UFRGS - INF01147 - Compiladores - 2015/1
Turma A - 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 **lang151**. 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 encontrados, associando os valores e tipos corretos aos *tokens*.

Funcionalidades necessárias:

A sua análise sintática deve fazer as seguintes tarefas:

a. o programa principal deve chamar uma única vez a rotina *yyparse* para reconhecer programas que fazem parte da linguagem, e 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 **lang151** é composto por um conjunto de declarações de variáveis globais e um conjunto de funções, que podem aparecer intercaladamente e em qualquer ordem. As declarações de variáveis globais devem ser terminadas por ponto e vírgula, enquanto que as funções não. 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 *input*, *output* e *return*.

Declarações de variáveis globais

As variáveis são declaradas pela sequência de seu tipo, seu nome, dois-pontos e o valor de inicialização, que é obrigatório. 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 caraceres em branco, entre o sinal de dois-pontos e o terminador ponto-e-vírgula. Se não estiver presente, o terminador ponto-e-vírgula segue imediatamente o tamanho entre colchetes, não havendo o caracter de dois-pontos. Variáveis e vetores podem ser dos tipos `word`, `byte` ou `bool`. A linguagem também aceita a declaração de ponteiros simples, feita pela inclusão do caractere "$" entre o tipo e o seu nome. Não podem ser declarados ponteiros para ponteiros, ponteiros para vetores ou vetores de ponteiros.

Definição de funções

Cada função é definida por seu cabeçalho seguido de uma lista de variáveis locais e de seu corpo. O cabeçalho consiste no tipo do valor de retorno, seguido do nome da função e depois de uma lista, possivelmente vazia, entre parênteses, de parâmetros de entrada, separados por vírgula, onde cada parâmetro é definido por tipo e nome, e não podem ser do tipo vetor. A lista de variáveis locais consiste em um conjunto de declarações de váriáveis no mesmo formato das variáveis globais, exceto pelo fato que não permite vetores. O corpo da função é composto por um bloco de comandos, como definido a seguir.

Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência, possivelmente vazia, de comandos simples, todos **terminados por ponto-e-vírgula**. 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, *input*, *output*, *return* e comando vazio. 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 *input* é identificado pela palavra reservada `input`, seguida de um nome de variável, na qual o valor lido da entrada padrão, se disponível e compatível, será colocado. O comando *output* é identificado pela palavra reservada `output`, seguida de uma lista de elementos separados por vírgulas, onde cada elemento pode ser um *string* ou uma expressão aritmética 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 a seguir.

Expressões Aritméticas e Lógicas

As expressões aritméticas têm como folhas identificadores, opcionalmente seguidos de expressão inteira entre colchetes, para acesso a vetores, ou podem ser literais numéricos e ascii. 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. Expressões também podem ser formadas considerando literais do tipo caracter. Nesta etapa do trabalho, 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 das próximas etapas verificar a validade dos operandos e operadores. Finalmente, um operando possível de expressão é uma chamada de função, feita pelo seu nome, seguido de argumentos entre parênteses. Também fazem parte das expressões os operadores de referenciação '&' e de de-referenciação '$', cuja finalidade é retornar o endereço de uma variável ou o valor indicado por um endereço.

Comandos de Controlde de Fluxo

Para controle de fluxo, a linguagem possui as seguintes contruções condicionais:

```
if (expressão) then comando
if (expressão) then comando else comando
loop (comando ; expressão ; comando) comando
```

Os comandos chamados recursivamente dentro dos comandos compostos da série condicional "if" e "loop" não devem ter um outro terminador adicional do tipo de ponto-e-vírgula. Isso significa que a existência de ponto-e-vírgula deve ser definida apenas na regra do bloco, para formação da lista de comandos.

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 um código especial para símbolos, através da série de definições abaixo:

```
#define     SYMBOL_UNDEFINED    0
#define     SYMBOL_LIT_INTEGER  1
#define     SYMBOL_LIT_FLOATING 2
#define     SYMBOL_LIT_TRUE     3
#define     SYMBOL_LIT_FALSE    4
#define     SYMBOL_LIT_CHAR     5
#define     SYMBOL_LIT_STRING   6
#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*. Utilize o exemplo disponibilizado na página como referência. 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”.

Atualizações e Dicas

Em caso de dúvida, consulte o professor.

Porto Alegre, 26 de Março de 2015