

# Projeto de Compilador: Etapa 2 de Análise Sintática

Prof. Lucas Mello Schnorr (INF/UFRGS)

schnorr@inf.ufrgs.br

O trabalho consiste no projeto e implementação de um compilador funcional para uma linguagem de programação que a partir de agora chamaremos de:

## Linguagem POA

Na segunda etapa do trabalho é preciso construir um analisador sintático utilizando a ferramenta de geração de reconhecedores `bison` e continuar o preenchimento da tabela de símbolos com outras informações, associando valores aos `tokens` (através da variável global `yylval`). O analisador sintático deve portanto verificar se a sentença fornecida – o programa de entrada a ser compilado – faz parte da linguagem ou não.

## 1 Funcionalidades Necessárias

### 1.1 Definir a gramática da linguagem

A gramática da linguagem POA deve ser definida a partir da descrição geral da Seção 2. As regras gramaticais devem ser incorporadas ao arquivo `parser.y`, arquivo este que conterá a gramática usando a sintaxe do `bison`.

### 1.2 Relatório de Erro Sintático

Caso a análise sintática termine de forma correta, o programa deve retornar zero. Na ausência de erros sintáticos, esse valor é automaticamente retornado pelo `bison` através da função `yyparse()`, chamada pela função `main` do programa. Caso a entrada não seja reconhecida, deve-se imprimir uma mensagem de erro informando a linha do código da entrada que gerou o erro sintático e informações adicionais que auxiliem o programador que está utilizando o compilador a identificar o erro sintático identificado. Na ocasião de uma mensagem de erro, o analisador sintático deve retornar um valor diferente de zero. Esses valores – zero ou diferente de zero – são utilizados durante a avaliação.

### 1.3 Enriquecimento da tabela de símbolos

Uma vez que vários lexemas da entrada podem representar `tokens` de tipos diferentes, a tabela de símbolos deve ser alterada de forma que a chave de cada uma das entradas não seja mais simplesmente o lexema, mas a combinação entre o lexema e o tipo do `token`. O tipo de um determinado `token` pode ser somente um dentre as seguintes constantes. Elas estão definidas no arquivo `main.h` do repositório e podem ser livremente utilizadas em qualquer parte do código.

```
#define POA_LIT_INT      1
#define POA_LIT_FLOAT    2
#define POA_LIT_CHAR     3
#define POA_LIT_STRING   4
#define POA_LIT_BOOL     5
#define POA_IDENT        6
```

O conteúdo de cada entrada na tabela de símbolos deve ter pelo menos três campos: número da linha da última ocorrência do lexema, o tipo do `token` da última ocorrência, e o valor do `token` convertido para o tipo apropriado (inteiro `int`, ponto-flutuante `float`, caractere `char`, booleano `bool` ou cadeia de caracteres `char*`). O segundo campo, representado pelo tipo do `token` deve ser o mesmo utilizado na chave da entrada.

O valor do `token` é um campo que pode assumir diferentes tipos: uma possibilidade é utilizar a construção `union` da linguagem C para conter os diferentes tipos possíveis para os símbolos. A conversão deve ser feita utilizando funções tais como `atoi`, no caso de números inteiros, e `atof`, no caso de ponto-flutuantes. Os tipos caractere e cadeia de caracteres não devem conter aspas no campo valor (e devem ser duplicados com `strdup`).

### 1.4 Associação de valor ao token (`yylval`)

O analisador léxico é o responsável pela criação da entrada na tabela de símbolos para um determinado `token` que acaba de ser reconhecido. Nesta etapa, deve-se associar um ponteiro para a estrutura de dados que representa o conteúdo da entrada na tabela de símbolos ao `token` correspondente. Esta associação deve ser feita pelo analisador léxico (ou seja, no arquivo `scanner.l`).

Ela é realizada através do uso da variável global `yylval`<sup>1</sup> que é usada pelo `flex` para dar um “valor” ao `token`, além do identificador (um número inteiro, como na E1) retornado imediatamente após o reconhecimento. Como esta variável global pode ser configurada com a diretiva `%union`, sugere-se o uso do campo `valor_lexico` para a associação. Portanto, a associação deverá ser feita através de uma atribuição para a variável `yylval.valor_lexico`. O tipo do `valor_lexico` deve ser um ponteiro para uma entrada na tabela de símbolos.

### 1.5 Remoção de conflitos gramaticais

Deve-se realizar a remoção de conflitos `Reduce/Reduce`<sup>2</sup> e `Shift/Reduce`<sup>3</sup> de todas as regras gramaticais. Es-

<sup>1</sup>[http://www.gnu.org/software/bison/manual/html\\_node/Token-Values.html](http://www.gnu.org/software/bison/manual/html_node/Token-Values.html)

<sup>2</sup>[http://www.gnu.org/software/bison/manual/html\\_node/Reduce\\_002fReduce.html](http://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html)

<sup>3</sup>[http://www.gnu.org/software/bison/manual/html\\_node/Shift\\_002fReduce.html](http://www.gnu.org/software/bison/manual/html_node/Shift_002fReduce.html)

tes conflitos devem ser tratados através do uso de configurações para o bison (veja a documentação sobre `%left`, `%right` ou `%nonassoc`). Os mesmos podem ser observados através de uma análise cuidadosa do arquivo `parser.output` gerado automaticamente quando compilado. Sugere-se um processo construtivo da especificação em passos, verificando em cada passo a inexistência de conflitos. Por vezes, a remoção de conflitos pode ser feita somente através de uma revisão de partes da gramática.

## 1.6 Listar o conteúdo tabela de símbolos

Implementar a função `comp_print_table` para listar todas as entradas da tabela de símbolos. Utilize a função `void cc_dict_etapa_2_print_entrada(char *key, int line, int tipo)` para imprimir uma entrada. Esta função será utilizada na avaliação para averiguar se a solução preenche a tabela de símbolos.

# 2 A Linguagem POA

Um programa na linguagem POA é composto por três elementos, todos opcionais: um conjunto de declarações de variáveis globais, um conjunto de declarações de novos tipos, um conjunto de funções. Esses elementos podem aparecer intercaladamente e em qualquer ordem.

## 2.1 Declarações de Novos Tipos

Novos tipos podem ser declarados apenas no escopo global em POA através da palavra reservada `class`, seguida de um nome e enfim uma lista de campos fornecida entre colchetes onde os campos são separados por dois pontos (através do caractere especial `':'`). Cada campo tem o encapsulamento, o tipo e um identificador do campo. Existem três encapsulamentos possíveis, identificados pelas palavras reservadas: `protected`, `private`, e `public`. Declarações de novos tipos são terminadas por ponto-e-vírgula. O tipo de um campo que faz parte de uma declaração de novo tipo não pode ser um tipo de usuário, ou seja, um tipo declarado com `class`.

## 2.2 Declarações de Variáveis Globais

As variáveis são declaradas pelo seu tipo, seguidas pelo seu nome. O tipo pode estar precedido opcionalmente pela palavra reservada `static`. A linguagem inclui também a declaração de vetores, feita pela definição de seu tamanho inteiro positivo entre colchetes, colocada à direita do nome, ou seja, ao final da declaração. Variáveis podem ser dos tipos primitivos `int`, `float`, `char`, `bool` e `string`; e também podem ser dos tipos declarados pelo usuário. Neste último caso, o nome do tipo é aquele utilizado depois da palavra reservada `class` quando este foi declarado. As declarações de variáveis globais são terminadas por ponto-e-vírgula.

## 2.3 Definição de Funções

Cada função é definida por um cabeçalho e um corpo, sendo que esta definição não é terminada por ponto-e-vírgula. O cabeçalho consiste no tipo do valor de retorno,

seguido pelo nome da função e terminado por uma lista. O tipo pode estar precedido opcionalmente pela palavra reservada `static`. A lista é dada entre parênteses e é composta por zero ou mais parâmetros de entrada, separados por vírgula. Cada parâmetro é definido pelo seu tipo e nome, e não pode ser do tipo vetor. O tipo de um parâmetro pode ser opcionalmente precedido da palavra reservada `const`. O corpo da função é um bloco de comandos.

## 2.4 Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência, possivelmente vazia, de comandos simples cada um **terminado** por ponto-e-vírgula. Um bloco de comandos é considerado como um comando único simples, recursivamente, e pode ser utilizado em qualquer construção que aceite um comando simples.

## 2.5 Comandos Simples

Os comandos simples da linguagem podem ser: declaração de variável local, atribuição, construções de fluxo de controle, operações de entrada, de saída, e de retorno, um bloco de comandos, e chamadas de função.

### Declaração de Variável

Consiste no tipo da variável precedido opcionalmente pela palavra reservada `static`, e o nome da variável. Os tipos podem ser aqueles descritos na Seção 2.2. As declarações locais, ao contrário das globais, não permitem vetores e podem permitir o uso da palavra reservada `const` antes do tipo (após a palavra reservada `static` caso esta aparecer). Uma variável local pode ser opcionalmente inicializada com um valor válido caso sua declaração seja seguida do operador composto `"<="` e de um identificador ou literal. Somente tipos primitivos podem ser inicializados.

### Comando de Atribuição

Existem duas formas de atribuição: para identificadores cujo tipo é primitivo (veja Seção 2.2), e para identificadores de tipo declarado pelo usuário (veja Seção 2.1). Identificadores de tipos primitivos simples podem receber valores assim:

```
identificador = expressão  
identificador[expressão] = expressão
```

Para os identificadores cujo tipo é aquele declarado pelo usuário pode ter seus campos acessados diretamente através do operador `$`, assim:

```
identificador$campo = expressão
```

### Comandos de Entrada e Saída

Identificado pela palavra reservada `input`, seguida de uma expressão. O comando de saída é identificado pela palavra reservada `output`, seguida de uma lista de expressões separadas por vírgulas.

### Chamada de Função

Uma chamada de função consiste no nome da função, seguida de argumentos entre parênteses separados por vírgula. Um argumento pode ser uma expressão.

### Comandos de Shift

Sendo número um literal inteiro positivo, temos:

```
identificador << numero  
identificador >> numero
```

### Comando de Retorno, Break, Continue e Case

Retorno é a palavra reservada `return` seguida de uma expressão. Os comandos `break` e `continue` são simples. O comando `case` é o único que não termina por ponto-e-vírgula, por ser considerado um marcador de lugar. Ele é seguido de um literal inteiro, seguido enfim por dois-pontos.

### Comandos de Controle de Fluxo

POA possui construções condicionais, iterativas e de seleção para controle estruturado de fluxo. As condicionais incluem o `if` com o `else` opcional, assim:

```
if (expressão) then bloco  
if (expressão) then bloco else bloco
```

As construções iterativas são as seguintes no formato:

```
foreach (identificador: lista) bloco  
for (lista: expressão: lista) bloco  
while (expressão) do bloco  
do bloco while (expressão)
```

A lista do `foreach` é uma lista de expressões separadas por vírgula. Os dois marcadores `lista` do comando `for` são listas de comandos separados por vírgula. A única construção de seleção é o `switch-case`, seguindo o seguinte padrão:

```
switch (expressão) bloco
```

Em todas as construções de controle de fluxo, o termo `bloco` indica um bloco de comandos (veja Seção 2.4).

## 2.6 Expressões Aritméticas e Lógicas

As expressões aritméticas podem ter como operandos: (a) identificadores, opcionalmente seguidos de expressão inteira entre colchetes, para acesso a vetores; (b) literais numéricos como inteiro e ponto-flutuante; (c) chamada de função. As expressões aritméticas podem ser formadas recursivamente com operadores aritméticos, assim como permitem o uso de parênteses para associatividade.

Expressões lógicas podem ser formadas através dos operadores relacionais aplicados a expressões aritméticas, ou de operadores lógicos aplicados a expressões lógicas, recursivamente. Outras expressões podem ser formadas considerando variáveis lógicas do tipo `bool`. Nesta etapa do trabalho, porém, não haverá distinção alguma entre expressões aritméticas, inteiras, de caracteres ou lógicas. A descrição sintática deve aceitar qualquer operadores e subexpressão de um desses tipos como válidos, deixando para a análise semântica das próximas etapas do projeto a tarefa de verificar a validade dos operandos e operadores.

## 3 Casos omissos

Casos não previstos serão discutidos com o professor. Abaixo os casos omissos já detectados e cujo interpretação já foi definida.

### 3.1 Quantidade mínima de campos

Uma declaração de novo tipo não pode ser vazia.

### 3.2 Valor inicial para variáveis globais

Variáveis globais não recebem valores no momento da sua declaração, ao contrário das variáveis locais conforme restrições da especificação.

### 3.3 Vetores em novos tipos

Novos tipos podem ter campos que são vetores de tipos primitivos.