Geradores de analisadores sintáticos

Introdução ao Bison

O que é Yacc?

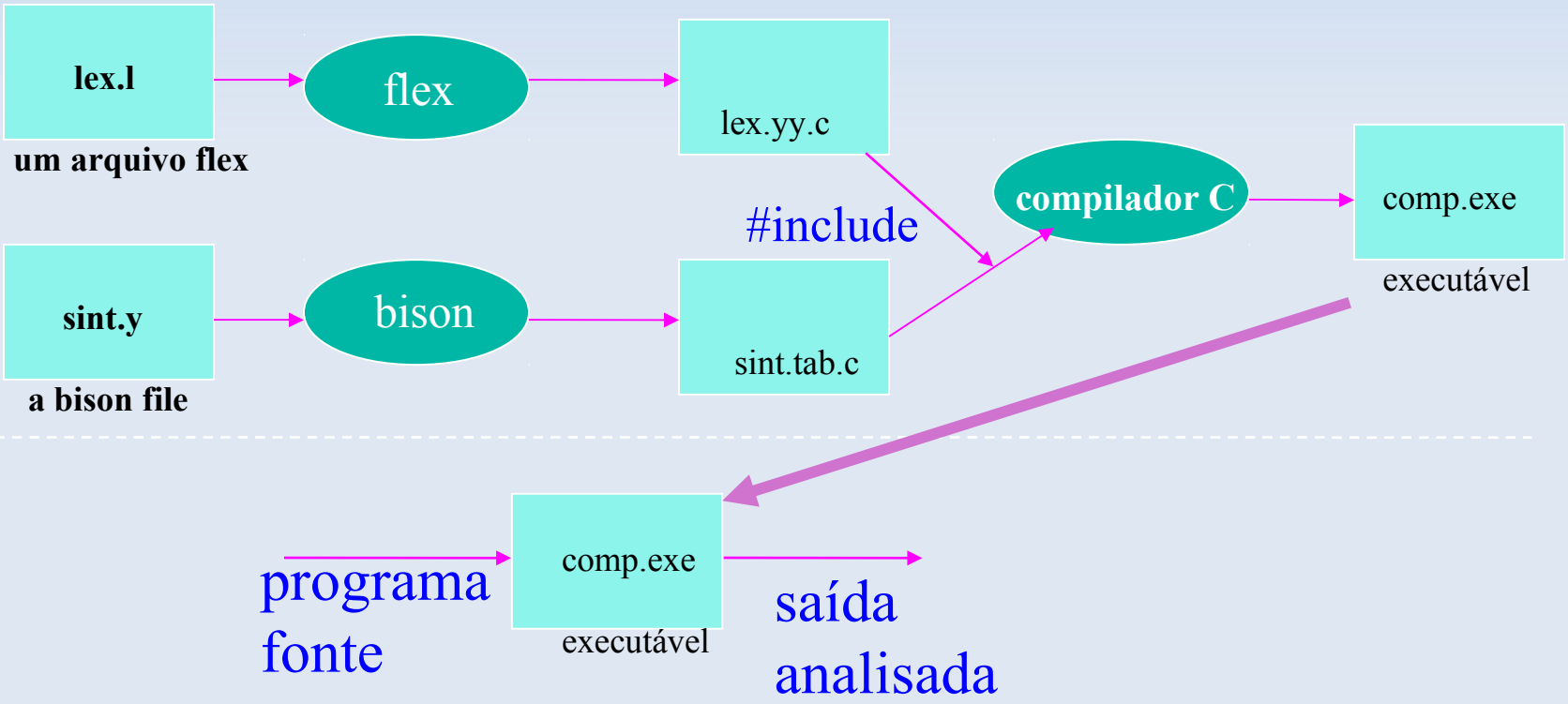
- Yacc (Yet Another Compiler Compiler) é uma ferramenta para traduzir uma gramática independente de contexto em analisador sintático LALR
  - Ele cria uma tabela sintática como a descrita na última aula
- Yacc é usado com lex para criar compiladores.

Bison

- Muitas pessoas usam o bison, que é uma versão aperfeiçoada do YACC
- bison trabalha em conjunto com flex (a versão mais rápida do lex).

Bison and Flex

```
$ flex lex.l
$ bison sint.y
$ gcc lex.yy.c sint.tab.c -o comp.exe
```




```
$ ./comp.exe < programa.txt
```

Visão geral do bison

Propósito: gerar um analisador sintático para uma gramática escrita em BNF.

Uso: deve-se escrever um arquivo fonte bison contendo regras em BNF.

O bison cria um programa em C que analisa

```
termo     : termo '*' fator { $$ = $1 * $3; }
          | termo '/' fator { $$ = $1 / $3; }
          | fator           { $$ = $1; }
          ;
fator     : ID              { $$ = valor_de($1); }
          | NUM             { $$ = $1; }
          ;
```

Visão geral do bison (2)

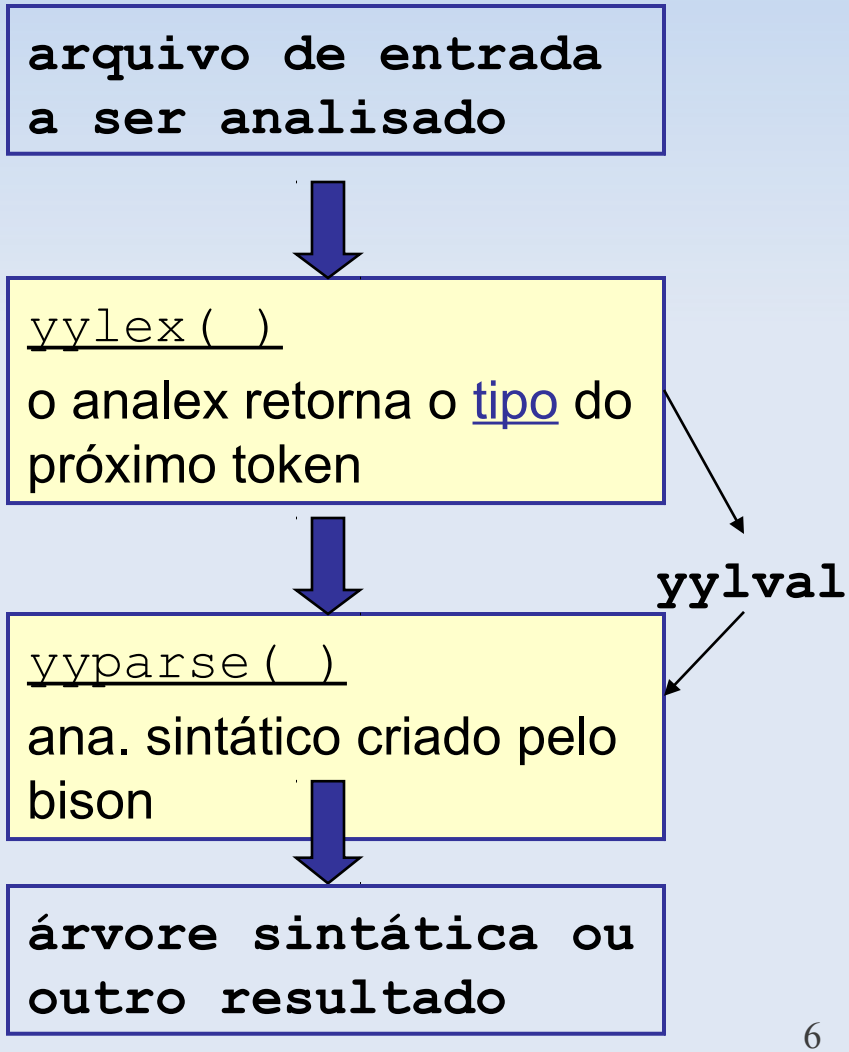
Em operação:

seu programa principal chama `yyparse( )`.

`yyparse( )` chama `yylex` quando ele quer um token.

`yylex` retorna o **tipo** do token.

`yylex` coloca o **valor** do token em uma variável global chamada `yylval`



Arquivo fonte do bison

O arquivo tem 3 seções separadas por "%%".

```
/* as declarações ficam aqui */

%%
/* as regras gramaticais ficam aqui */

%%
/* código C adicional fica aqui */
```

Arquivo fonte do bison com declarações em C

Normalmente inclui-se código C na seção de declarações.

```
%{
  /* declaraçẽos em C e as macros #define ficam aqui */
  #include <stdio.h>
  #define YYSTYPE double
%}
/* as declarações do bison ficam aqui */

%%
/* as regras gramaticais ficam aqui */

%%
/* código C adicional fica aqui */
```

Declara que **yylval** será do tipo "double".

Exemplo em bison

Criar um analisador sintático para a seguinte gramática:

```
expressao => expressao + termo
           | expressao - termo
           | termo
termo     => termo * fator
           | termo / fator
           | fator
fator     => ( expressao )
           | NUM
```

Arquivo bison/Yacc para o exemplo (1)

Estrutura de uma entrada Bison ou Yacc:

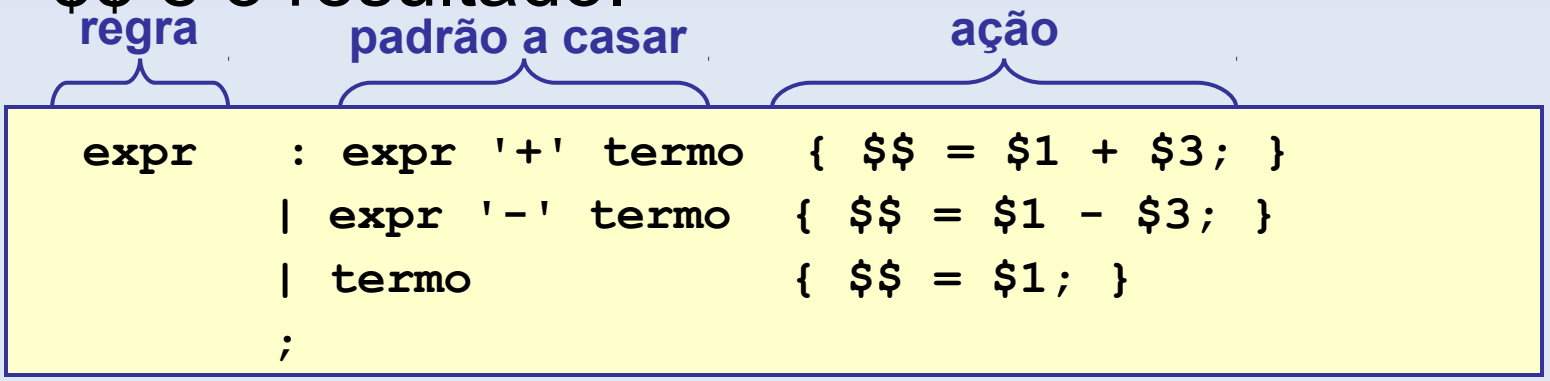
```
%{
/* declarações em C e macros #DEFINE */
  #include <stdio.h>
  #define YYSTYPE double
%}
/* declarações do bison */
%token NUM      /* define o tipo do token NUM */
%left '+' '-'    /* + e - são associativos à esquerda */
%left '*' '/'    /* * e / são associativos à esquerda */

%%
/* regras gramaticais ficam aqui */
%%
/* código C adicional fica aqui */
```

Exemplo do bison (2)

```
%%          /* regras gramaticais no Bison */
entrada : /* produção vazia para permitir uma entrada
  vazia */
        | entrada linha
        ;
linha   : expr '\n'   { printf("O resultado é %f\n",
  $1); }
expr    : expr '+' termo    { $$ = $1 + $3; }
        | expr '-' termo    { $$ = $1 - $3; }
        | termo             { $$ = $1; }
        ;
termo   : termo '*' fator { $$ = $1 * $3; }
        | termo '/' fator { $$ = $1 / $3; }
        | fator           { $$ = $1; }
        ;
fator   : '(' expr ')'    { $$ = $2; }
        | NUM             { $$ = $1; }
        ;
```

Exemplo do bison (3)

- $1, $2, ... representam os valores reais dos tokens ou não-terminais (regras) que se casam com a produção.
- $$ é o resultado.

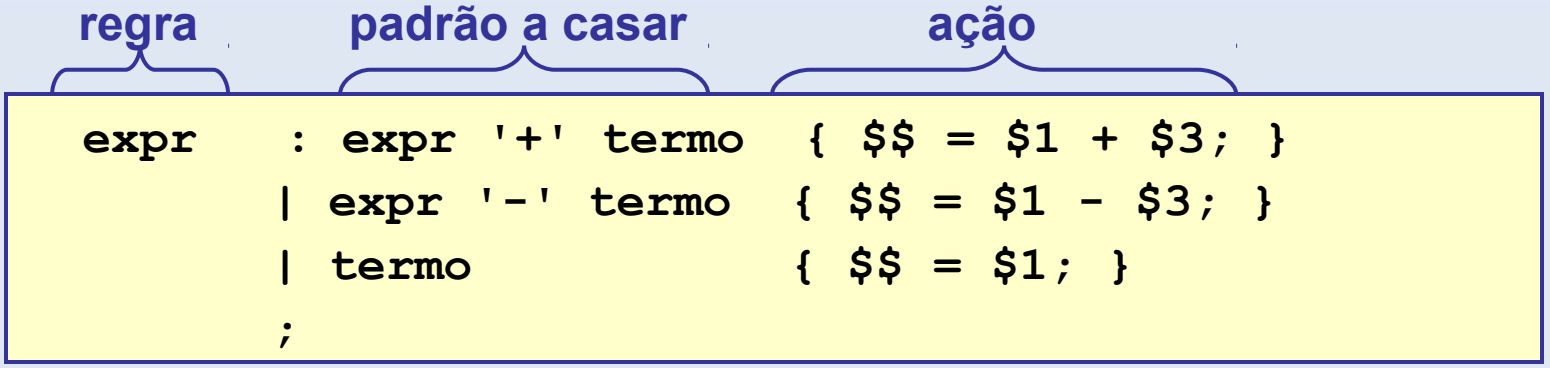


Exemplo:
se a entrada casa com **`expr + termo`** então o resultado (**`$$`**) é igual à soma de **`expr`** com **`termo`** (**`$1 + $3`**).

Exemplo do bison (4)

Q: por que podemos escrever "`$$ = $1 + $3`" ?

A: porque declaramos "`#define YYSTYPE double`", assim todos os tokens e resultados são `double`.



Regras

- Formato das regras:

```
naoterminal : corpo 1   {ação 1}
            | corpo 2   {ação 2}
            . . .
            | corpo n   {ação n)
            ;
```

- As ações são opcionais e são código em C.
- As ações são colocadas normalmente no final de um corpo de regra, mas podem ser colocadas em qualquer posição no corpo.

Função de varredura: `yylex( )`

- Deve-se fornecer um analisador léxico chamado **`yylex`**.

```
int yylex( void ) {
    int c = getchar();      /* leia de stdin */
    if (c < 0) return 0;    /* fim da entrada*/
    if ( c == '+' || c == '-' ) return c;
    /* para tokens caracteres, TIPO = ao próprio caracter */
    if ( isdigit(c) ) {
        yylval = c - '0'; /* yylval é uma variável global */
        while( isdigit( c=getchar() ) )
            yylval = 10*yylval + (c - '0');
        if (c >= 0) ungetc(c,stdin);
        return NUM;  /* o tipo do token é NUM  */
    }
...
}
```

Onde está o *valor* do token?

- O **valor** do token está armazenado em uma variável global chamada `yylval`.

```
int yylex( void ) {
    int c = getchar();      /* leia de stdin */
    if (c < 0) return 0;    /* fim da entrada*/
    if ( c == '+' || c == '-' ) return c;
    /* para tokens caracteres, TIPO = ao próprio caracter */
    if ( isdigit(c) ) {
        yylval = c - '0'; /* yylval é uma variável global */
        while( isdigit( c=getchar() ) )
            yylval = 10*yylval + (c - '0');
        if (c >= 0) ungetc(c,stdin);
        return NUM;  /* o tipo do token é NUM  */
    }
...
}
```

Outras funções em C: yyerror

- Bison requer uma função de erro chamada de **`yyerror`**.
- **`yyerror`** é chamada pelo analisador sintático quando existir um erro.

```
/* exibe mensagens de erro */
int yyerror( char *msg ) {
     printf("%s\n", msg);
}
```

Outras funções em C: main

- Deve-se escrever uma função **`main()`** que inicia o analisador sintático.
- Para um analisador sintático simples, **`main()`** somente chama **`yyparse()`**.

```
/* função principal para executar o programa */
int main( ) {
     printf("Digite alguma entrada.\n");
     yyparse( );
}
```

Executando Bison

- Compile o arquivo ***simples.y***

```
CMD> bison simples.y
```

- A saída é "***simples*.tab.c**", que é o código C para o analisador sintático.

Exemplo simples: definições

Arquivo: **simples.y**

```
/* A seção de declarações do bison */
%{
/* declarações em C e macros #DEFINE */
#include <math.h>
#define YYSTYPE double
%}
%token NUM        /* define o tipo do token para números
  */
%token '+' '-'    /* + e - são associativos à esquerda
  */
```

Nenhuma associatividade (left/right) especificada

Exemplo simples: regras gramaticais

```
%%          /* regras gramaticais */
entrada  : /* permite entrada vazia */
         | entrada linha
         ;
linha    : expr '\n'   { printf("resposta: %d\n", $1); }
expr     : expr '+' termo   { $$ = $1 + $3; }
         | expr '-' termo   { $$ = $1 - $3; }
         | termo            { $$ = $1; }
         ;
termo    : NUM          { $$ = $1; }
         ;
```

yyerror e main

```
%% /* código C extra */
/* exibe mensagem de erro */
int yyerror( char *msg ) { printf("%s\n", msg); }

/* main */
int main() {
    printf("digite uma expressao:\n");
    yyparse( );
}
```

Exemplo simples: explorando a BNF

```
%%          /* regras gramaticais */
entrada  : /* entrada vazia */
         | entrada linha
         ;
linha    : expr '\n'    { printf("resposta: %d\n", $1); }
expr     : expr '+' expr    { $$ = $1 + $3; }
         | expr '-' expr    { $$ = $1 - $3; }
         | termo            { $$ = $1; }
         ;
termo     : NUM             { $$ = $1; }
         | '-' NUM          { $$ = -$2; }
         ;
```

```
%token NUM
%right '+' '-'
```

Exercício

Expandir a gramática para incluir estas operações:

4 * 5 multiplicação

2 / 3 divisão

10 + 3 * 4 – 1 / 2 ordem correta das operações

2 * (3 + 4) agrupamento

Exemplo completo: definições

Arquivo: **simples.y**

```
/* A seção de declarações do Bison */
%{
/* declarações em C e macros #DEFINE */
#include <math.h>
#define YYSTYPE double
%}
%token NUM          /* define o tipo do token para números
  */
%left '+' '-'       /* + e - são associativos à esquerda
  */
%left '*' '/'       /* * e / são associativos à esquerda
  */
```

Exemplo completo: regras gramaticais

```
%%          /* regras gramaticais */
entrada  : /* permite entrada vazia */
         | entrada linha
         ;
linha    : expr '\n'    { printf("Resultado = %f\n",
  $1); }
expr     : expr '+' termo    { $$ = $1 + $3; }
         | expr '-' termo    { $$ = $1 - $3; }
         | termo             { $$ = $1; }
         ;
termo    : termo '*' fator { $$ = $1 * $3; }
         | termo '/' fator { $$ = $1 / $3; }
         | fator           { $$ = $1; }
         ;
fator    : '(' expr ')'     { $$ = $2; }
         | NUM              { $$ = $1; }
         | '-' NUM          { $$ = -$2; }
         ;
```

Erros comuns em Bison

1. Esquecer de colocar os literais entre apóstrofes: termo `'+'` `termo`
2. Não saltar espaço onde o espaço é permitido

```
%token NUM
%token + -
%% /* regras gramaticais */

expr    : termo + termo      { $$ = $1 + $3; }
        | termo - termo      { $$ = $1 - $3; }
        | termo            { $$ = $1; }
termo   : NUM       { $$ = $1; }
        | - NUM        { $$ = -$2; }
```

Desloca / Reduz e ordem de operadores

- Bison usa uma pilha e também faz a antecipação de tokens (look-ahead). Ele *desloca* tokens para a pilha até que ele possa escolher qual regra usar para *reduzir* (substituir) os tokens com um não-terminal. Exemplo:

```
expr ::=  expr + expr
      |   expr - expr
      |   expr * expr
      |   expr / expr
      |   termo
termo ::=  NUM | ( expr )
```

- Suponha que a entrada lida seja `10 -`
- *desloque* estes tokens para a pilha pois nenhuma regra pode ser aplicada ainda.
- Suponha que o próximo token seja `2`. O que o bison deveria fazer?

Desloca / Reduz e ordem dos operadores

PILHA: 10 - TOKEN ATUAL: 2

- Esta gramática é ambígua. O Bison poderia usar "***expr*** **-** ***expr***" ou poderia deslocar 2 para a pilha e procurar o próximo token, que talvez seja um ***** como em: **10 - 2 * 3**
- Isto é chamado um "conflito desloca / reduz".
- A menos que você especifique uma *regra de desambiguação* (próx. slide), Bison prefere "deslocar" a "reduzir".
- Significado: *se não é claro como resolver conflitos, espere*.

ENTRADA	AÇÃO	PILHA
10	*desloca*	*10*
-	*desloca*	*10 -*
2	*desloca*	*10 - 2*
***	*desloca*	*10 - 2 **

ENTRADA	AÇÃO	PILHA
3	*reduz*	*10 - 6*
	reduz	*4*

Depurando gramáticas

- O bison pode gerar informações extras sobre conflitos, que poderão auxiliá-lo na depuração de sua gramática.
  - use a opção -v

```
$ bison gramatica.y

gramatica.y: conflicts: 4 shift/reduce

$ bison -v gramatica.y
gramatica.y: conflicts: 4 shift/reduce
```

Ele cria um arquivo chamado `gramatica.output` com informações extras sobre conflitos

Dentro de gramatica.output

```
State 9 conflicts: 2 shift/reduce
State 10 conflicts: 2 shift/reduce


Grammar

    0 $accept: expr $end

    1 expr: expr '+' expr
    2     | expr '*' expr
    3     | '(' expr ')'
    4     | NUMBER

        :  // many state blocks
```

estados 9 e 10 são os problemas

As regras são numeradas

Dentro de gramatica.output

quando bison está nestes tipos de estados

```
state 9

    1 expr: expr . '+' expr
    1     | expr '+' expr .
    2     | expr . '*' expr

    '+'  shift, and go to state 6
    '*'  shift, and go to state 7

    '+'        [reduce using rule 1 (expr)]
    '*'        [reduce using rule 1 (expr)]
    $default  reduce using rule 1 (expr)
```

bison faz isto

mas ele deveria fazer isto

Dentro de gramatica.output

quando bison está nestes tipos de estados

```
state 10

    1 expr: expr . '+' expr
    2     | expr . '*' expr
    2     | expr '*' expr .

    '+'  shift, and go to state
    '*'  shift, and go to state 7

    '+'        [reduce using rule 2 (expr)]
    '*'        [reduce using rule 2 (expr)]
    $default   reduce using rule 2 (expr)
```

bison faz isto

mas ele deveria fazer isto

A seção de declarações

- A seção de declarações pode conter diretivas do Bison e também do C.
- Já vimos o uso de %left, %right, etc.

```
%{
/* valor semântico dos tokens é um tipo de dados do C */
    #define YYSTYPE double
    #include <math.h>
    /* por que temos que declarar as linhas abaixo? */
    int yylex (void);
    void yyerror (char const *);
%}
%token NUM
%left  '+' '-'
%left  '*' '/'
```

Especificando a ordem dos operadores

- Na seção de declarações você pode escrever:

```
%noassoc NEG      /* sinal de menos unário: - 3 */
%left  '+' '-'
%left  '*' '/' '%'
%right '^'
```

- `'+'` e `'-'` são associativos à **esquerda** e possuem a **mesma precedência**.
- `'*'`, `'/'`, e `'%'` são associativos à **esquerda** e possuem a **mesma precedência**; entretanto, eles tem precedência mais ***alta.***
- `'^'` é assocativo à **direita** e tem precedência **mais alta** que `+ - * / %`
- '**`NEG`**' não possui associatividade: `"- - 3"` é um erro

Pontos principais da função de varredura

- Os tokens tem um TIPO e um VALOR.
- O analisador léxico (yylex) retorna o TIPO do próximo token
- Para tokens de um caracter como '+', '=', "(' o próprio caracter pode ser usado como o tipo.
- Defina nomes simbólicos para os tipos dos tokens em Bison usando **`%token NOME`**
- Retorne o VALor de um token usando a variável global **`yylval`**.
- Por default, **`yylval`** é do tipo "int". Altere-o usando: **`#define YYSTYPE double`**

Pontos principais da função de varredura (1)

```
%{
    #include <ctype.h>
    #include <math.h>
    #define YYSTYPE double
%}
%token NUM
%token SQRT
%left  '+' '-'
%left  '*' '/'
%%
linha  : expr '\n' { printf("%g\n", $$); }
       ;
expr   :  expr * expr { $$ = $1 + $3; }
       |  SQRT '(' expr ')' { $$ = sqrt( $2 ); }
       ...
       /* mais regras */
```

Os valores dos tokens são double

Token para a função raiz quadrada

Pontos principais da função de varredura (2)

```
%%
int yylex(void) {
    int c = getchar( );
    while ( c == ' ' || c == '\t' ) c = getchar( );
    if ( isdigit(c) ) { ungetc(c,stdin);
        scanf("%f", &yylval);
        return NUM;
    }
    if ( isalpha(c) ) {
        char *word = getword(c);  /* get next word */
        if ( strcmp(word,"sqrt") ) return SQRT;
    ...
}
```

Token é um número

Token é a função sqrt

Tratando tipos de dados múltiplos

- Para uma gramática mais geral, o analex deveria poder retornar diferentes tipos de dados como o valor de **`yylval`**.
- Na seção de definições, defina "**`%union`**" como a união de todos os tipos de dados que **`yylval`** (e assim $1, $2, ...) podem ter.

Defina todos os tipos de dados que os valores dos tokens podem ter

Para cada tipo de token, defina o tipo de dado de seu valor.

Para tokens que representam seu próprio valor não é necessário definir um tipo de dados.

```
%union {
    double number;
    char*  string;
    }
%token <number> NUM
%token <string> IDENT
%type  <number> expr
%type  <number> termo
%left  '+' '-'
%left  '*' '/'
```

Tratando tipos de dados múltiplos (2)

```
%%
int yylex(void) {
    int c = getchar( );
    while ( c == ' ' || c == '\t' ) c = getchar( );
    if ( isdigit(c) ) { ungetc(c,stdin);
        double x;
        scanf("%f", &x);
        yylval.number = x;
        return NUM;
    }
    if ( isalpha(c) ) {
        char *pal = getword(c);/* obtem a próx. palavra */
        yylval.string = pal;
        return IDENT;
    ...
}
```

o valor do token é double

o valor do token é uma string

Usando um arquivo separado para yylex

- Pode-se colocar o analex (yylex) em um arquivo separado.
- MAS, **`yylex`** precisa de valores que são definidos em Bison, tais como NUM, **`IDENT`**, **`yylval`**.
- Solução: adicionar a opção "%defines" nas suas regras
- Bison criará um arquivo de cabeçalho chamado "***`simples`*`.tab.h`**".

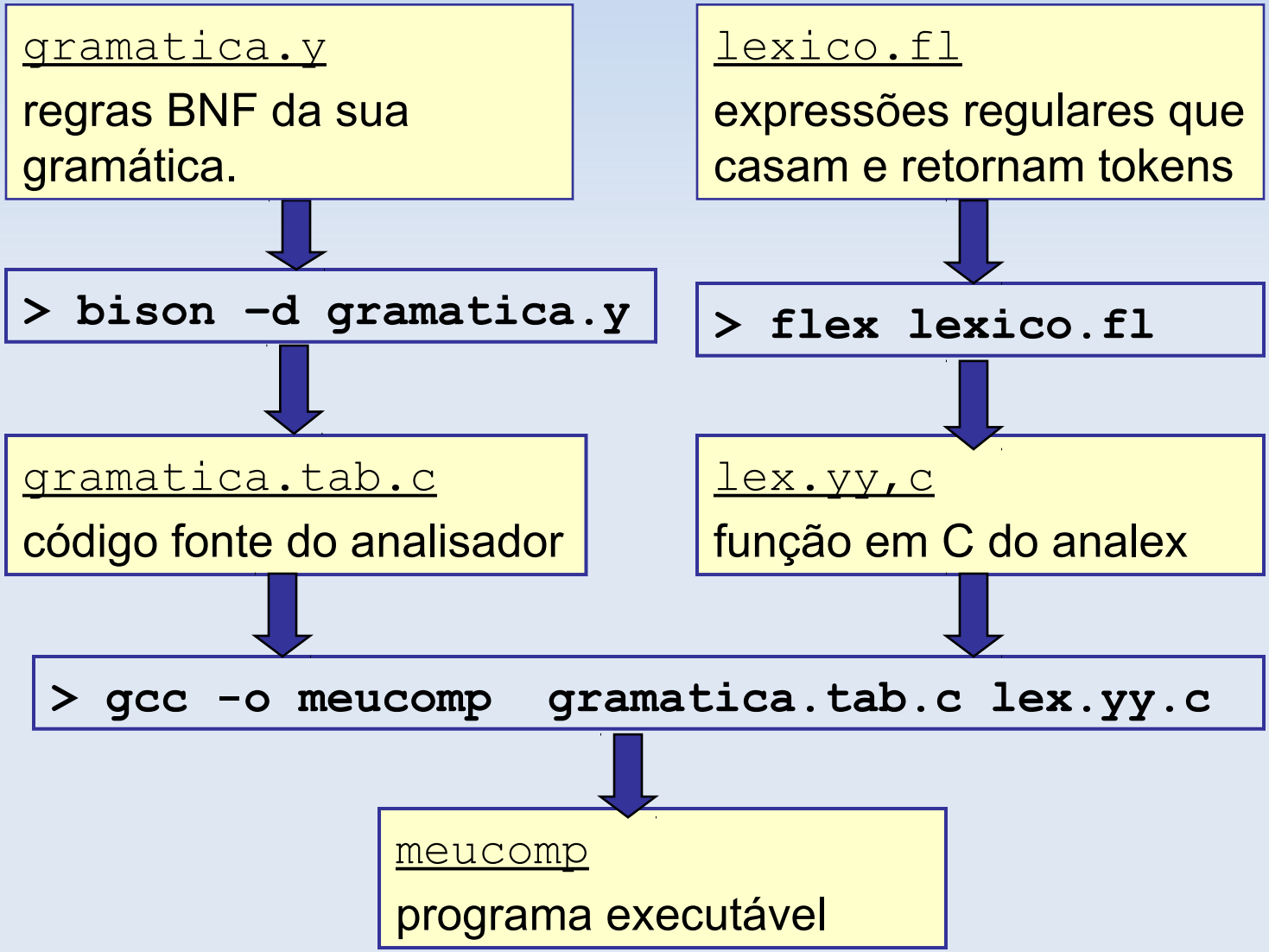
```
%defines
%{
    #include <math.h>
    #define YYSTYPE double
%}
token NUM
...
```

Usando flex com bison

1. Execute "**`bison -d arquivo.y`**" to create **`arquivo.tab.h`**
2. **`#include "arquivo.tab.h"`** no arquivo fonte do flex.
3. Atribua cada **`yylval`** ao valor do token.
4. Execute flex.
5. Compile e ligue **`lex.yy.c`** e **`arquivo.tab.c`**

```
%{
    #include <math.h>
    #include "arquivo.tab.h"
%}
LETRA  [A-Za-z]  /* mesmo que [[:alpha:]] */
%%
-?[0-9]+    yylval.itype = atoi(yytext); return INT;
{LETRA}+    yylval.ctype = yytext; return IDENT;
"+"|"-"     yylval.ctype = yytext; return OP;
"="          return ASSIGN;
```

Usando Flex e Bison



Recuperação de erro

- Quando um erro ocorre, bison chama yyerror() e depois termina.

- Uma melhor abordagem é chamar yyerror(), e depois tentar continuar
  - Isto pode ser feito usando a palavra reservada `error` nas regras gramaticais

Exemplo

- Se existir um erro na regra `cmd` , então ignore o restante dos tokens da entrada até que ';' ou '}' sejam vistos, depois continue como antes:

```
cmd : ';'
        | expr ';'
        | VAR '=' expr ';'
        | '{' cmd_list '}'
        | error ';'
        | error '}'
        ;
```

Ações encaixadas

- As ações podem ser colocadas em qualquer lugar em uma regra, não somente no final:

```
par: item1 { trata_item1($1); }
     item2 { trata_item2($3); }
```

A variável de ação no segundo bloco de ação é $3 pois o primeiro bloco de ação é contado como parte da regra

Onde encontrar o Bison?

- No Linux
  - sudo apt-get install bison
- No Windows
  - http://gnuwin32.sourceforge.net/packages/bison.htm
- As mesmas referências bibliográficas que o Flex

Geradores sintáticos disponíveis para Java

- CUP: gerador de analisadores sintáticos do tipo LALR
  - http://www2.cs.tum.edu/projects/cup/
  - Livro *Computer Language Implementation* disponível em http://www.

    cs.auckland.ac.nz/~bruce-h/lectures/330ChaptersPDF/, cap. 4.
- JavaCC: gerador de analisadores sintáticos do tipo LL(k)
  - http://javacc.java.net/
- ANTLR: gerador de analisadores sintáticos do tipo LL(k)
  - http://www.antlr.org/
  - Pode gerar analisadores em várias linguagems: Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python, and Ruby.

Geradores sintáticos disponíveis para Haskell

- Happy: gerador de analisadores sintáticos do tipo LALR
  - http://haskell.org/happy/
- Mesmas referências bibliográficas do Alex