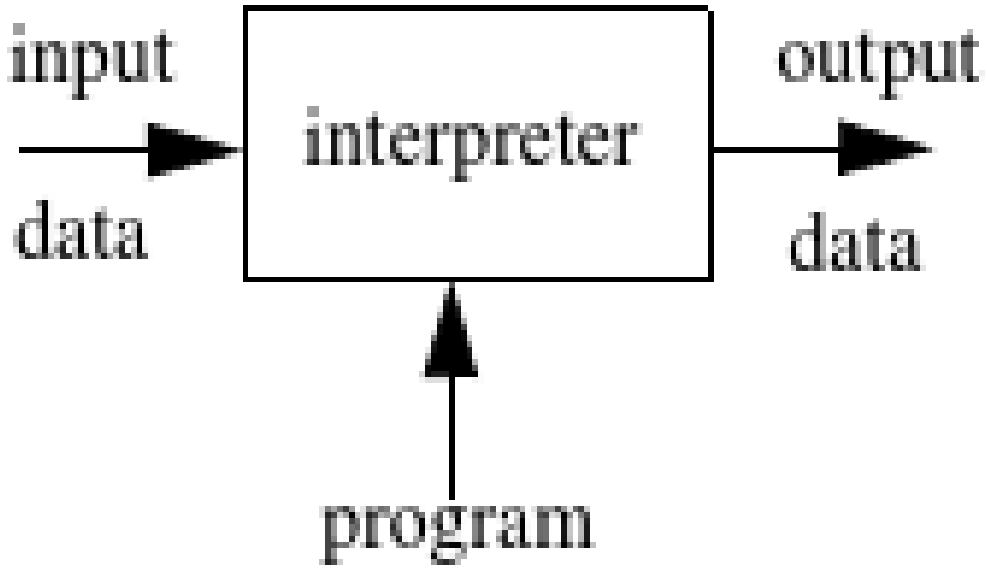
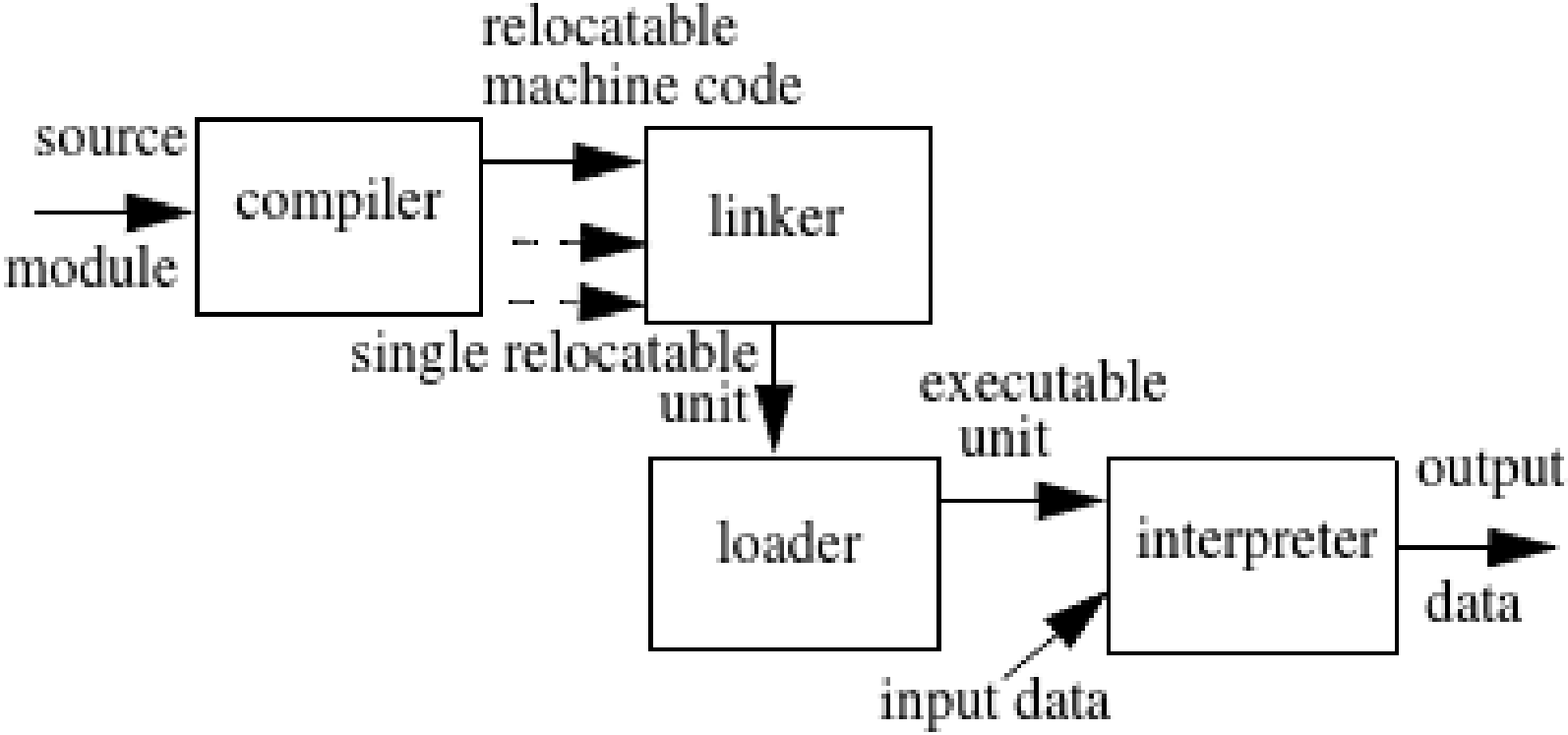
Tipos de Linguagem:

- Interpretadas: Python, Tcl, Lisp, Haskel...
- Compiladas: C, C++, FORTRAN, Java (?)..



(a) Interpretation



(b) Translation (+ interpretation)

```
for (i = 0; i < n; i++)
    soma += valor[i];
```

```
...
LOOP: BGE R1, R2, EXIT
LD R3, V(R1)
ADD R4, R3, R4
ADDI R1, R1, #4
J LOOP
EXIT:
```

- Sintaxe: o conjunto de regras que forma a linguagem.
- Semântica: Define o significado de programas semânticamente corretos.

Duas construções que diferem apenas no nível léxico seguem a mesma sintaxe abstrata, mas diferem na sintaxe concreta:
Ex:

```
while (x != y) {
..
}
```

```
while x <> y do
begin
..
end
```

Semântica se divide em:

- Semântica Estática: verificável em tempo de compilação.
- Semântica Dinâmica: somente verificável em tempo de execução.

Regras Sintáticas

<operator>::= + | - | * | / | = | ¦ | < | > | ≤ | ≥
<identifier>::= <letter> <ld>*
<ld>::= <letter> | <digit>
<number>::= <digit>+
<letter>::= **a** | **b** | **c** | . . . | **z**

Regras Léxicas

<program>::={ <statement>* }
<statement>::=<assignment> | <conditional> | <loop>
<assignment>::=<identifier> = <expr> ;
<conditional>::=**if** <expr> { <statement> + } |
if <expr> { <statement> + } **else** { <statement> + }
<loop>::=**while** <expr> { <statement> + }
<expr> ::=<identifier> | <number> | (<expr>) | <expr> <operator> <expr>

Amarração

- Em tempo de definição da linguagem.
- Em tempo de implementação da linguagem.
- Em tempo de compilação.
- Em tempo de execução.

Tipos de dados

- Tipos básicos primitivos: **char, int, float, double**.
- Modificadores dos tipos: **unsigned, signed, short, lont**.
- Arrays ou vetores.

Nesta aula:

- Tipos definidos pelo usuário.
- Conversão de tipos.
- Checagem de tipos.

Estruturas (ou registros)

- Permitem que o programador defina tipos que armazenam diversos itens de diferentes tipos.
- Suponha que você deseje armazenar informações dos livros em uma coleção, cada livro tendo os seguintes atributos: Título, autor e número de catalogação.

```
struct Livro {
   char titulo[50];
   char autor[50];
   int id;
};

struct Livro livro[10];
```

- Instanciando:
  - `struct Livro meulivro;`
  - `struct Livro livros[10];`
  - `typedef struct Livro Livro_t;`
- Acessando variáveis da estrutura:
  - `meulivro.id = 10;`
  - `printf("Titulo:  %s", meulivro.titulo);`

```
struct Ponto {
        int x; int y;

};
void main(void) {
        struct Ponto p0 = {1, 1}, p1;

        p1.x = -1;
        p1.y = 1;

        if (p0 == p1)
                printf("p0 e p1 sao o mesmo ponto\n");

}
```

```
struct Ponto {
        int x; int y;

};
void main(void) {
        struct Ponto p0 = {1, 1}, p1;

        p1.x = -1;
        p1.y = 1;

        if (p0 == p1) /* NAO FUNCIONA! */
                printf("p0 e p1 sao o mesmo ponto\n");

        if (p0.x == p1.x && p0.y == p1.y) /* CORRETO! */
                printf("p0 e p1 sao o mesmo ponto\n");
}
```

- Em Pascal, as estruturas (chamadas records nessa linguagem) podem ter campos diferentes, de acordo com o valor de uma variável.
- Essas estruturas são chamadas de variant records.

```
type Pagamento is (Dinheiro, Cheque, Cartao);

type Transacao(Tipo: Pagamento := Dinheiro) is

    Total: Integer;

    case Tipo is
        when Dinheiro =>
            Desconto: boolean;
        when Cheque =>
            NumeroCheque: Positive;
        when Cartao =>
            NumberoCartao: String(1..5);
            Validade: String(1..5);
    end case;
end record;
```

Uniões (*Unions*)

- Tipos de dados criados pelo usuário que são manipulados de maneira semelhante às estruturas.
- Ao contrário das structs, nas unions os diferentes campos ocupam o mesmo espaço.
- Unions são úteis em situações nas quais dados de diferentes tipos/tamanhos são manipulados por uma função. Por exemplo, em um programa cliente/servidor.

```
union Dados {
int i;
float f;
char str[30];
};
...
Union Dados meusdados; //TAMNHO: 30 bytes
dados.i = 10;
```

Exercício: Escreva um programa em C no qual duas estruturas representam pontos no plano e no espaço, respectivamente. Desejamos então armazenar em um vetor pontos dos dois tipos e poder decidir corretamente que tipo de operação fazer sobre cada ponto (considere que há funções diferentes para manipular pontos no plano e no espaço).

```
struct PontoPlano {
        int x;
        int y;
};
struct PontoEspaco {
        int x;
        int y;
        int z;
};
struct Ponto {
        char EstaNoPlano;
        union {
                struct PontoPlano r2;
                struct PontoEspaco r3;
        } valor;
};
```

```
int main(void) {
struct Ponto p[10];
        ...

if (p[10].EstaNoPlano)
   operacao_2d(p[10]);
        else
   operacao_3d(p[10]);

}
```

Enumerações (enum)

- Tipo de dados definidos pelo usuário composto de constantes inteiras.
- Útil para tornar um programa mais legível.
- enum booleano { falso; verdadeiro; };
- enum booleano var;
- var = falso;

Exercício: Escreva um programa no qual números de ponto flutuante armazenados em uma estrutura podem ter precisão simples ou dupla (indicada em um campo). O programa deve ser capaz de identificar a precisão e utilizar as operações corretas em cada valor. Utilize uma enumeração para identificar a precisão.

```
struct Dado {
        enum Precisao {
                PRECISAO_DUPLA, PRECISAO_SIMPLES
        } precisao;
        union Valor {
                float f;
                double d;
        } valor;
};
```

```
struct Dado dado;

dado.precisao = PRECISAO_SIMPLES;
dado.valor.f = 10.1;
...
if (dado.precisao == PRECISAO_SIMPLES)
        printf("%f\n", dado.valor.f);
else
        printf("%lf\n", dado.valor.d);
```

Typedef

- Cria um "apelido" (*alias*) para um tipo (básico ou criado pelo usuário).
- Equivalência de nome: tipos criados com typedef são compatíveis se o tipo original é o mesmo.

```
typedef int celsius_temp;
typedef int fahrenheit_temp;

celsius_temp c;
fahreheit_temp f;

...

f = c;
```

Construção correta em linguagens como C, mas incorreta em linguagens como Ada, que adotam equivalência de nome estrita. (Caso fahrenheit_t fosse *alias* de float também funcionaria em C).

Checagem de erro

- Erros podem ser checados de duas maneiras: dinâmica e estática.
- A checagem dinâmica ocorre durante a execução, enquanto a estática ocorre em tempo de compilação.
- Como consequência, para verificar erros com checagem dinâmica é preciso executar o programa com dados de teste.
- Preferencialmente deve se fazer checagem estática quando possível.
- Apesar de mais eficiente, nem todo erro pode ser checado de maneira estática.

Checagem de tipo

- A checagem de tipo é responsável por verificar erros de tipo, isto é, operações entre variáveis de tipos incompatíveis.
- Um sistema de tipos é chamado de forte se garantir que programas escritos seguindo suas regras não gerarão erros de tipo.
- Uma linguagem é chamada de fortemente tipada se o seu sistema de tipos for forte.
- Em linguagens fortemente tipadas, o compilador é capaz de detectar erros de tipo.

- Um sistema de tipos estático é um no qual erros de tipo são encontrados em tempo de compilação.
- Linguagens estaticamente tipadas são linguagens fortemente tipadas.
- Pergunta: a linguagem C é estaticamente tipada?
- Pergunta: Quais as vantagens e desvantagens de uma linguagem formente tipada?

Exercício: No exemplo em que pontos podem estar no plano e no espaço, considere que a struct Ponto função soma(struct Ponto a, struct Ponto b) soma dois pontos de um mesmo "tipo" (no plano ou no espaço). Indique como retornar um erro durante a execução caso se tente somar um ponto no espaço com um ponto no plano. Perceba que você está implementando um sistema de checagem de tipo dinâmico e discuta os problemas e vantagens desse tipo de abordagem.

```
struct Ponto soma(struct Ponto a, struct Ponto b) {
    struct Ponto resultado;
       if (a.EstaNoPlano != b.EstaNoPlano) {
                printf("Tentando somar um ponto \
com duas coordenadas \
com um ponto com tres coordenadas\n");
                exit(1);
        }
if (a.EstaNoPlano) {
          resultado.valor.r2.x = a.valor.r2.x + \
        b.valor.r2.x;
          resultado.valor.r2.y = a.valor.r2.y + \
        b.valor.r2.y;
} else
          ...
        return(resultado);
}
```

Compatibilidade de Tipos

- Sistemas de tipo rigorosos podem requerer que uma operação que espera um operando de um tipo T só seja invocada sem erros com parâmetros desse mesmo tipo T.
- No entanto as linguagens em geral permitem maior flexibilidade, permitindo que operandos de outro tipo Q seja usado para invocar a operação sem violar as regras de tipagem.
- Dizemos então, que nessas linguagens T e Q são tipos compatíveis.
- A compatibilidade entre dois tipos em geral pode ser:
  1. Compatibilidade de nome.
  2. Compatibilidade estrutural.

Conversão de Tipos

- Em alguams situações é desejável que uma operação que espera um parâmetro do tipo T também possa ser invocada com um parâmetro do tipo Q.
- Para tal, são adotadas conversões de tipo na linguagem.
- Na grande maioria das linguanges, quando uma operação (soma ou multiplicação, por exemplo) é feita sobre um número de ponto flutuante e um inteiro, o inteiro é primeiramente convertido para ponto flutuante e então a operação é realizada retornando um número de ponto flutuante (isto é, 2 + 1.1 retornará 3.1).

Exercício: Considere agora que você deseja simular no programa anterior o comportamento de uma linguagem na qual uma operação entre um ponto no plano e um ponto no espaço converte o ponto de duas coordenadas para um ponto de três coordenadas. Quando a operação de soma for invocada com um ponto de duas coordenadas e um ponto de três coordenadas, interpretamos o ponto de duas coordenadas como um ponto no plano $z = 0$ e efetuamos a soma como se tivessemos recebido dois pontos no espaço.

```
        if (a.EstaNoPlano != b.EstaNoPlano) {
                if (a.EstaNoPlano) {
                        a.EstaNoPlano = 0;
                        a.valor.r3.z = 0;
                } else {
                        b.EstaNoPlano = 0;
                        b.valor.r3.z = 0;
                }
        }
        if (a.EstaNoPlano) {
                resultado.valor.r2.x = a.valor.r2.x + \
                                b.valor.r2.x;
                resultado.valor.r2.y = a.valor.r2.y + \
b.valor.r2.y;
        } else {
                resultado.valor.r3.x = a.valor.r3.x +\
 b.valor.r3.x;
...
        }
```

Variáveis, formalmente são compostas por:

1. Nome.
2. Escopo.
3. Tipo.
4. l_value.
5. r_value.

Escopo de variáveis

- Amarração de escopo estática: definida pelo código.
- Amarração de escopo dinâmica: definida pelo fluxo de controle da execução.

Analise o seguinte exemplo para ambos os tipos de amarração de escopo.

```
{
    //Bloco A
    int x;
    ....
}
...
{
    //Bloco B
    int x;
    ...

}
...
{
    //Bloco C
    x = x + 1;
    ..

}
```

O l_value (valor_l) é a área de armazenamento amarrado à variável durante tempo de execução. O tempo de vida de uma variável é o período de tempo no qual essa amarração existe. Essa área de armazenamento é usada para guardar o r_value da variável.

Alocação de Memória é a ação que obtem a área para armazenar a variável e, portanto, estabelece a amarração. O **tempo de vida** da variável se estende do momento da alocação até o momento da dealocação. A alocação pode ser estática ou dinâmica.

O **r_value** da variável é o valor codificado e armazenado na posição de memória apontada pelo l_value. A representação codificada é interpretada de acordo com o tipo da variável.

A amarração entre a variável e seu valor é em geral dinâmica. Há, no entanto exceções como a amarração de valor a variáveis do tipo `const` na linguagem Pascal, que deve ser feita pelo compilador.

Classificação de linguagens de acordo com sua estrutura de tempo de execução:

1. Linguagens Estáticas: Todos os requisitos de memória devem ser avaliados antes do início da execução. Dessa maneira toda a memória necessária pode ser alocada antes do programa realmente iniciar. Dessa forma, essas linguagens não permitem recursividade. Exemplos: As primeiras versões de FORTRAN e COBOL.
2. Linguagens Baseadas em Pilha: alocações feitas dinâmicamente seguinto a disciplina *last-in-first-out*.
3. Linguagens totalmente Dinâmicas: permitem o uso de memória totalmente imprevisível. Introduzem o conceito de *heap* para o armazenamento de variáveis alocadas dinamicamente.

Passagem de parâmetro para funções:

1. Chamada por resultado: copia ao sair da função.
2. Chamada por valor: copia ao entrar na função.
3. Chamada por valor-resultado: copia ao entrar e ao sair da função.
4. Chamada por referência: passa um ponteiro.
5. Chamada por nome: reavalia a expressão a cada uso.

```
begin
integer n;
procedure p(k: integer);
    begin
    n := n+1;
    k := k+4;
    print(n);
    end;
n := 0;
p(n);
print(n);
end;
```

- Chamada por valor: 1 1
- Chamada por valor-resultado: 1 4
- Chamada por referência: 5 5

```
begin
integer n;
procedure p(k: integer);
    begin
    print(k);
    n := n+1;
    print(k);
    end;
n := 0;
p(n+10);
end;
```

- Chamada por valor: 10 10
- Chamada por nome: 10 11