UFRGS - INF01147 - Compiladores - 2022/1
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 **ret20221**. Na segunda etapa do trabalho é preciso fazer um analisador sintático utilizando a ferramenta de geração de reconhecedores *yacc* (ou *bizon*) 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 *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 **ret20221** é composto por uma lista de declarações gobais, 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 *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 de variáveis globais

Cada variável é declarada pela sequência de seu tipo, nome, e valor de inicialização entre parênteses, que é obrigatório, e pode ser um literal inteiro ou um literal caractere, para as variáveis char ou int, ou literal float para as variáveis do tipo float. 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 *float* ou inicializar variável *float* 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 *float*. Note que essa definição de “separados por espaços” significa que na sintaxe não há nenhum outro token entre eles, e a separação pelos espaços já é dada pela análise léxica, e essa separação pode ter sido dada pelos caracateres ‘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 espaços**, 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, **separados entre sí por ponto-e-vírgula**. Observe que o separador ‘;’ é uma característica da lista de comandos de um bloco, e não dos comandos em si, que podem ocorrer aninhados recursivamente. Um bloco de comandos é considerado como um comando, recursivamente, e pode ser utilizado em qualquer lugar que aceite um comando. Considerando que o ponto-e-vírgula é um separador, ele não deve aparecer após o último comando dentro de um 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 se estiver dentro da lista de comandos de um bloco, deve estar separado do comando anterior por ‘;’. Isso significa que aparentemente pode haver um ‘;’ como último elemento visível 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, e opcionalmente indexado por uma expressão entre colchetes, representando respectivamente que o valor lido será escrito em uma variável ou em uma posição de um vetor. 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

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

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 espaços**, onde cada argumento é uma expressão, como definido aqui, recursivamente.

Comandos de Controle de de Fluxo

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

```
if ( expr ) comando
if ( expr ) comando else comando
while ( expr ) 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_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 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, 07 de Julho de 2022, Updated 12/07