; //O não-terminal funcao ($1) possui um ponteiro para a estrutura de uma única função,
                  //se houver um outra função, haverá um ponteiro para a mesma no não-terminal funcao_prime ($2),
                  //dessa forma, basta fazer $1->next = $2; para criar uma lista ligada.
    $$ = $1; //retorna o ponteiro da lista de funções montada até agora para o não-terminal funcao_prime

}

| { $$ = (struct funcao*) NULL; /* retorna NULL pois não existem mais funções. */ }

;

struct program /* estrutura para programa */
{
    //tabela de símbolos global
    SYMBOL_TABLE_TYPE *globalSymbolTable;
    //outras estruturas...
    //lista de funções
    struct function *lista_de_funcoes;
};
```

Análise Semântica

type checking

Análise Semântica: Verificação de Tipos

O que é um tipo?

- A definição de tipo pode mudar de linguagem para linguagem.

Consenso:

- Um conjunto de valores
- Um conjunto de operações sobre estes valores

Classes são uma instanciação moderna da noção de tipo.

Análise Semântica: Verificação de Tipos

Por que os tipos são necessários?

Considere o seguinte fragmento de linguagem *assembly*:

```
add $r1, $r2, $r3
```

Quais os tipos de `$r1`, `$r2` e `$r3` ?

Análise Semântica: Verificação de Tipos

Certas operações são válidas para valores de alguns tipos

- Em C **não** faz sentido somar dois ponteiros para inteiros
- Em C faz sentido somar dois inteiros

Ambas as construções gerariam o mesmo código *assembly*:

```
add $r1, $r2, $r3
```

O sistema de tipos de uma linguagem serve para especificar quais operações são válidas para cada um dos tipos.

O objetivo da verificação de tipos é assegurar que as operações da linguagem estão sendo usadas com os tipos corretos.

Análise Semântica: Verificação de Tipos

As linguagens podem ser basicamente divididas em 3 tipos:

- Estaticamente tipadas: Toda ou quase toda a verificação de tipos é feita em tempo de compilação (C, Java, Pascal, etc)
- Dinamicamente tipadas: Toda a verificação de tipos é feita em tempo de execução do programa (basic)
- Não-tipadas: Não existe verificação de tipos (*assembly*)

Análise Semântica: Verificação de Tipos

Verificação de tipos

- O programador declara tipos para os identificadores
- O compilador infere o tipo para cada expressão

A verificação de tipos, verifica se para cada operação no programa se os tipos utilizados são válidos para a mesma.

No processo de inferência de tipo, o compilador preenche as informações que não estão presentes.

Análise Semântica: Verificação de Tipos

Verificação de tipos: Regras de Inferência

Se duas expressões E_1 e E_2 possuem um determinado tipo, então uma operação sobre elas irá gerar uma expressão E_3 com um determinado tipo.

Ex.:

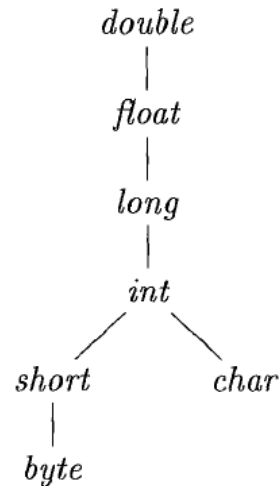
Se e_1 tem tipo `int` e e_2 tem tipo `int`,
Então $e_1 + e_2$ irá ter tipo `int`.

Se e_1 tem tipo `int*` e e_2 tem tipo `int`,
Então $e_1 + e_2$ irá ter tipo `int*`.

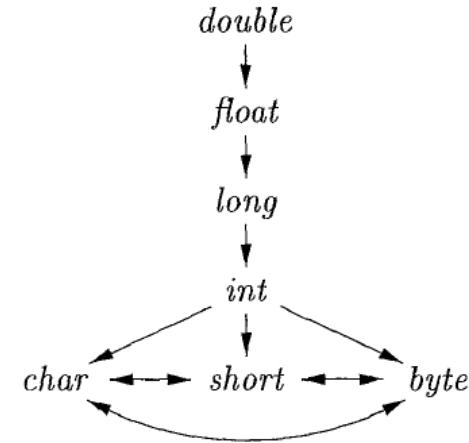
Análise Semântica: Verificação de Tipos

Verificação de tipos: Regras de Inferência

Se duas expressões E_1 e E_2 possuem tipos diferentes, pode ser necessário converter um deles, para verificar a validade da operação entre as expressões. Ex.:



(a) Widening conversions



(b) Narrowing conversions

Análise Semântica: Verificação de Tipos

Verificação de tipos

A verificação de tipos é feita realizando-se um percurso em *pós-ordem* na *AST*, ou seja, das folhas até a raiz, onde os tipos são passados dos nós filhos para os nós pais, onde se verifica a compatibilidade deles em relação a operação sendo realizada.

Análise Semântica: Verificação de Tipos

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

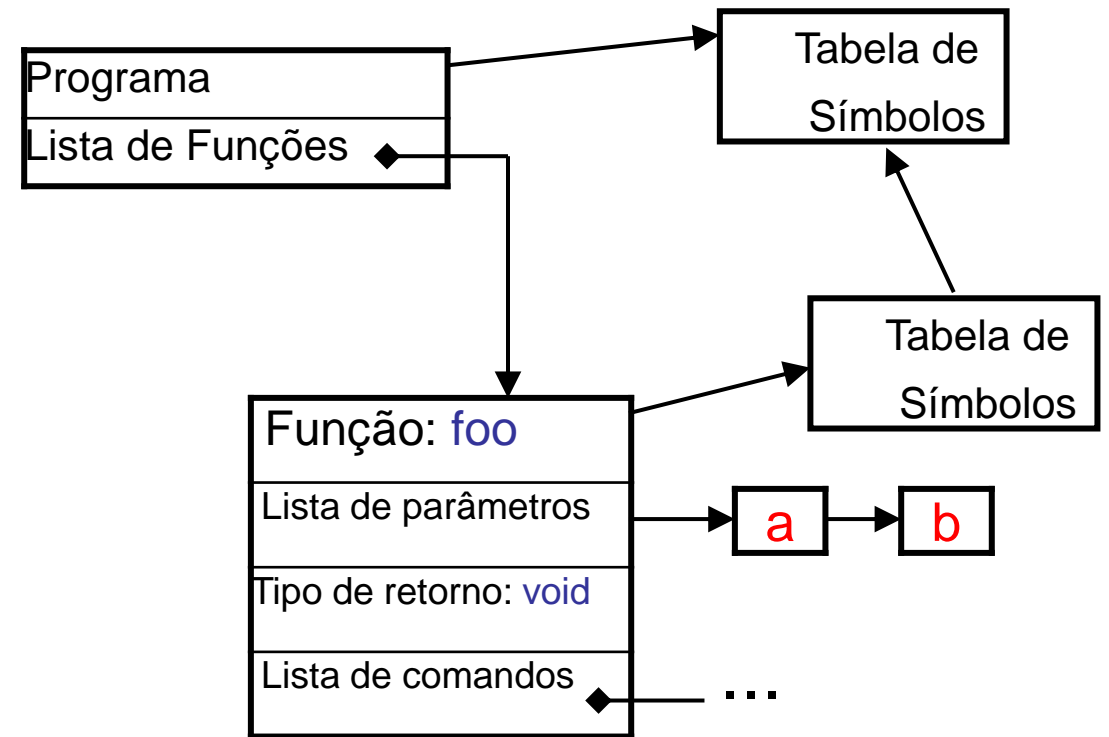
    i = a + b + i;
    f = i + a + c;
    i = p + (f + a);
}
```

Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    i = p + (f + a);
}
```

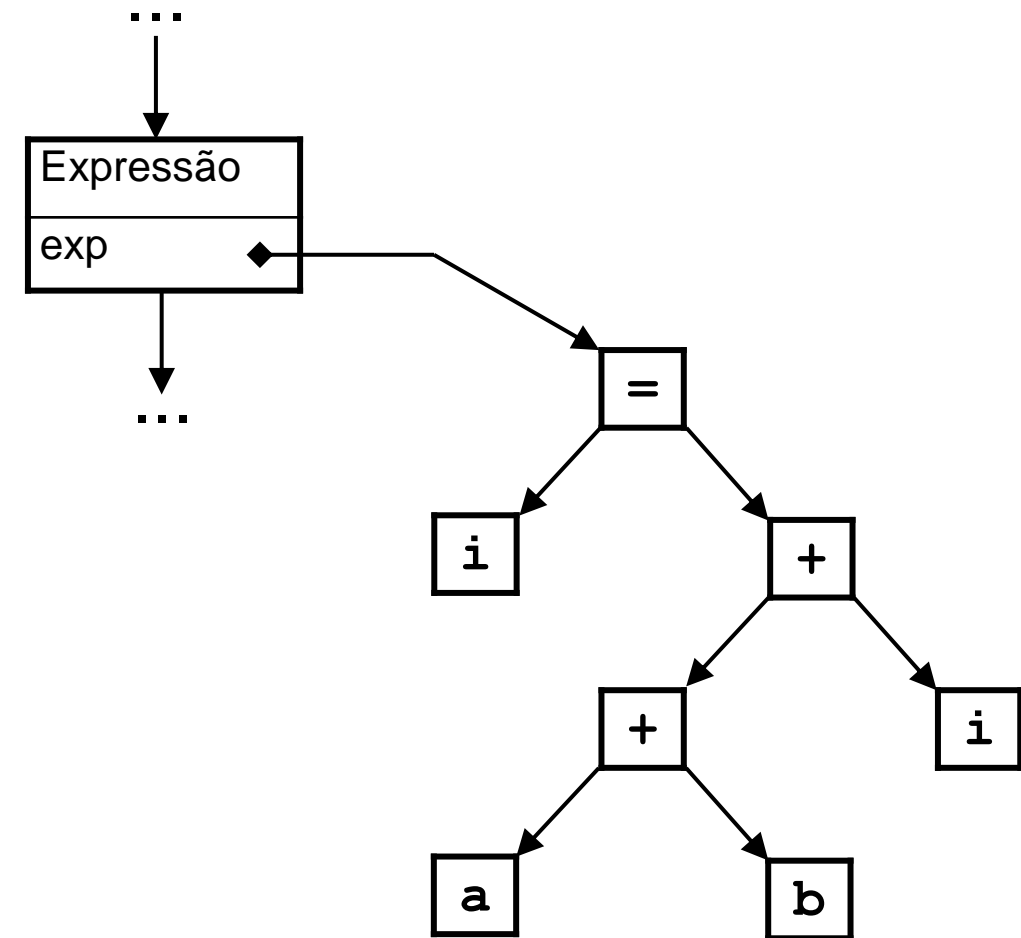


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

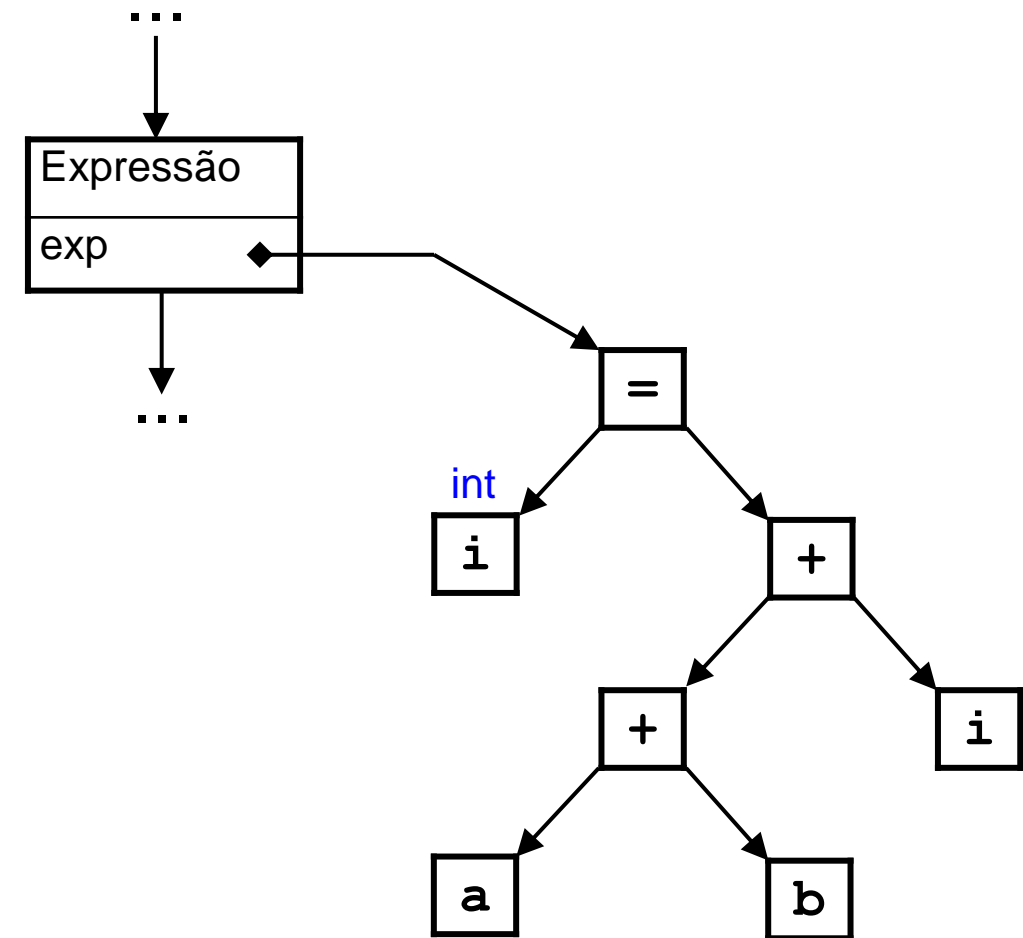


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

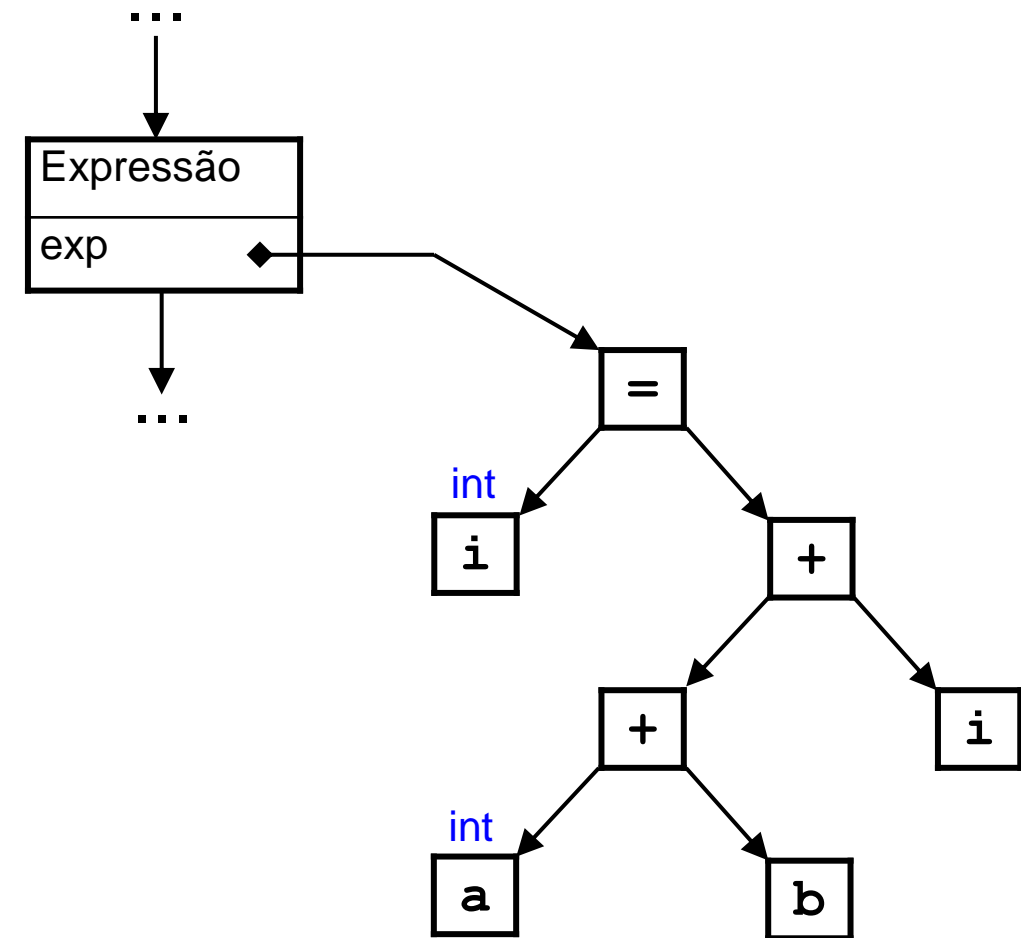


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

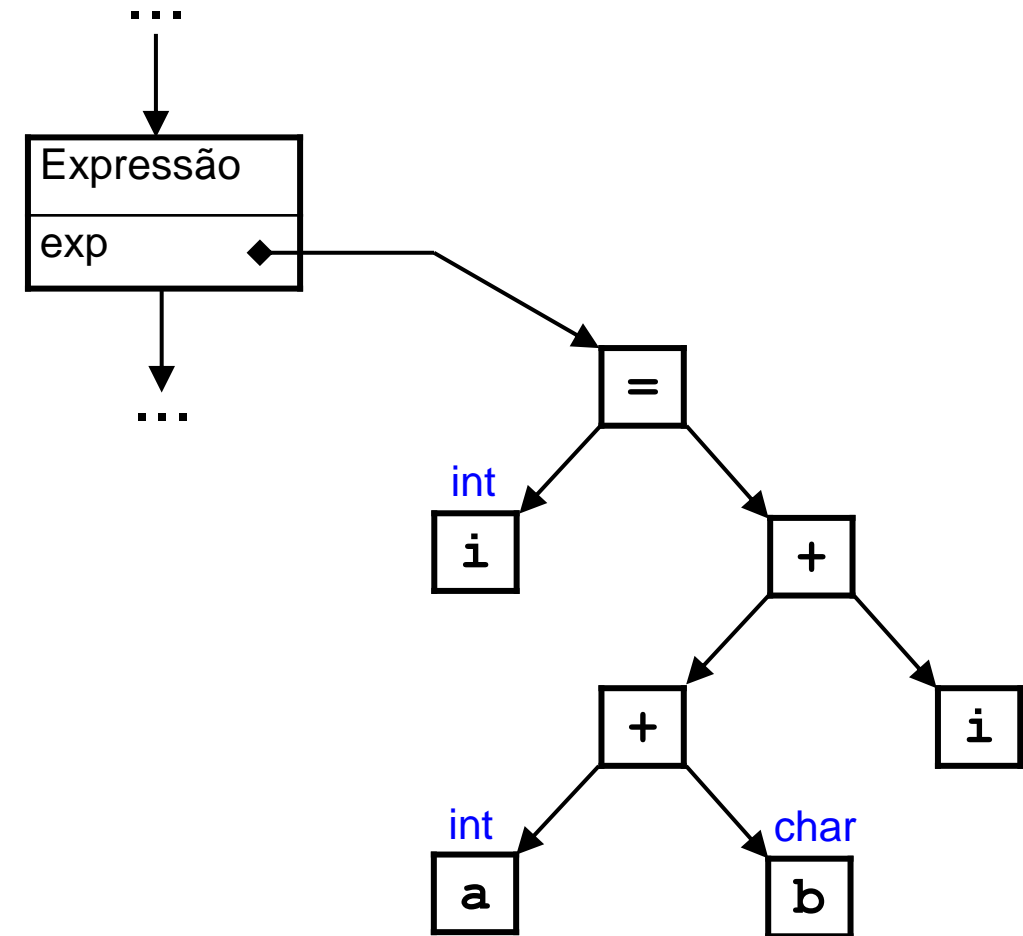


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

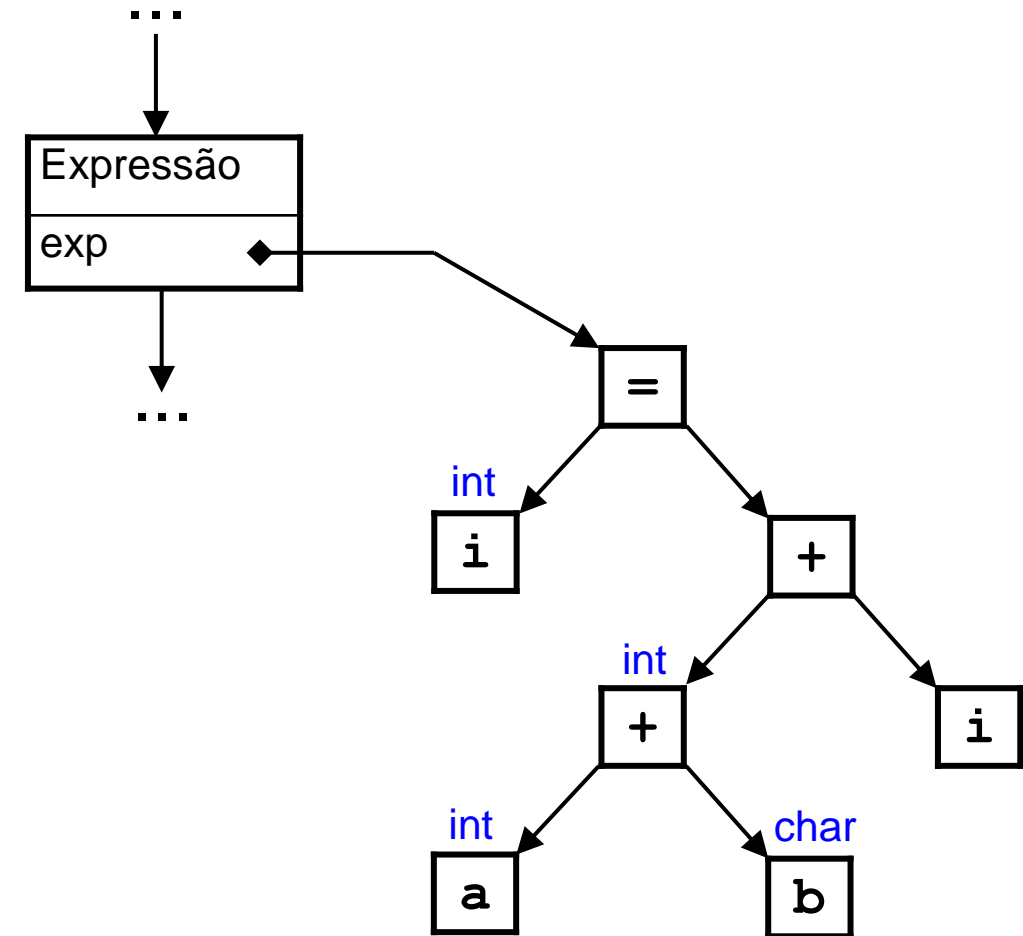


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

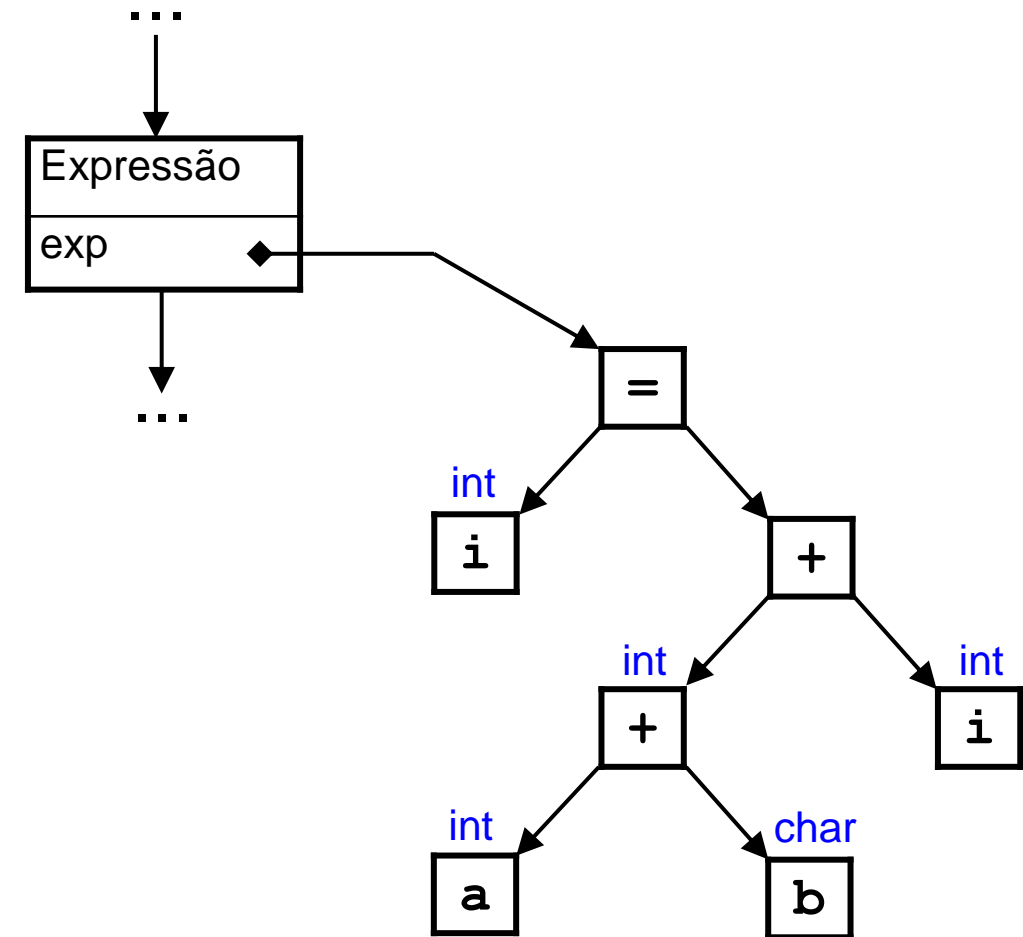


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

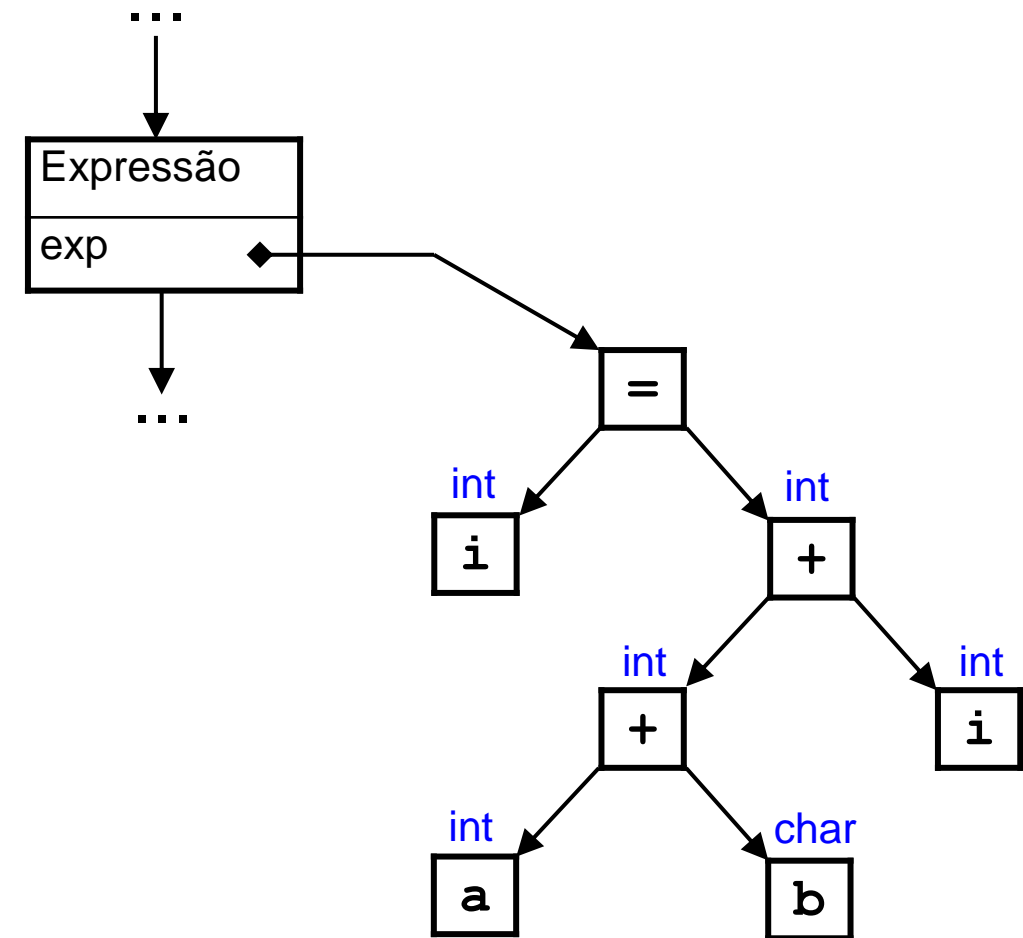


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

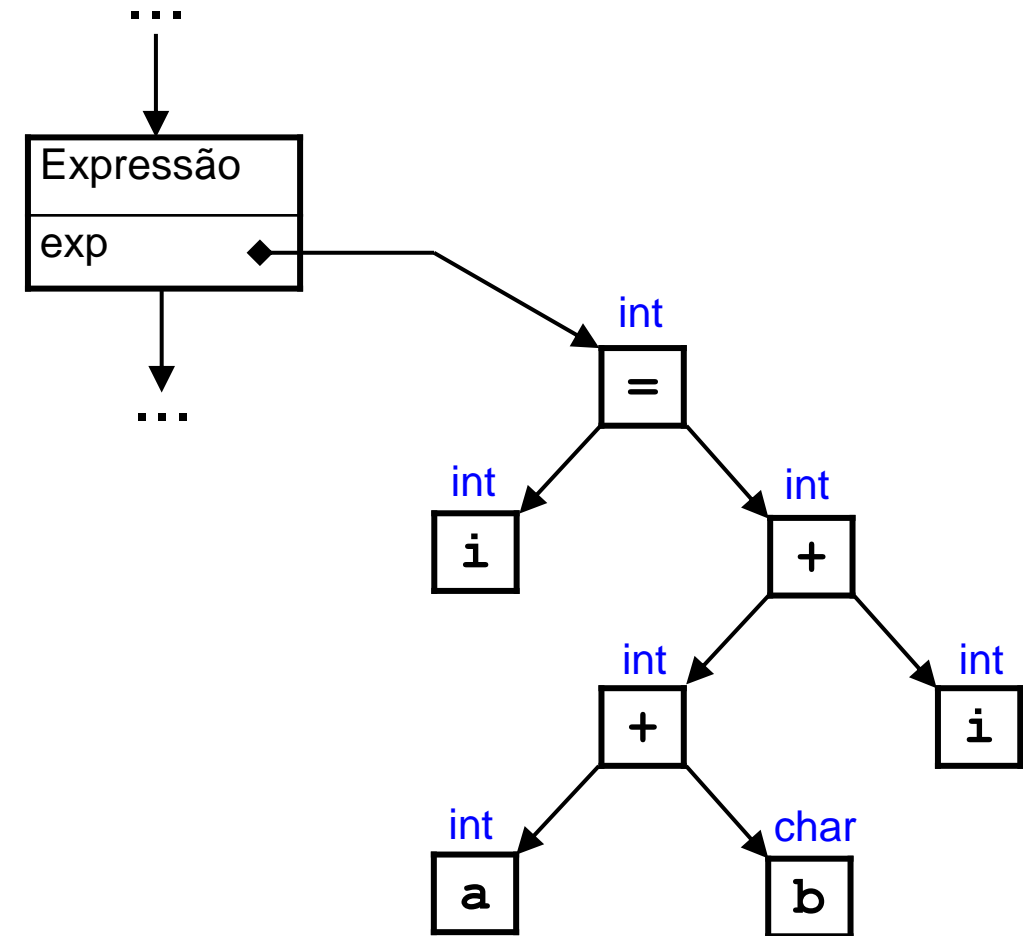


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

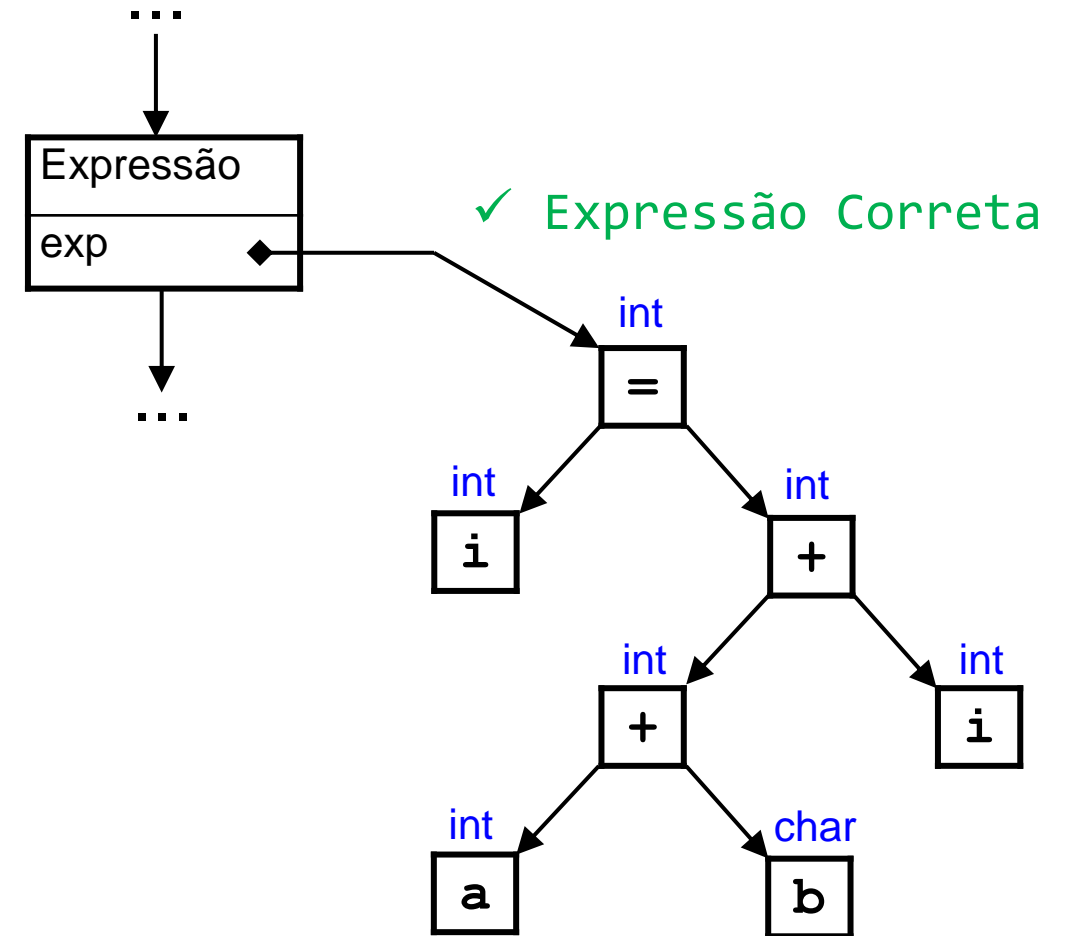


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    → i = a + b + i;
      f = i + a + c;
      i = p + (f + a);
}
```

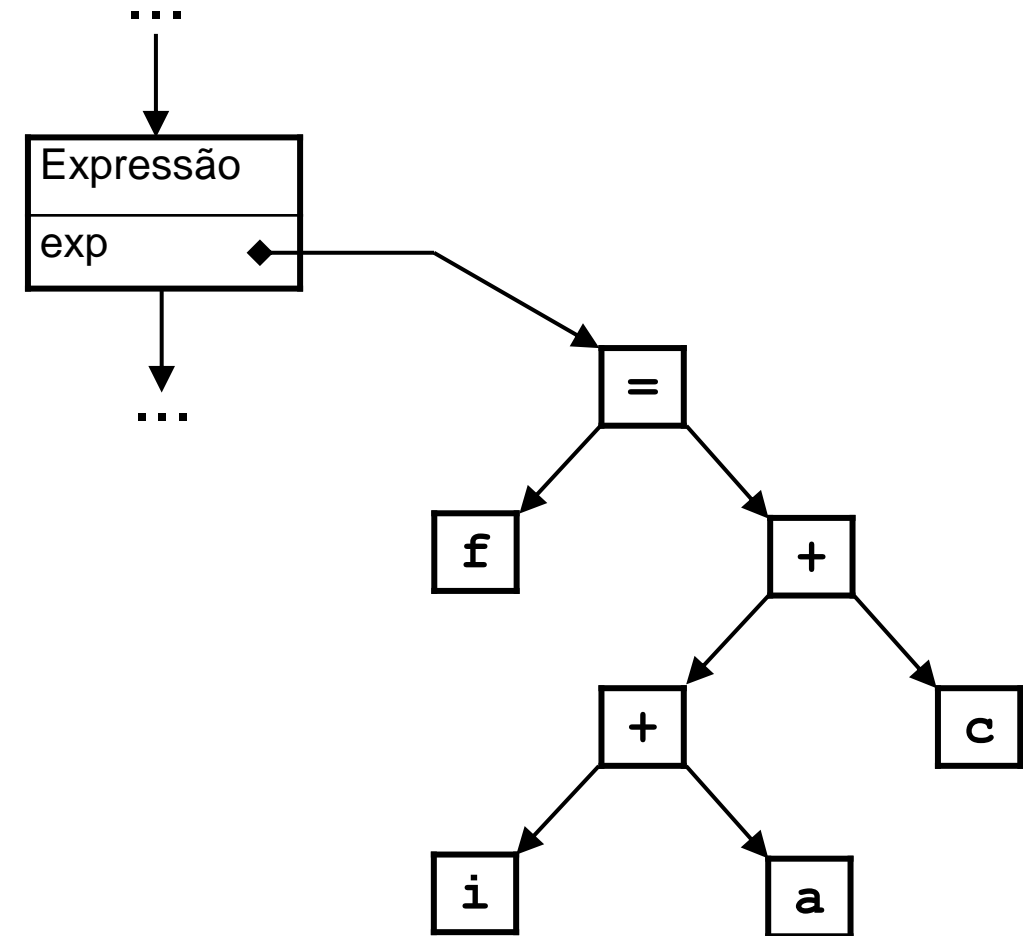


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    → f = i + a + c;
    i = p + (f + a);
}
```

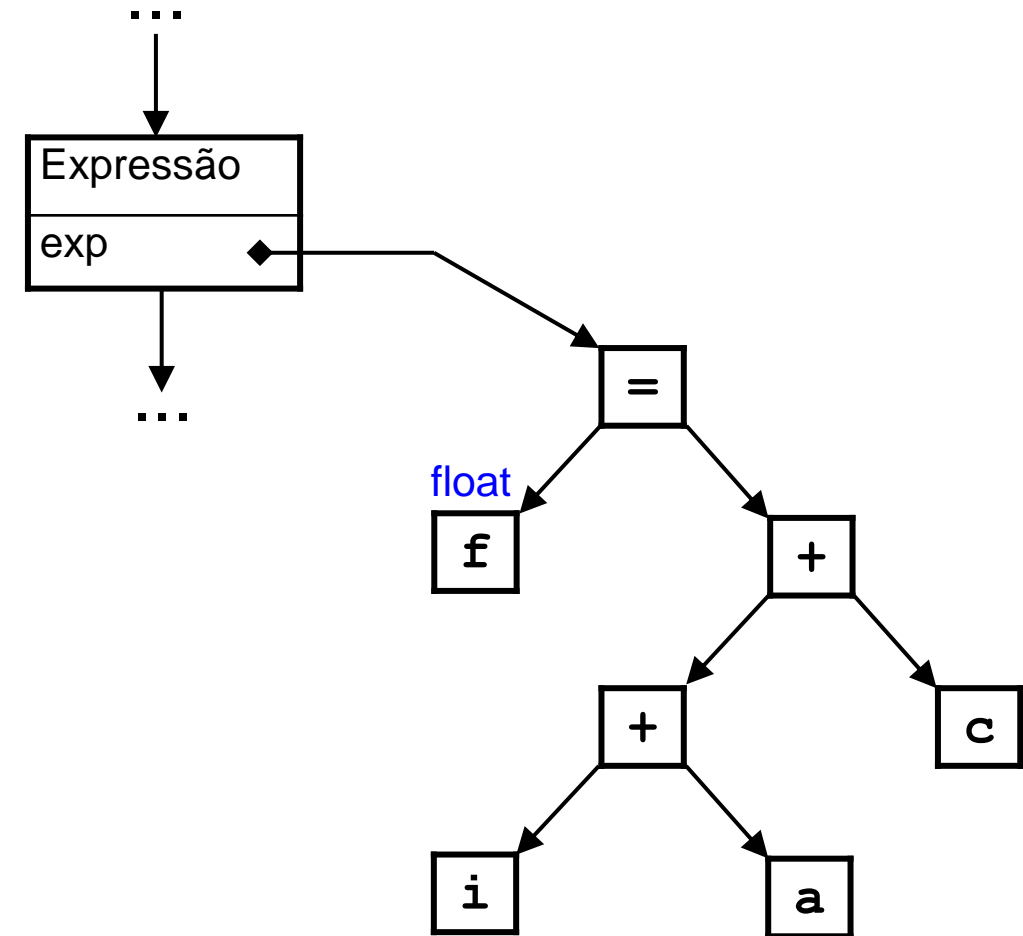


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    → f = i + a + c;
    i = p + (f + a);
}
```

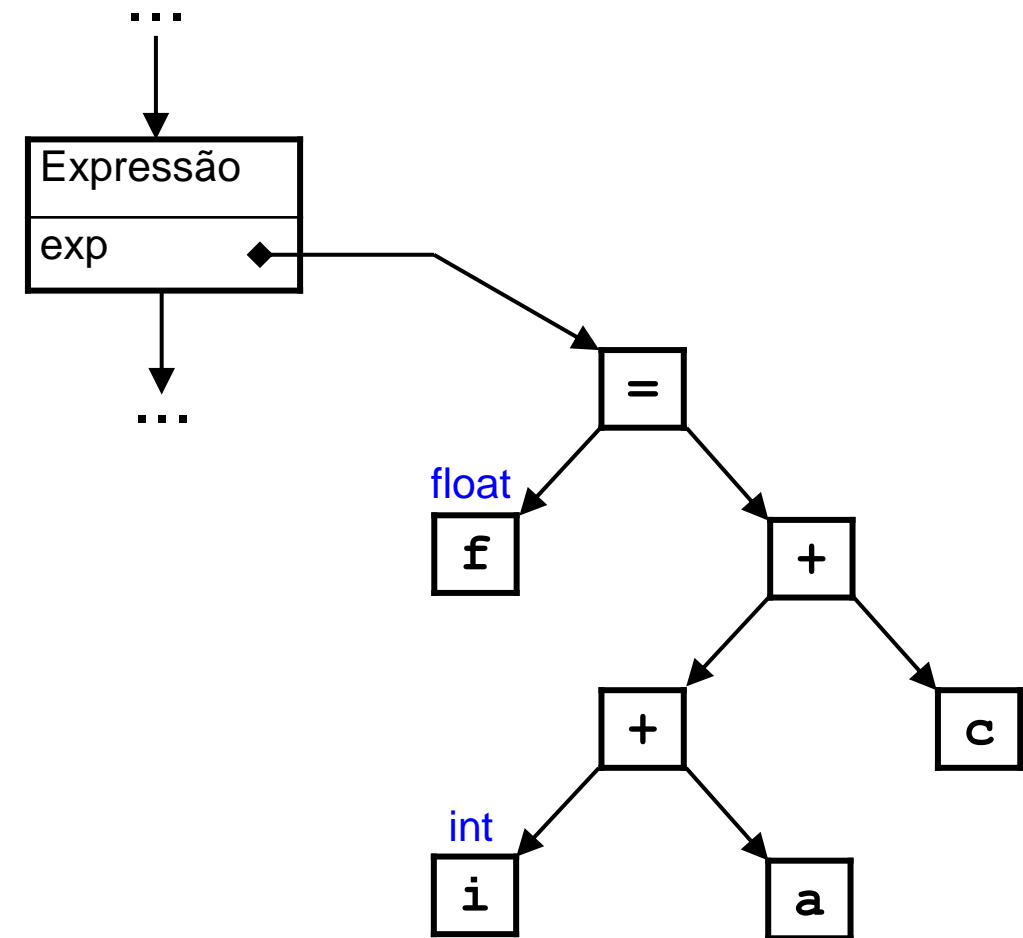


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    → f = i + a + c;
    i = p + (f + a);
}
```

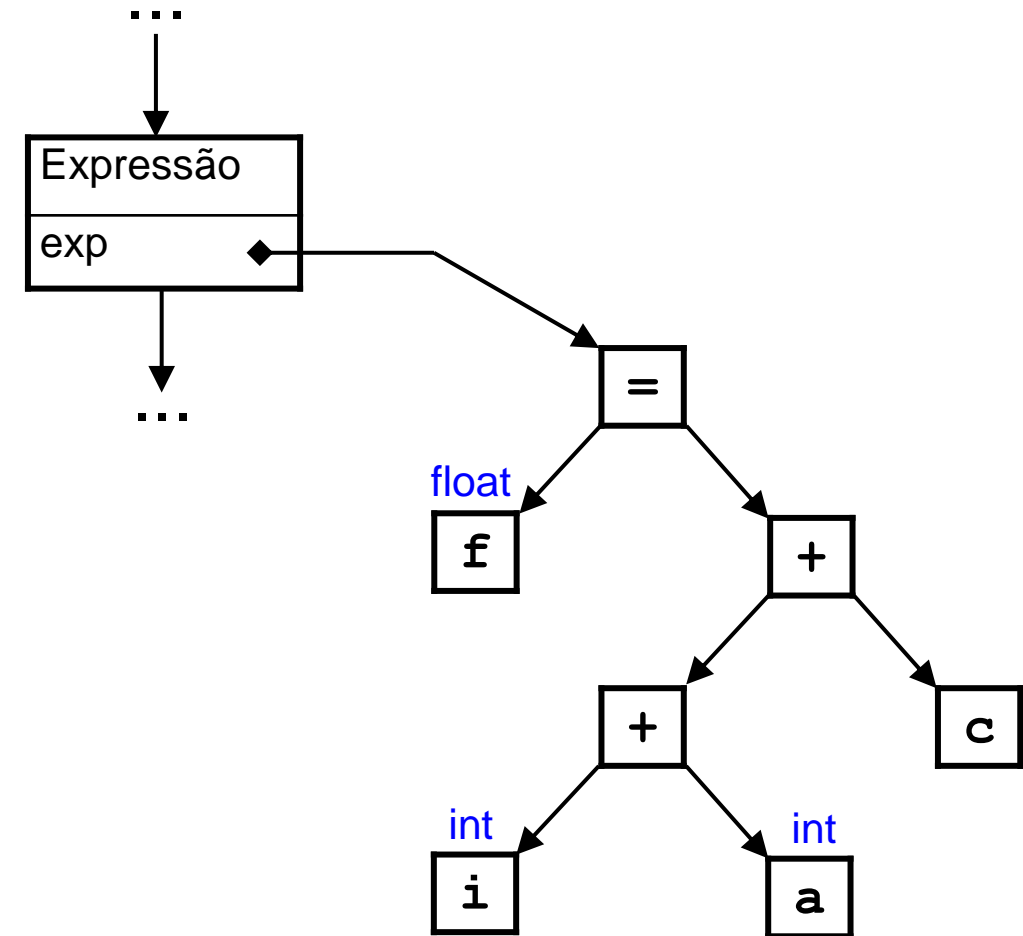


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    → f = i + a + c;
    i = p + (f + a);
}
```

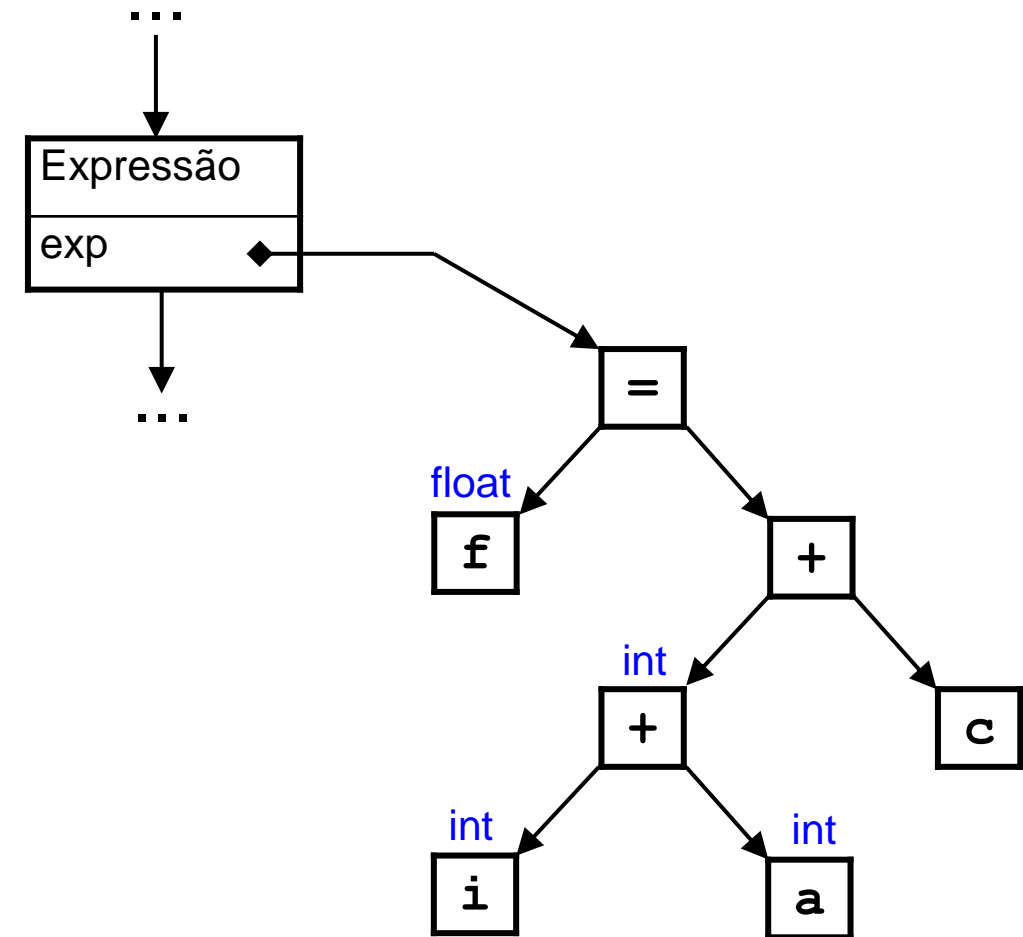


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    → f = i + a + c;
    i = p + (f + a);
}
```

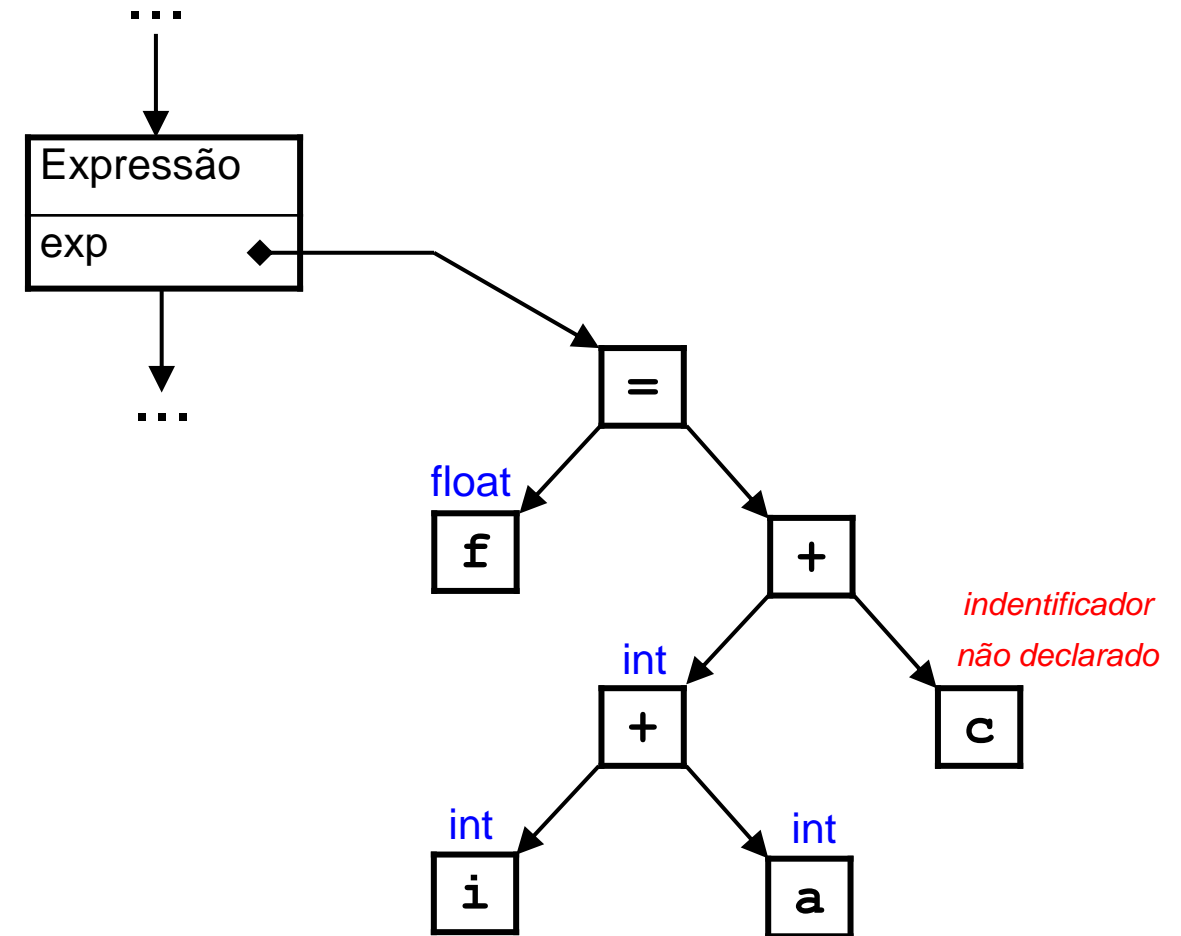


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    → f = i + a + c;
    i = p + (f + a);
}
```

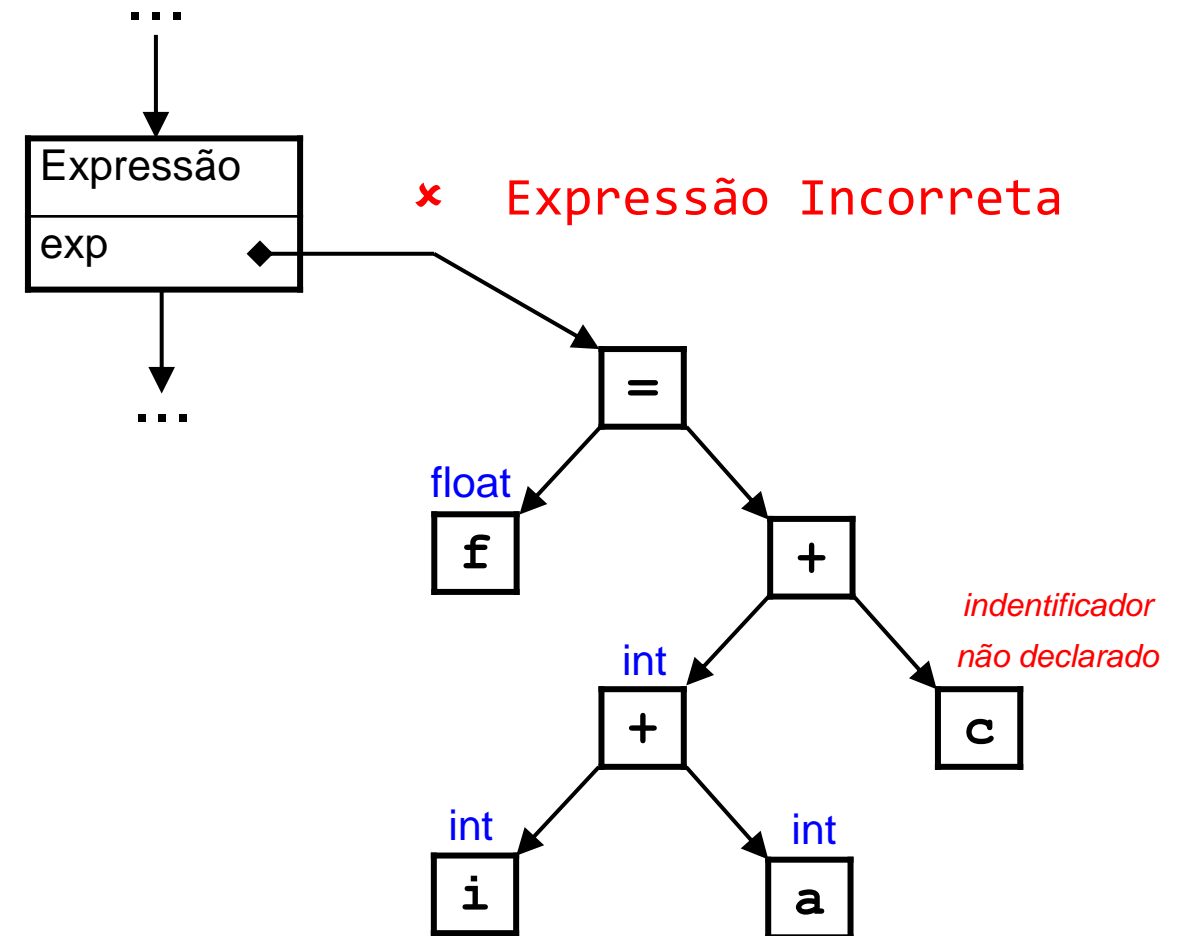


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    → f = i + a + c;
    i = p + (f + a);
}
```

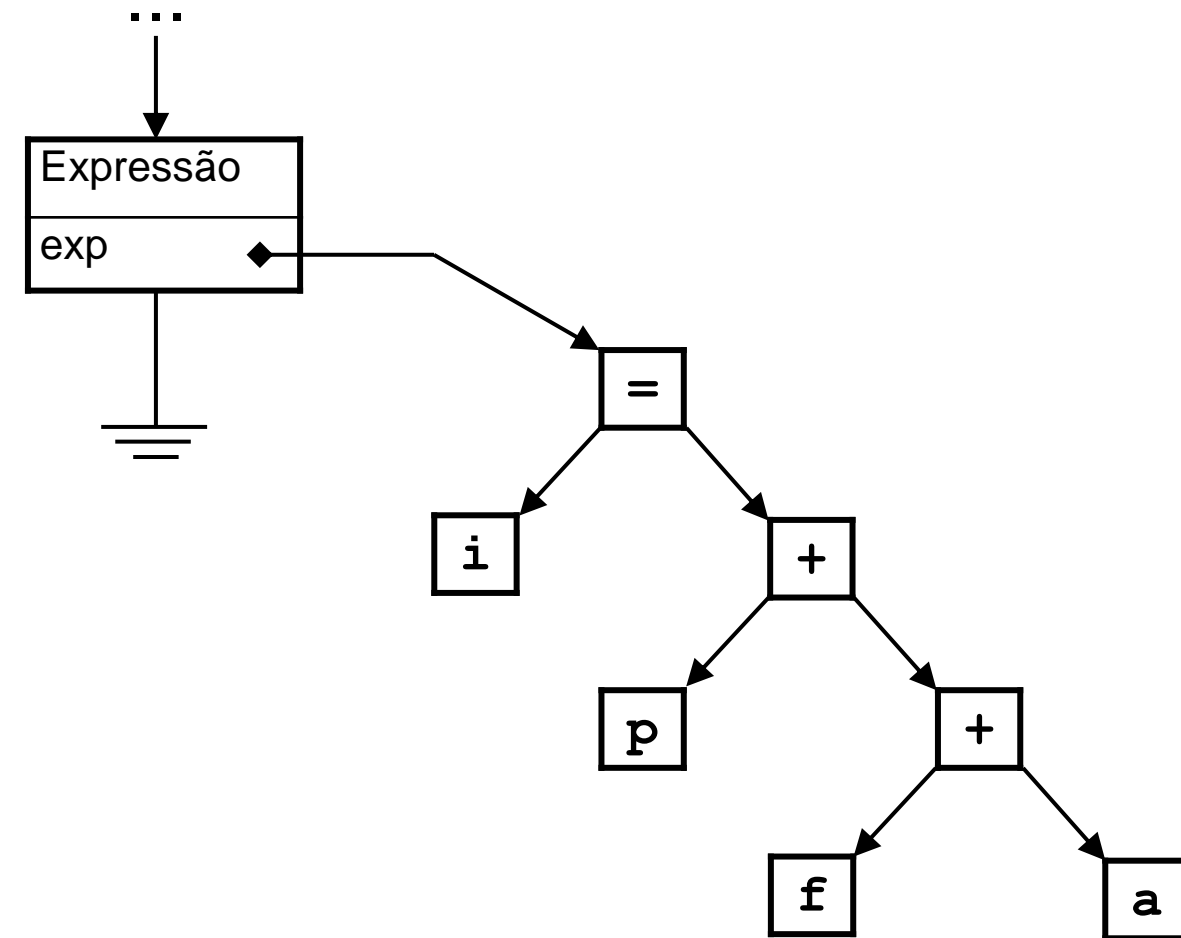


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```

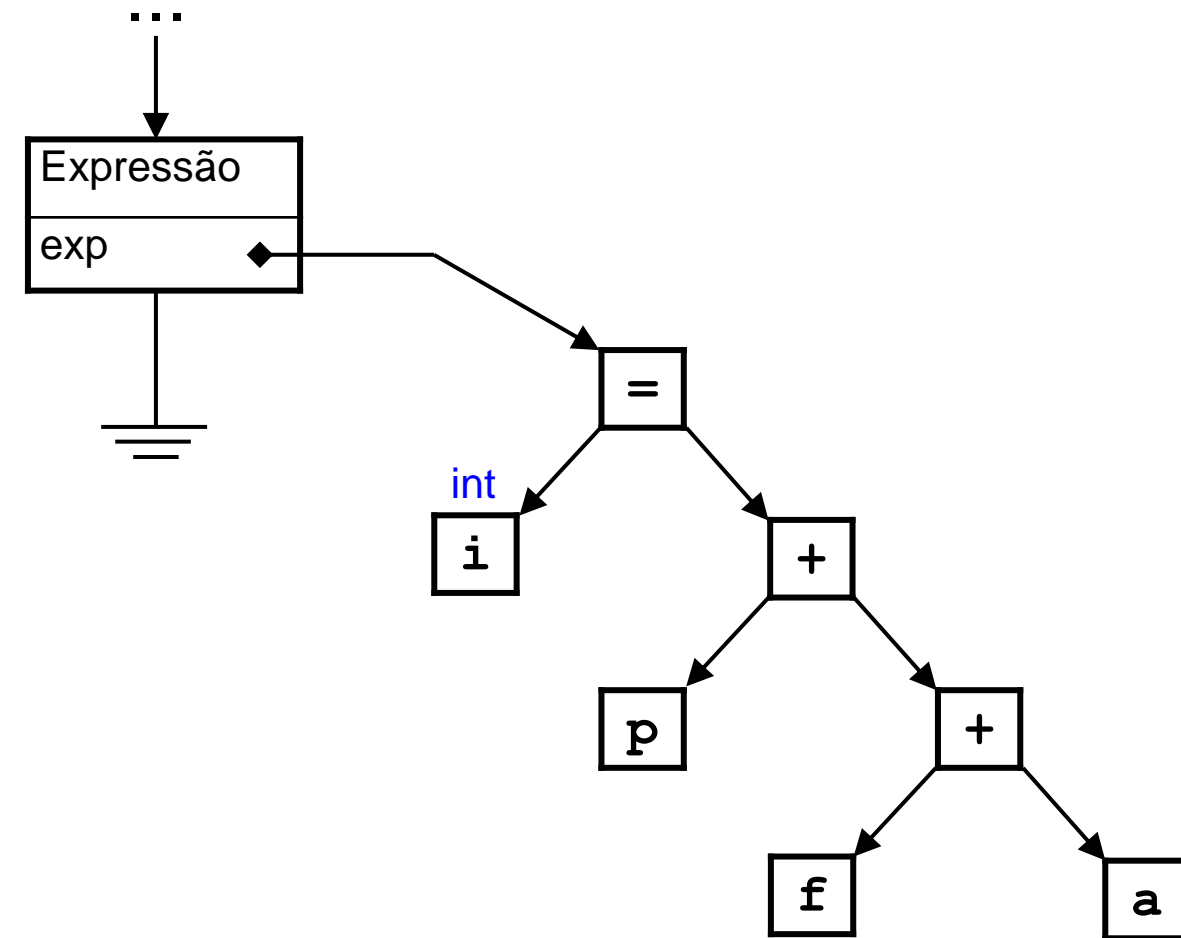


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```

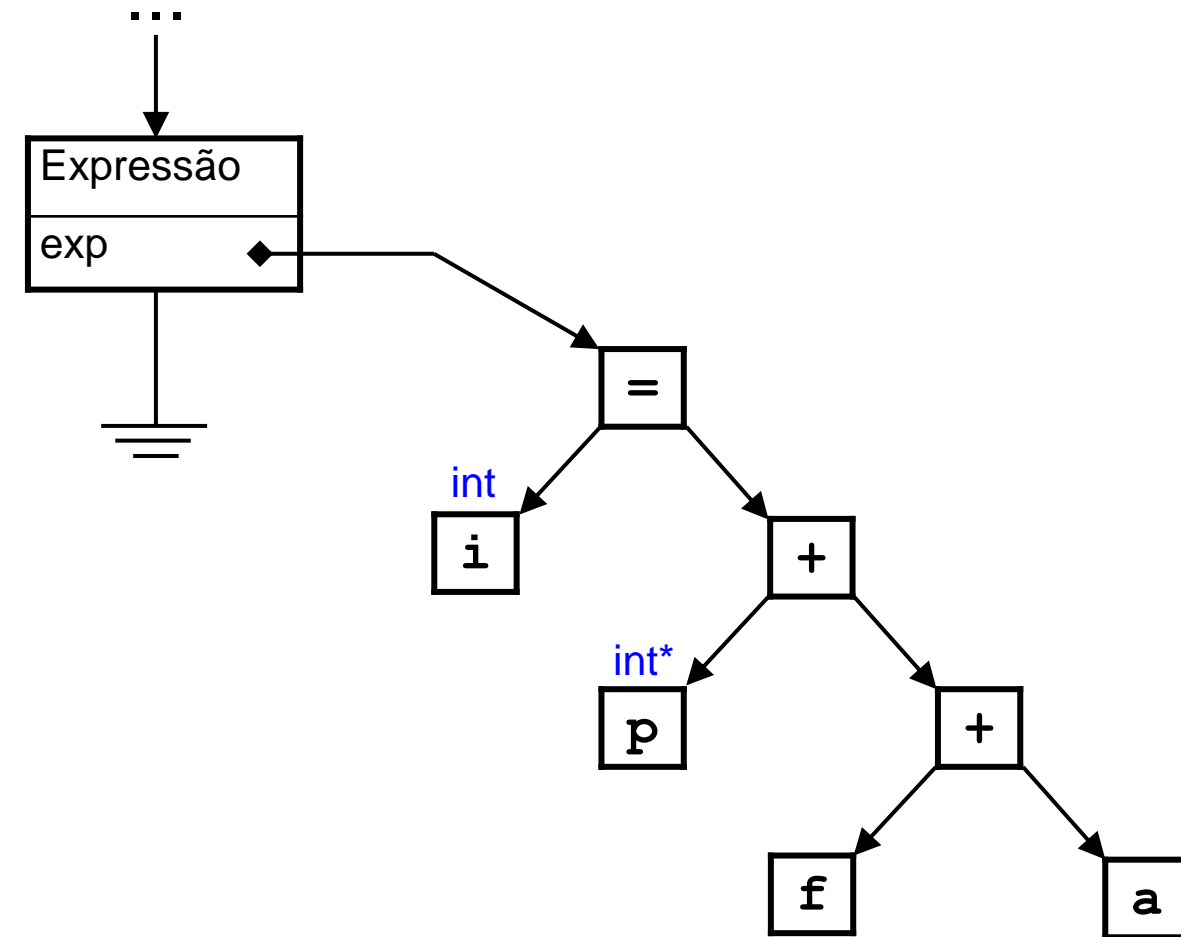


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```

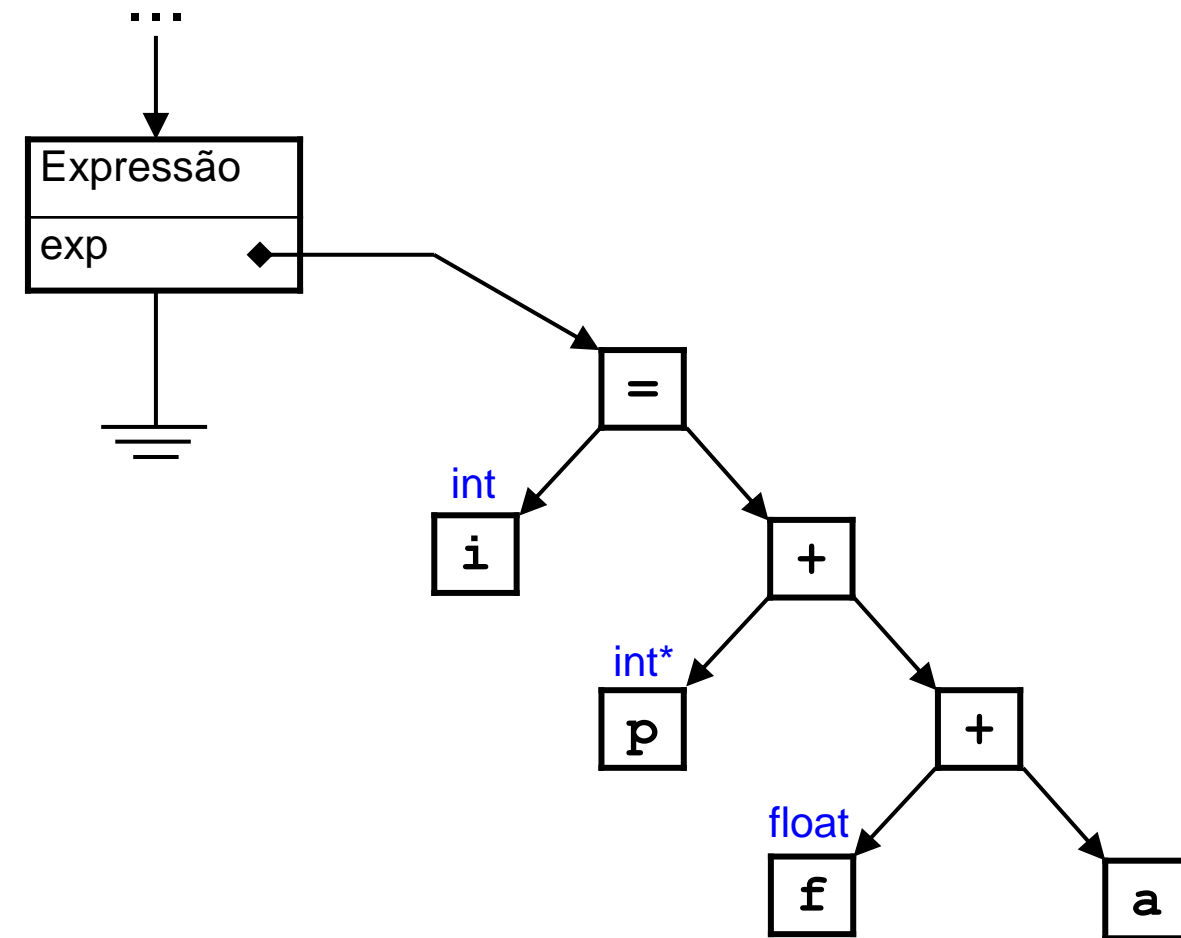


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```

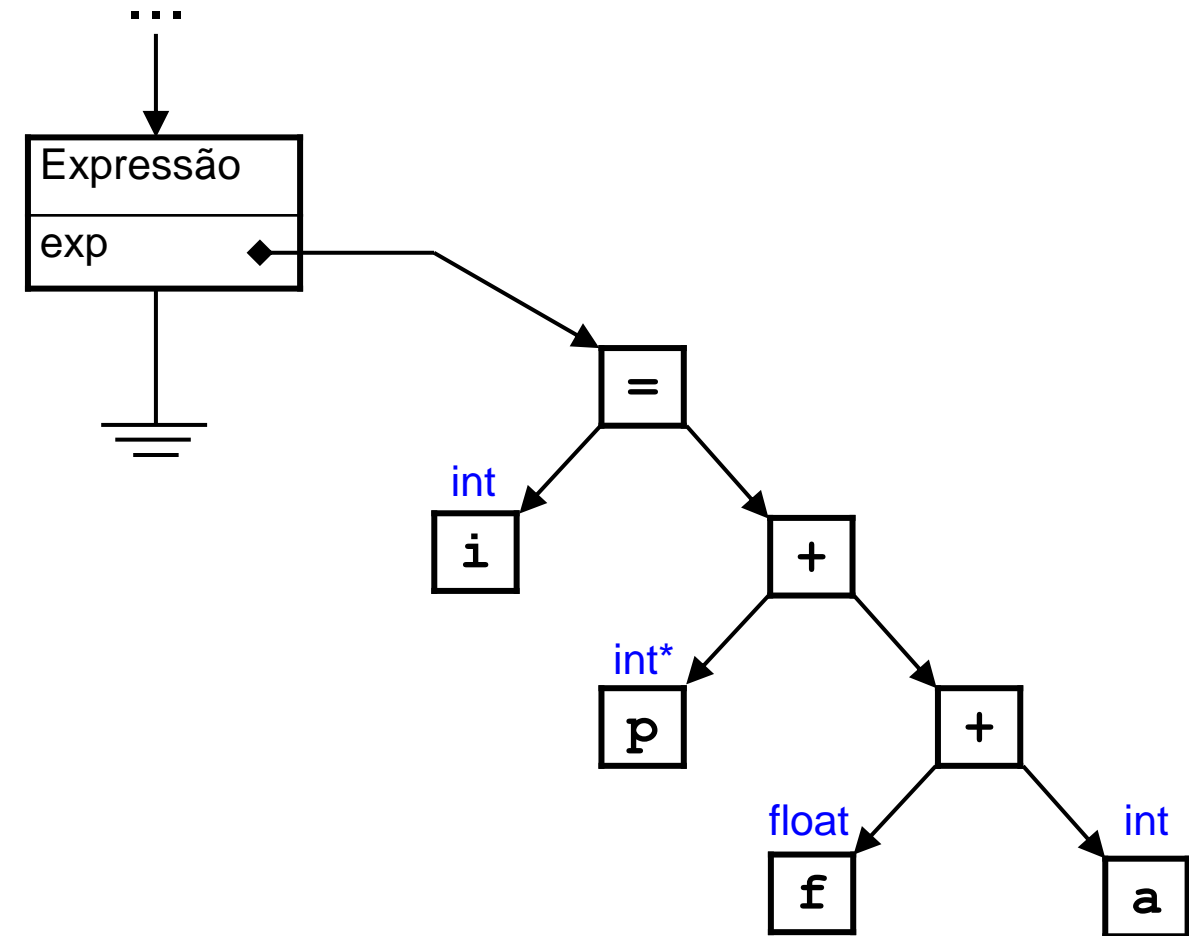


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```

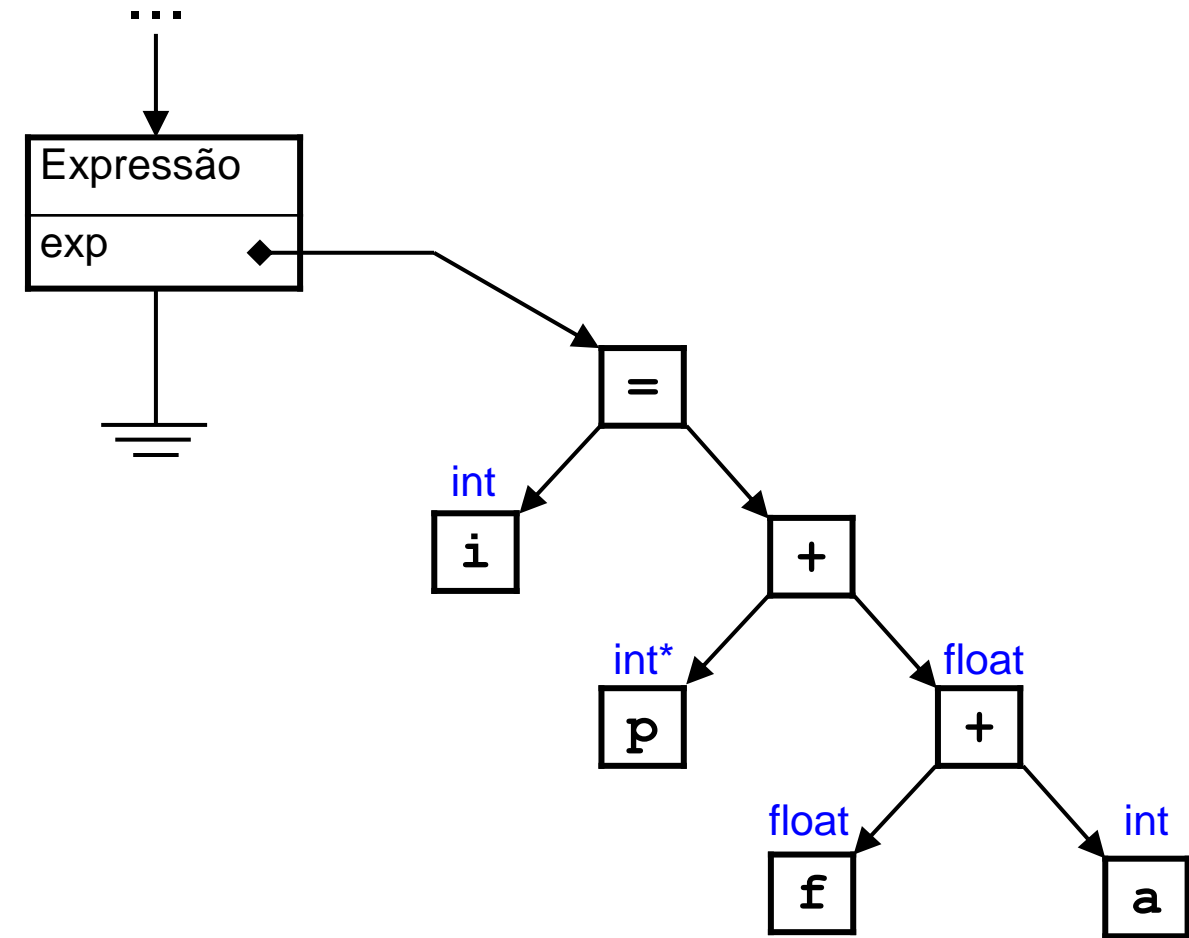


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```

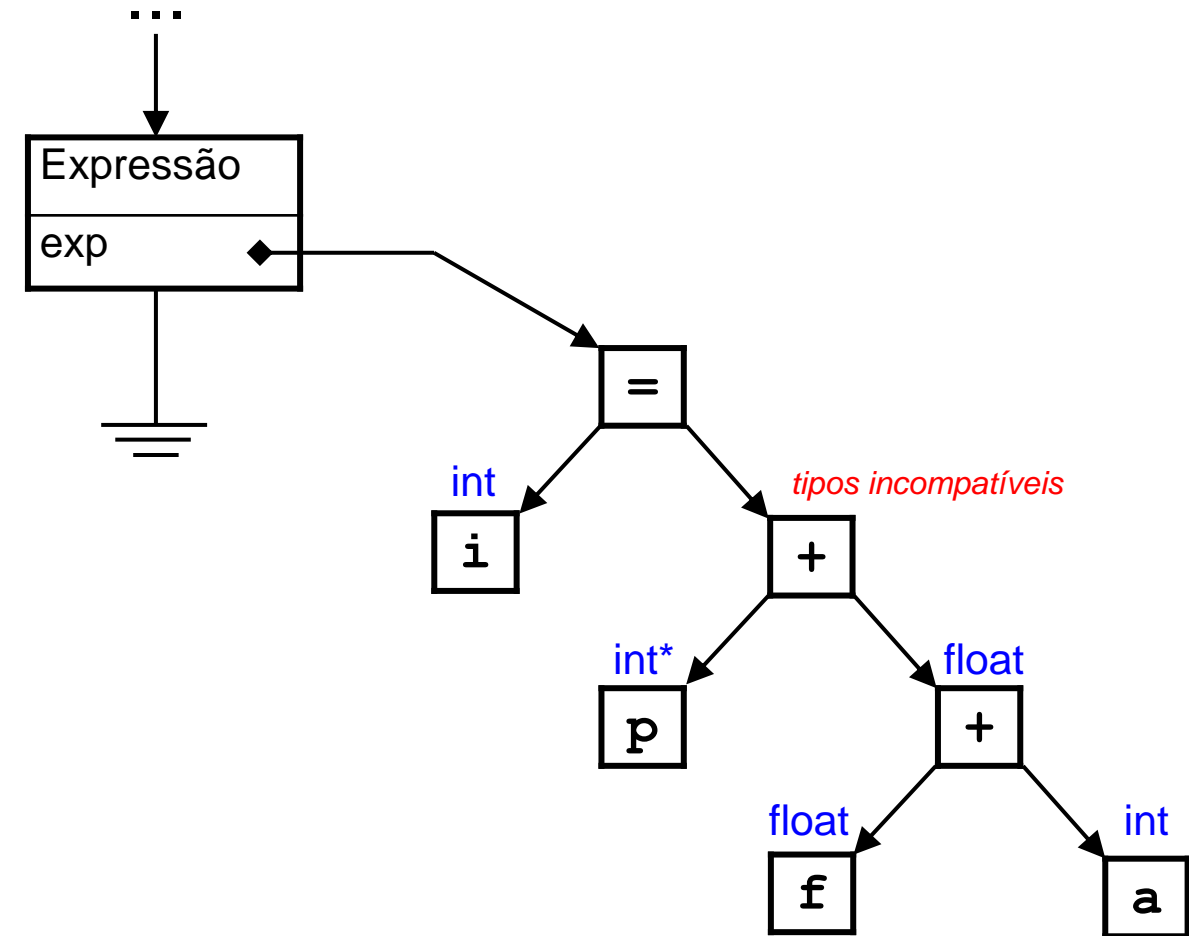


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```

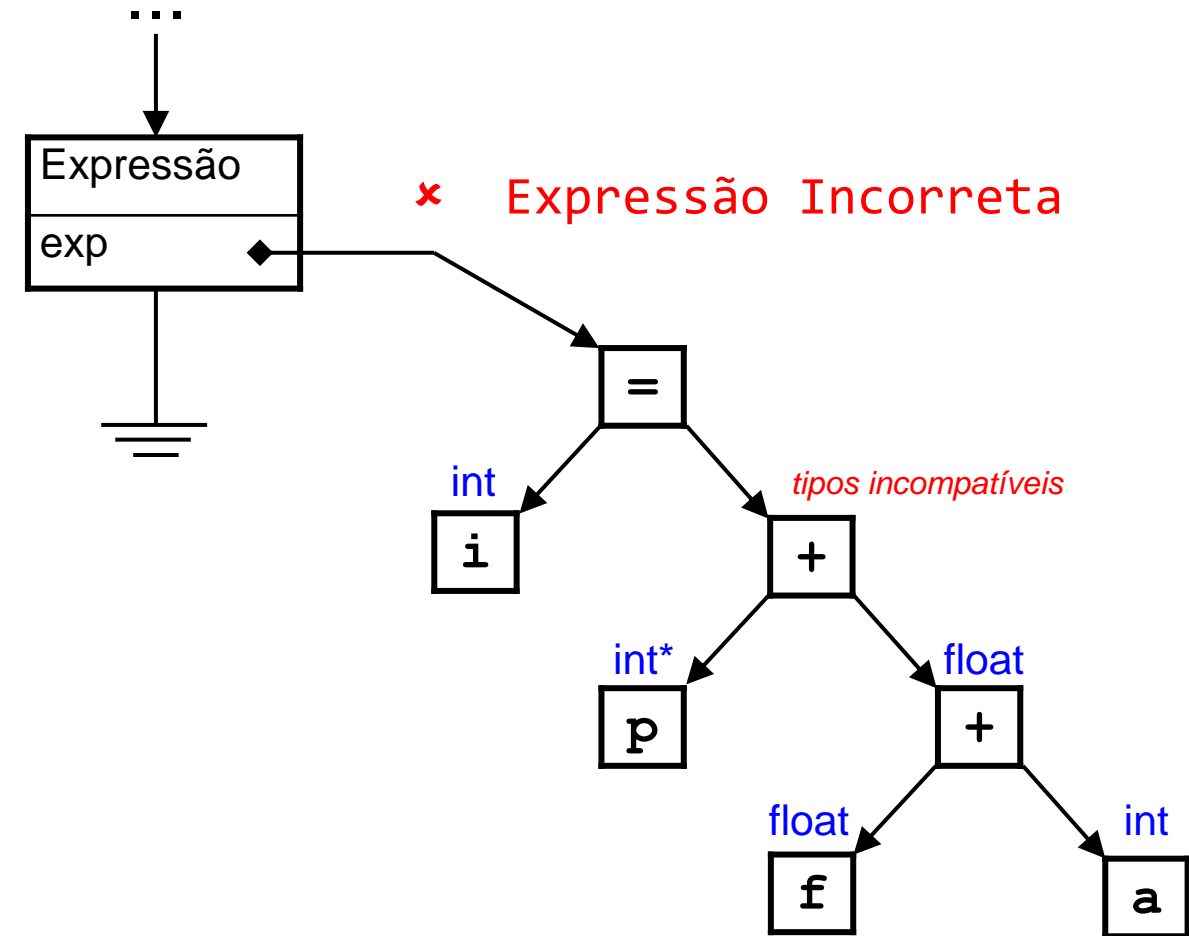


Análise Semântica: Criação da AST

Como seria a verificação de tipos para:

```
void foo(int a, char b)
{
    int i, *p;
    float f;

    i = a + b + i;
    f = i + a + c;
    → i = p + (f + a);
}
```



Conteúdo Programático e Cronograma

1º Bimestre:

- ~~Organização e estrutura de compiladores~~
- ~~Análise Léxica~~
- ~~Análise Sintática~~
- ~~Ferramentas de geração automática de compiladores~~

2º Bimestre:

- ~~Análise Semântica~~

Conteúdo Concluído!!!



Lista de Exercícios

Lista 15

- Exercício Prático.

Compilador C

Próxima etapa do Compilador

- Compilador C – Analisador Semântico

