UFRGS - INF01147 - Compiladores - 2025/2
Turma - Prof. Marcelo Johann

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 **2025++2**. Na segunda etapa do trabalho é preciso fazer um analisador sintático utilizando a ferramenta de geração de reconhecedores *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:

a. 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)*;
b. 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)*;
c. os nodos armazenados na tabela de símbolos devem distinguir entre os tipos de símbolos armazenados, com o campo *type* preenchido com uma série de *#define*s ou um *enum* como listado no final desta especificação;

Descrição Geral da Linguagem

Um programa na linguagem **2025++2** é composto por uma lista global de declarações que podem ser declarações de variáveis, de vetores ou de funções, em qualquer ordem, mesmo intercaladas. O corpo das funções é um bloco com uma lista de comandos, que podem ser de atribuição, controle de fluxo ou os comandos *read*, *print* e *return*. Um bloco também é considerando sintaticamente como um comando, podendo aparecer no lugar de qualquer comando, e a linguagem também aceita o comando vazio.

Declarações 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,* um literal real para as variáveis *float,* ou as palavras reservadas *true* ou *false* para booleanos. Você deve optar entretanto por permitir qualquer tipo de literal para qualquer tipo de variável de forma a simplificar a descrição sintática nesta etapa. As declarações de variáveis ou vetores globais 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 por espaços**, após um símbolo de igual 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, bool* ou *float*.

Declaração de funções

A declaração da função consiste no tipo do valor de retorno e o nome da função, seguido de uma lista, possivelmente vazia, entre parênteses, de parâmetros formais, **separados por vírgula**, onde cada parâmetro é definido por seu tipo e nome, não podem ser do tipo vetor e não têm inicialização, depois uma lista de declaração de variáveis locais (chamadas automáticas) e finalmente pelo seu corpo. O corpo da função segue imediatamente seu cabeçalho, e é definido como um bloco de comandos. As declarações de funções **não são** terminadas por ';'. As declarações de variáveis locais seguem as mesmas regras que as de variáveis globais.

Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência de comandos, normalmente 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 quando está presente no lugar de um comando 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, este terminador 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, que terminam 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, *read print*, 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 *read* é identificado pela palavra reservada *read*, seguida de um identificador. O comando *print* é identificado pela palavra reservada *print*, seguida de uma lista de um ou mais elementos a imprimir, **separados por espaços**, onde cada elemento pode ser *string* ou expressão. A 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. O comando *return* é identificado pela palavra reservada *return* seguida de uma expressão que dá o valor de retorno.

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, booleanos ou 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.

Comandos de Controle de de Fluxo

Para controle de fluxo, a linguagem possui as quatro construções estruturadas listadas abaixo. Observe que nos comandos condicionais ("if" e "if-else"), os parênteses estão especificados como parte da estrutura do comando de controle de fluxo, e são portando obrigatórios. Já nos comandos de laço ("while-do" e "do-while"), esses parênteses não são exigidos, mas ainda podem aparecer porque qualquer expressão pode ser formada com parênteses. Outro detalhe importante é que cada comando será terminando por ';' na sua definição, e isto vale também para o comando de controle de fluxo do tipo *do-while*, o qual termina com uma expressão antes do ';', mas não está presente nos outros três comandos de controle de fluxo. Isto ocorre porque estes três comandos terminam por um outro comando aninhado, que terá seu próprio terminador ';', ou então será um bloco delimitado por chaves ('{' e '}'), quando o ';' é desnecessário.

```
if ( expr ) comando
if ( expr ) comando else comando
while ( expr ) comando
```

Tipos e Valores na tabela de Símbolos

A tabela de símbolos não deve representar o tipo do símbolo usando os mesmos **#defines** criados para os *tokens* (agora gerados pelo *yacc/bison*). Será necessário fazer mais distinções, principalmente pelo tipo dos identificadores. Assim, é preferível criar códigos especiais para símbolos, através de definições como (a serem completadas pelos alunos):

```
#define     SYMBOL_LIT_INT      1
#define     SYMBOL_LIT_CHAR     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”.

Porto Alegre, 04 de Setembro de 2025