

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 **first2023**. 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, guardando o texto e tipo dos *lexemas/tokens*.

Funcionalidades necessárias:

A sua análise sintática deve fazer as seguintes tarefas:

- o programa principal deve receber um nome de arquivo por parâmetro e chamar a rotina *yyparse* para reconhecer se o conteúdo do arquivo faz parte da linguagem. Se concluída com sucesso, a análise deve retornar o valor 0 (zero) 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 **first2023** é composto por uma lista de declarações globais, que podem ser de variáveis ou funções, em qualquer ordem, mesmo intercaladas. 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 *output* e *return*. Um bloco também é considerado sintaticamente como um comando, podendo aparecer no lugar de qualquer comando, e a linguagem também aceita o comando vazio.

Declarações de variáveis globais

Cada variável é declarada pela sequência de seu tipo, nome, o sinal de igual e valor de inicialização, que é obrigatório, e pode ser um literal inteiro ou um literal caractere, para as variáveis *char* ou *int*, ou literal real para as variáveis do tipo *real*. Você deve optar entretanto por permitir qualquer tipo de literal para qualquer tipo de variável de forma a simplificar a descrição sintática nesta etapa. Isso significa que inicializar *char* ou *int* com literal *real* ou inicializar variável *real* com qualquer literal não serão considerados erros sintáticos. Todas as

declarações de variáveis são terminadas por ponto-e-vírgula (;). 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 apenas por *espaços***, após os colchetes e antes do terminador ponto-e-vírgula. Se não estiver presente, o terminador ponto-e-vírgula segue imediatamente o tamanho do vetor. Vetores também podem ser dos tipos *char*, *int* e *real*. Note que essa definição de “separados por espaços” significa que na sintaxe não há nenhum outro *token* entre eles, a separação pelos espaços já é dada pela análise léxica, e essa separação pode ter sido dada pelos caracteres ‘espaço’, ‘tabulação’, ‘quebra de linha’, ou mesmo pelo final da formação possível de *tokens* diferentes, como nas situações que irão ocorrer em outras listas da linguagem.

Definição de funções

Cada função é definida por seu cabeçalho seguido de seu corpo. O cabeçalho consiste no tipo do valor de retorno e o nome da função, seguido de uma lista, possivelmente vazia, entre parênteses, de parâmetros de entrada, **separados por vírgula**, onde cada parâmetro é definido por seu tipo e nome, não podem ser do tipo vetor e não têm inicialização. O corpo da função é definido como um bloco. As declarações de funções **não são** terminadas por ;.

Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência de comandos, **terminados por ponto-e-vírgula**. Um bloco de comandos é considerado como um comando, recursivamente, e pode ser utilizado em qualquer lugar que aceite um comando. O bloco, entretanto, é uma excessão na lista de comandos (dentro do próprio bloco), por não exigir o ponto-e-vírgula como terminador. Observe que para o terminador ; não aparecer após a ocorrência do bloco, ele terá que ser modelado nos comandos em si, e não irá aparecer na definição nem no final do bloco **e nem no final dos comandos de controle de fluxo**, já que estes devem terminar por outro comando (aninhado), o qual, terá ; ou não (se for bloco).

Comandos simples

Os comandos da linguagem podem ser: atribuição, construções de controle de fluxo, *output*, *return*, e **comando vazio**. O comando vazio segue as mesmas regras dos demais, e deve ser **terminado por ;**. Isso significa que aparentemente pode haver um ; após um bloco, dentro de um bloco, pois isso representa que há um comando vazio após ele. 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 *output* é identificado pela palavra reservada *output*, seguida de uma lista de elementos **separados por vírgula**, onde cada elemento pode ser um *string* ou uma expressão. 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

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 literais de caractere. As expressões aritméticas podem ser formadas recursivamente com operadores aritméticos, assim como permitem o uso de parênteses para associatividade. Os operadores válidos são: `+, -, *, /, <, >, <=, >=, ==, !=, &, |, ~`, listados na etapa1. Nesta etapa, ainda não haverá verificação ou consistência entre operadores e operandos. A descrição sintática deve aceitar qualquer operador e sub-expressão de um desses tipos como válidos, deixando para a análise semântica verificar a validade dos operandos e operadores. Outra expressão possível é uma chamada de função, feita pelo seu nome, seguido de lista de argumentos entre parênteses, **separados por vírgula**, onde cada argumento é uma expressão, como definido aqui, recursivamente. Outra expressão possível é a ocorrência da palavra reservada *input*, seguida de um tipo entre parênteses, que significa que o programa pedirá um valor para o *stream* de entrada padrão do processo;

Comandos de Controle de Fluxo

Para controle de fluxo, a linguagem possui as três construções estruturadas listadas abaixo.

```
if ( expr ) comando
if ( expr ) comando else comando
if ( expr ) loop 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_INTE    1
#define    SYMBOL_LIT_CARA    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 agora será usada sem alterações para os testes da etapa2 e seguintes. Você deve utilizar um *Makefile* para que seu programa seja completamente apagado com *make clean* e compilado com o comando *make*. 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”.