UFRGS - INF01147 - Compiladores - 2020/1 - UPDATED
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 **ere2020**. 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 **ere2020** é composto por um conjunto de declarações de funções ou declarações de variáveis, globais, que podem aparecer em qualquer ordem. Todas as declarações devem ser terminadas por ponto e vírgula. 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.

Declarações de variáveis globais

As variáveis são declaradas pela sequência de seu nome, sinal de igual, tipo, caractere ‘:’ e o valor de inicialização, que é obrigatório, e pode ser qualquer literal, exceto string. A linguagem inclui também a declaração de vetores, feita pela definição de seu tamanho inteiro positivo entre colchetes, colocada imediatamente à direita do tipo. 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 dois pontos (“:”) após o tamanho e o

terminador ponto-e-vírgula. Se não estiver presente, o terminador ponto-e-vírgula segue imediatamente o tamanho entre colchetes. Variáveis e vetores podem ser dos tipos `bool`, `char`, `int`, `float`. Os valores de inicialização podem ser literais inteiros, literais reais ou literais de caracteres entre aspas simples, TRUE ou FALSE, independentemente do tipo da variável que está sendo declarada.

Definição de funções

Cada função é definida por seu cabeçalho seguido de seu corpo. O cabeçalho consiste no nome da função e depois uma lista, possivelmente vazia, entre parênteses, de parâmetros de entrada, separados por vírgula, onde cada parâmetro é definido por seu nome, sinal de igual e tipo, não podem ser do tipo vetor e não têm inicialização, e após a lista entre parênteses, o cabeçalho ainda continua pelo sinal de igual e o tipo de valor de retorno da função. O corpo da função é composto por um bloco, como definido adiante.

Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência de comandos, **sem qualquer caractere ou símbolo especial de separação**. Assim, os comandos apenas estão separados entre si por caracteres em branco, o que não aparece na especificação sintática. 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, *read*, *print*, *return* e **comando vazio**. O comando vazio segue as mesmas regras dos demais, e como os comandos não possuem caractere especial de separação ou terminação, uma lista potencialmente infinita de comandos vazios poderia ser interpretada pelo analisador sintático. Isto deve gerar um conflito do tipo reduce/reduce na especificação do yacc, mas ele escolhe como default a regra mais curta e encerrará a lista de comandos corretamente.

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 variável, na qual o valor lido da entrada padrão, se disponível e compatível, será colocado. Somente variáveis escalares são aceitas no comando *read*, e não vetores ou posições de vetores. O comando *print* é identificado pela palavra reservada `print`, seguida de uma lista de elementos ***potencialmente 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 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 caracteres. 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: `+,-,*,/,<,>,|,^,~,<=,>=,==` e `!=`, listados na etapa1. Expressões também podem ser formadas considerando literais do tipo caractere. Nesta etapa, porém, não haverá distinção alguma entre expressões aritméticas, inteiras, de caractere 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. Finalmente, um operando 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 construções listadas abaixo.

```
if ( expr ) then comando
if ( expr ) then comando else comando
while ( expr ) comando
loop ( TK_IDENTIFIER : expr , expr , 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_REAL     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. 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 Setembro de 2020