UFRGS - INF01147 - Compiladores - 2021/2
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 **ere20212**. 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 **ere20212** 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 comando, 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

Cada variável é declarada pela sequência de seu tipo, nome, sinal de dois pontos (‘:’), e o valor de inicialização, que é obrigatório, e pode ser um literal inteiro ou um literal caractere, para as variáveis char ou int, ou uma fração para as variáveis do tipo float, onde a fração são apenas dois literais inteiros separados pelo operador ‘/‘. 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, antes do sinal de dois pontos ou ponto-e-vírgula. 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 (‘:') que segue os colchetes e o terminador ponto-e-vírgula. Se não estiver presente, o terminador ponto-e-vírgula segue imediatamente o tamanho do vetor. Vetores podem ser dos tipos *char*, *int* e *float*. Os valores de inicialização para vetores podem ser apenas literais inteiros ou literais de caractere entre aspas simples, independentemente do tipo da variável que está sendo declarada, e não é possível inicializar vetores do tipo ponto flutuante com frações, como é nas variáveis.

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 apenas um comando, mas pode ser um bloco, que é um comando, como definido adiante. As declarações de funções não são terminadas por ponto-e-vírgula.

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. Existe uma exceção quanto ao terminador ponto-e-vírgula para o uso de rótulos (labels), que serão “comandos" especiais. Um rótulo será definido como um comando isolado, não associado a nenhum outro, e só pode ocorrer dentro da lista de comandos de um bloco. Mas o comando-rótulo não deve ser terminado por ponto-e-vírgula, e sim por dois pontos (‘:’).

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. 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. A linguagem deste semestre não possui expressões lógicas (booleanas) , mas valores lógicos booleanos podem ser o resultado de operadores relacionais aplicados a expressões aritméticas. Os operadores válidos são: `+`, `-`, `*`, `/`, `<`, `>`, `<=`, `>=`, `==`, `!=`, listados na etapa1. Nesta etapa, ainda não haverá verificação ou consistência entre opradores 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. Finalmente, uma última opção de expressão é a palavra reservada "*read*" sozinha, 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 then comando
if expr then comando else comando
while expr comando
```

O fluxo de execução dos programas também poderá ser controlado com rótulos e saltos. Um rótulo será considerado como um comando isolado, mas especial, definido apenas por um identificador e terminado por ':', 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. O rótulo é considerado como um comando especial tanto por ter o terminador ':' quanto por somente ser permitido como uma das opções de comando dentro da lista de um bloco, e não em outros locais onde um comando simples é possível. Assim, é válido o seguinte trecho de código `{ if (a<5) {labela:}; }`, mas não é válido o trecho de código `{ if (a<5) labela:; }`. Note que também não são válidos os dois trechos `{ if (a<5) labela: }` ou `{ if (a<5) labela; }`. Todos esses três últimos trechos não são válidos porque um rótulo não pode ocorrer como comando simples aninhado dentro do comando condicional "if", independentemente dos terminadores.

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 (*label*). Os saltos não estruturados serão executados com o comando "*goto*" seguido do nome do rótulo.

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, 02 de Fevereiro de 2022