

## Trabalho Prático - Especificação da Etapa 2: Análise Sintática e Preenchimento da Tabela de Símbolos

### Resumo:

O trabalho consiste na implementação de um compilador para a linguagem que chamaremos a partir de agora de **lang162**. Na segunda etapa do trabalho é preciso fazer um analisador sintático utilizando a ferramenta de geração de reconhecedores *yacc* (ou *bison*) e completar o preenchimento da tabela de símbolos encontrados, associando os valores e tipos corretos aos *tokens*.

### Funcionalidades necessárias:

A sua análise sintática deve fazer as seguintes tarefas:

- o programa principal deve chamar uma única vez a rotina *yyparse* para reconhecer programas que fazem parte da linguagem, e se concluída com sucesso, a análise deve retornar o valor 0 com *exit(0)*;
- imprimir uma mensagem de erro sintático para os programas não reconhecidos, informando a linha onde o erro ocorreu, e retornar o valor 3 como código genérico de erro sintático, chamando *exit(3)*;
- os nodos armazenados na tabela hash devem distinguir entre os tipos de símbolos armazenados, e o nodo deve ser associado ao token retornado através da atribuição para *yylval.symbol*;

### Descrição Geral da Linguagem

Um programa na linguagem **lang162** é composto por um conjunto de declarações de funções ou declarações de variáveis, globais, que podem aparecer em qualquer ordem. Todas as declarações globais devem ser terminadas por ponto e vírgula. Cada função é descrita por um cabeçalho seguido de seu corpo, sendo que o corpo da função é um bloco, como definido adiante. Os comandos podem ser de atribuição, controle de fluxo ou os comandos *read*, *print* e *return*. Um bloco também é considerado sintaticamente como um comando, podendo aparecer no lugar de qualquer comando.

### Declarações de variáveis globais

As variáveis são declaradas pela sequência de seu tipo, seu nome, dois-pontos e o valor de inicialização, que é obrigatório. A linguagem inclui também a declaração de vetores, feita pela definição de seu tamanho inteiro positivo entre colchetes, colocada imediatamente à direita do nome. No caso dos vetores, a inicialização é opcional, e

quando presente, será dada pela sequência de valores literais separados por caracteres em branco, entre o sinal de dois-pontos e o terminador ponto-e-vírgula. Se não estiver presente, o terminador ponto-e-vírgula segue imediatamente o tamanho entre colchetes, não havendo o caracter de dois-pontos. Variáveis e vetores podem ser dos tipos `int`, `float`, `char` ou `bool`. Os valores de inicialização podem ser literais inteiros, literais de caracteres entre aspas simples, ou os valores booleanos `TRUE` e `FALSE`.

### Definição de funções

Cada função é definida por seu cabeçalho seguido de seu corpo. O cabeçalho consiste no tipo do valor de retorno, seguido do nome da função e depois de uma lista, possivelmente vazia, entre parênteses, de parâmetros de entrada, separados por vírgula, onde cada parâmetro é definido por tipo e nome, e não podem ser do tipo vetor. O corpo da função é composto por um bloco, como definido a seguir.

### Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência, que não pode ser vazia, de comandos, **cada um deles terminado por ponto-e-vírgula**. O símbolo de ponto-e-vírgula, por definição, está associado à lista de comandos presentes dentro de um bloco, e não a cada comando em si. Considerando que os comandos poderão aparecer recursivamente dentro de outros comandos, não é necessário e nem permitido que cada um deles venha acompanhado de um ponto-e-vírgula adicional, mas apenas quando ocorrem na formação da lista dentro do bloco. Um bloco de comandos é considerado como um comando único simples, recursivamente, e pode ser utilizado em qualquer construção que aceite um comando simples.

### Comandos simples

Os comandos da linguagem podem ser: atribuição, construções de controle de fluxo, *read*, *print*, *return* e **comando vazio**. O comando vazio segue as mesmas regras dos demais, sendo terminado por ponto-e-vírgula quando está na lista do bloco, ou não quando estiver dentro de um comando recursivo. Na atribuição usa-se uma das seguintes formas:

```
variável = expressão  
vetor[ expressão ] = expressão
```

Os tipos corretos para o assinalamento e para o índice serão verificados somente na análise semântica. O comando *read* é identificado pela palavra reservada `read`, seguida de variável, na qual o valor lido da entrada padrão, se disponível e compatível, será colocado. Somente variáveis escalares são aceitas no comando `input`, e não vetores ou posições de vetores. O comando *print* é identificado pela palavra reservada `print`, seguida de uma lista de elementos separados por espaços, onde cada elemento pode ser um *string* ou uma expressão aritmética a ser impressa. O comando *return* é identificado pela palavra reservada `return` seguida de uma expressão que dá o valor de retorno. Os comandos de controle de fluxo são descritos adiante.

## Expressões Aritméticas e Lógicas

As expressões aritméticas têm como folhas identificadores, opcionalmente seguidos de expressão inteira entre colchetes, para acesso a posições de vetores, ou podem ser literais numéricos e ASCII. As expressões aritméticas podem ser formadas recursivamente com operadores aritméticos, assim como permitem o uso de parênteses para associatividade. Expressões Lógicas (Booleanas) podem ser formadas através dos operadores relacionais aplicados a expressões aritméticas, ou de operadores lógicos aplicados a expressões lógicas, recursivamente. Expressões também podem ser formadas considerando literais do tipo caractere. Nesta etapa do trabalho, porém, não haverá distinção alguma entre expressões aritméticas, inteiras, de caractere ou lógicas. A descrição sintática deve aceitar qualquer operador e sub-expressão de um desses tipos como válidos, deixando para a análise semântica das próximas etapas verificar a validade dos operandos e operadores. Finalmente, um operando possível de expressão é uma chamada de função, feita pelo seu nome, seguido de argumentos entre parênteses.

## Comandos de Controle de Fluxo

Para controle de fluxo, a linguagem possui as construções condicionais listadas abaixo. Para implementar a segunda versão do comando de iteração 'for', você precisará acrescentar a palavra reservada 'to' no seu analisador léxico, já que ela não estava presente na definição da etapa 1.

```
if (expressão) then comando
if (expressão) then comando else comando
for (expressão) comando
for (identificador = expressão to expressão) comando
```

## Tipos e Valores na tabela de Símbolos

A tabela de símbolos até aqui poderia representar o tipo do símbolo usando os mesmos **#defines** criados para os *tokens* (agora gerados pelo *yacc*). Mas logo será necessário fazer mais distinções, principalmente pelo tipo dos identificadores. Assim, é preferível criar códigos especiais para símbolos, através de definições como:

```
#define      SYMBOL_LIT_INT          1
#define      SYMBOL_LIT_REAL        2
...
#define      SYMBOL_IDENTIFIER      7
```

## Controle e organização do seu código fonte

O arquivo `tokens.h` usado na etapa1 não é mais necessário. Você deve seguir as demais regras especificadas na etapa1, entretanto. A função *main* escrita por você será usada sem alterações para os testes da etapa2. Você deve utilizar um *Makefile* para que seu programa seja completamente compilado com o comando *make*. Utilize o exemplo disponibilizado na página como referência. O formato de entrega será o mesmo da etapa1, e todas as regras devem ser observadas, apenas alterando o nome do arquivo executável e do arquivo .tgz para “etapa2”.

Porto Alegre, 22 de Agosto de 2016