UFRGS - INF01147 - Compiladores - 2021/1
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 **ere20211**. 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 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 **ere20211** é composto por uma seção de dados que deve aparecer no início, contendo a definição de todas as variáveis globais, e depois uma lista de declarações de funções. 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 *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

Todas as variáveis devem ser declaradas na seção inicial de dados, delimitada pela palavra reservada ‘data’, e entre ‘{‘ e ‘}’. Cada variável é declarada pela sequência de seu tipo, sinal de dois pontos (‘:’), nome, sinal de igual (‘=’), e o valor de inicialização, que é obrigatório, e pode ser um literal inteiro ou um literal caractere. 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 tipo, antes do sinal de dois pontos. No caso dos vetores, a inicialização é opcional, e quando presente, será dada pela sequência de valores literais separados por caracteres em branco, entre um sinal de igual ('=') após o nome e o terminador ponto-e-vírgula. Se não estiver presente, o terminador ponto-e-vírgula segue imediatamente o nome do vetor. Variáveis e vetores podem ser dos tipos *char*, *int* e *float*. Os valores de inicialização podem ser literais inteiros ou literais de caracteres entre aspas simples, independentemente do tipo da variável que está sendo declarada, e não existe forma de descrever literais em ponto flutuante.

Definição de funções

Cada função é definida por seu cabeçalho seguido de seu corpo. O cabeçalho consiste no tipo do valor de retorno, sinal de dois pontos e 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, sinal de dois pontos e nome, não podem ser do tipo vetor e não têm inicialização. O corpo da função é composto por um bloco, como definido adiante. As declarações de funções não são terminadas por ponto-e-vírgula, assim como também não deve haver ponto-e-virgula após as chaves da seção de dados.

Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência de comandos, **cada comando dessa lista terminado por ponto-e-vírgula**. Observe que o terminador ';' 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 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, *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 ser terminado por ';'.

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 *print* é identificado pela palavra reservada *print*, seguida de uma lista de elementos ***separados por vírgulas***, 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.

Note que se você quiser imprimir ou retornar o valor de uma variável ou um literal inteiro, terá que separar esses valores das palavras reservadas "*print*" ou "*return*" com algum símbolo diferente de espaço, caso contrário tudo será interpretado como um único identificador. Isso

pode ser feito com tabulação, quebra de linha, mas o mais lógico e elegante nesse caso é usar parênteses, já que qualquer expressão pode ser impressa ou retornada. Assim, observe que esses parênteses não são estritamente exigidos na sintaxe, mas na prática estarão presentes na maior parte dos exemplos. 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 literais de caractere. 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. Os operadores válidos são: `+,-,*,/,<,>,|,&,~,<=,>=,==,!=`, listados na etapa1. O operador ~ é unário e está associado ao operando à sua direita. Nesta etapa, ainda não haverá distinção alguma entre expressões aritméticas inteiras ou lógicas. 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. Finalmente, uma última opção de expressão é a palavra reservada "*read*", que retorna um valor lido como valor dessa expressão.

Comandos de Controle de de Fluxo

Para controle de fluxo, a linguagem possui as três construções estruturadas listadas abaixo, e também rótulos e saltos, como definido a seguir.

```
if ( expr ) comando
if ( expr ) comando else comando
until ( expr ) comando
```

O fluxo de execução dos programas também poderá ser controlado com rótulos e saltos. Um rótulo será considerado como um comando simples, isolado, definido apenas por um identificador, e a definição nesta forma serve pra simplificar seu processamento, já que o rótulo não está associado nem é uma parte opcional de outro comando. Note também que o tipo de identificador não é ainda verificado nessa etapa, e você não deve se preocupar com isso agora. Assim, no comando "*print (a);*" *a* é uma variável; no comando "*x = a();*" *a* é uma função; e no comando "*a;*" o identificador *a* é um rótulo (*lable*). Os saltos serão executados à partir dos rótulos para onde houver o comando "*comefrom*" seguido do nome do mesmo rótulo. Esse comando é o inverso do mais comum "*goto*". Ver em: <https://en.wikipedia.org/wiki/COMEFROM>

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 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, 18 de Agosto de 2021