Projeto de Compilador
Etapa 2 : Análise Sintática

Stéfano Drimon Kurz Mór e Gabriel Bronzatti Moro

23 de Agosto de 2016

O trabalho consiste no projeto e implementação de um compilador funcional para uma linguagem de programação que a partir de agora chamaremos de linguagem Saga. Na segunda etapa do trabalho é preciso construir um analisador sintático utilizando a ferramenta de geração de reconhecedores `bison` e continuar o preenchimento da tabela de símbolos com outras informações encontradas, associando os valores corretos aos `tokens`. O analisador sintático deve portanto verificar se a sentença fornecida (o programa de entrada) faz parte da linguagem ou não.

1 Funcionalidades Necessárias

1.1 Definir a Gramática da Linguagem

A gramática da linguagem Saga deve ser definida de acordo com a descrição geral apresentada na Seção 2. As regras gramaticais devem ser incorporadas ao arquivo `parser.y`, arquivo este que serve como base para o analisador sintático.

Caso a análise sintática termine de forma correta, o programa deve retornar o valor da constante de pré-processamento identificada por `SINTATICA_SUCESSO` (cujo valor é zero). Para que isso ocorra, esse valor deve ser retornado pela função `yyparse()`, chamada pela função `main` do programa. Portanto, caso não existam erros sintáticos, o programa deve retornar essa constante no local apropriado do arquivo `parser.y`. Caso a entrada não seja reconhecida, deve-se imprimir uma mensagem de erro informando a linha do código da entrada que gerou o erro sintático e informações adicionais que auxiliem o programador que está utilizando o compilador a identificar o erro sintático identificado. Na ocasião de uma mensagem de erro, o analisador sintático deve retornar o valor da constante de pré-processamento identificada por `SINTATICA_ERRO` (cujo valor é 1). Essas constantes são utilizadas durante o processo de avaliação automática (incluindo os testes realizados com `ctest`).

1.2 Enriquecimento da Tabela de Símbolos

Uma vez que vários lexemas da entrada podem representar `tokens` de tipos diferentes, a tabela de símbolos deve ser alterada de forma que a chave de cada uma das entradas não seja mais simplesmente o lexema, mas a combinação entre

o lexema e o tipo do `token`. O tipo de um determinado `token` pode ser somente um dentre as seguintes constantes. Elas estão definidas no arquivo `main.h` do repositório e podem ser livremente utilizadas em qualquer parte do código.

```
#define SIMBOLO_LITERAL_INT    1
#define SIMBOLO_LITERAL_FLOAT  2
#define SIMBOLO_LITERAL_CHAR   3
#define SIMBOLO_LITERAL_STRING 4
#define SIMBOLO_LITERAL_BOOL   5
#define SIMBOLO_IDENTIFICADOR  6
```

O conteúdo de cada entrada na tabela de símbolos deve ter pelo menos três campos: número da linha da última ocorrência do lexema (já realizado na etapa 1 do projeto), o tipo do `token` da última ocorrência, e o valor do token convertido para o tipo apropriado (inteiro `int`, ponto-flutuante `float`, caractere `char`, booleano `bool` ou cadeia de caracteres `char*`). O segundo campo, representado pelo tipo do `token` é o mesmo utilizado na chave da entrada. O valor do token é um campo que pode assumir diferentes tipos: uma possibilidade é utilizar a construção `union` da linguagem C para conter os diferentes tipos possíveis para os símbolos. A conversão deve ser feita utilizando funções tais como `atoi`, no caso de números inteiros, e `atof`, no caso de ponto-flutuantes. Os tipos caractere e cadeia de caracteres não devem conter aspas no campo valor (e podem ser duplicados com `strdup`).

1.3 Associação de Valor ao `token` (`yylval`)

O analisador léxico é o responsável pela criação da entrada na tabela de símbolos para um determinado `token` que acaba de ser reconhecido. Nesta etapa, deve-se associar um ponteiro para a estrutura de dados que representa o conteúdo da entrada na tabela de símbolos ao `token` correspondente. Esta associação deve ser feita pelo analisador léxico (ou seja, no arquivo `scanner.l`).

Ela é realizada através do uso da variável global yylval[1] que é usada pelo `flex` para dar um "valor" ao `token`, além do identificador retornado imediatamente após o reconhecimento. Como esta variável global pode ser configurada com a diretiva `%union`, sugere-se o uso do campo `valor_simbolo_lexico` para a associação. Portanto, a associação deverá ser feita através de uma atribuição para a variável `yylval.valor_simbolo_lexico`. O tipo do `valor_simbolo_lexico` deve ser um ponteiro para uma entrada na tabela de símbolos.

1.4 Remoção de Conflitos Gramaticais

Deve-se realizar a remoção de conflitos Reduce/Reduce[2] e Shift/Reduce[3] de todas as regras gramaticais. Estes conflitos devem ser tratados através do uso de configurações para o bison (veja a documentação sobre `%left`, `%right` ou `%nonassoc`). Os mesmos podem ser observados através de uma análise cuidadosa do arquivo `parser.output` gerado automaticamente quando compilado. Notem que a remoção de conflitos pode ser feita, em alguns casos, somente através da re-escrita da gramática.

[1] http://www.gnu.org/software/bison/manual/html_node/Token-Values.html
[2] http://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html
[3] http://www.gnu.org/software/bison/manual/html_node/Shift_002fReduce.html

1.5 Listar o Conteúdo Tabela de Símbolos

Implementar a função `comp_print_table`, em `cc_misc.c`, para listar todas as entradas da tabela de símbolos. Utilize a função `void cc_dict_etapa_2_print_entrada (char *key, int line, int tipo)` para imprimir uma entrada. Esta função será utilizada na avaliação automática para averiguar se a solução insere somente os tokens que devem ser inseridos na tabela de símbolos.

2 A Linguagem Saga

Um programa na linguagem SAGA é composto por três elementos (todos são opcionais): um conjunto de declarações de variáveis globais, um conjunto de declarações de novos tipos, um conjunto de funções. Esses elementos podem aparecer intercaladamente e em qualquer ordem.

2.1 Declarações de Novos Tipos

Novos tipos podem ser declarados no escopo global em SAGA através da palavra reservada `class`, seguida de um nome e enfim uma lista de campos fornecida entre colchetes onde os campos são separados por dois pontos (através do caractere especial ':'). Cada campo tem o encapsulamento, o tipo e um identificador do campo. Existem três tipos de encapsulamento, identificados pelas palavras reservadas: `protected`, `private`, e `public`. Declarações de novos tipos são terminadas por ponto-e-vírgula.

2.2 Declarações de Variáveis Globais

As variáveis são declaradas pelo seu tipo, seguidas pelo seu nome. O tipo pode estar precedido opcionalmente pela palavra reservada `static`. A linguagem inclui também a declaração de vetores, feita pela definição de seu tamanho inteiro positivo entre colchetes, colocada à direita do nome, ou seja, ao final da declaração. Variáveis podem ser dos tipos primitivos `int`, `float`, `char`, `bool` e `string`; e também podem ser dos tipos declarados pelo usuário. Neste último caso, o nome do tipo é aquele utilizado depois da palavra reservada `class` quando este foi declarado. As declarações de variáveis globais são terminadas por ponto-e-vírgula.

2.3 Definição de Funções

Cada função é definida por um cabeçalho e um corpo. O cabeçalho consiste no tipo do valor de retorno, seguido pelo nome da função e terminado por uma lista. O tipo pode estar precedido opcionalmente pela palavra reservada `static`. A lista é dada entre parênteses e é composta por zero ou mais parâmetros de entrada, separados por vírgula. Cada parâmetro é definido pelo seu tipo e nome, e não pode ser do tipo vetor. O tipo de um parâmetro pode ser opcionalmente precedido da palavra reservada `const`. O corpo da função é um bloco de comandos. As funções não são terminadas por ponto-e-vírgula.

2.4 Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência, possivelmente vazia, de comandos simples, cada um **terminado** 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.

2.5 Comandos Simples

Os comandos simples da linguagem podem ser: declaração de variável local, atribuição, construções de fluxo de controle, operações de entrada, de saída, e de retorno, um bloco de comandos, chamadas de função, e o comando vazio. O comando vazio só aparece em um bloco de comandos.

Declaração de Variável

Consiste no tipo da variável precedido opcionalmente pela palavra reservada `static`, e o nome da variável. Os tipos podem ser aqueles descritos na Seção 2.2. As declarações locais, ao contrário das globais, não permitem vetores e podem permitir o uso da palavra reservada `const` antes do tipo (após a palavra reservada `static` caso esta aparecer). Uma variável local pode ser opcionalmente inicializada com um valor válido caso sua declaração seja seguida do operador composto "`<=`" e de um identificador ou literal. Somente tipos primitivos podem ser inicializados.

Comando de Atribuição

Existem duas formas de atribuição: para identificadores cujo tipo é primitivo (veja Seção 2.2), e para identificadores de tipo declarado pelo usuário (veja Seção 2.1). Identificadores de tipos primitivos simples podem receber valores assim:

```
identificador = expressão
identificador[expressão] = expressão
```

Para os identificadores cujo tipo é aquele declarado pelo usuário pode ter seus campos acessados diretamente através do operador `!`, assim:

```
identificador!campo = expressão
```

Comandos de Entrada e Saída

Identificado pela palavra reservada `input`, seguida de uma expressão. O comando de saída é identificado pela palavra reservada `output`, seguida de uma lista de expressões separadas por vírgulas.

Chamada de Função

Uma chamada de função é um comando, consiste no nome da função, seguida de argumentos entre parênteses separados por vírgula. Um argumento pode ser uma expressão.

Comandos de Shift

Sendo número um literal inteiro positivo, temos as formas:

```
identificador << numero
identificador >> numero
```

Comando de Retorno, Break, Continue e Case

Retorno é a palavra reservada `return` seguida de uma expressão. Os comandos `break` e `continue` são simples. O comando `case` é o único que não termina por ponto-e-vírgula, por ser considerado um marcador de lugar. Ele é seguido de um literal inteiro, seguindo enfim por dois-pontos.

Comandos de Controle de Fluxo

Saga possui construções condicionais, iterativas e de seleção para controle estruturado de fluxo. As condicionais incluem o `if` com o `else` opcional, da seguinte forma:

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

As construções iterativas incluem, o `foreach`, `for`, `while do` e `do while` nos seguintes formatos:

```
foreach (identificador: lista) comando
for (lista: expressão: lista) comando
while (expressão) do comando
do comando while (expressão)
```

A `lista` do `foreach` é uma lista de expressões separadas por vírgula. Os dois marcadores `lista` do comando `for` são listas de comandos separados por vírgula. A única construção de seleção é o `switch-case`, seguindo o seguinte padrão:

```
switch (expressão) comando
```

2.6 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. As expressões aritméticas podem ser formadas recursivamente com operadores aritméticos, assim como permitem o uso de parênteses para associatividade. Expressões lógicas podem ser formadas através dos operadores relacionais aplicados a expressões aritméticas, ou de operadores lógicos aplicados a expressões lógicas, recursivamente. Outras expressões podem ser formadas considerando variáveis lógicas do tipo `bool`. Nesta etapa do trabalho, porém, não haverá distinção alguma entre expressões aritméticas, inteiras, de caracteres ou lógicas. A descrição sintática deve aceitar qualquer operadores e subexpressão de um desses tipos como válidos, deixando para a análise semântica das próximas etapas do projeto a tarefa de verificar a validade dos operandos e operadores. Finalmente, um operando possível de expressão é uma chamada de função.

3 Casos Omissos

Casos não previstos serão discutidos com o professor. Abaixo os casos omissos já detectados e cujo interpretação já foi definida.

3.1 Caso Omisso #1

O tipo de um campo que faz parte de uma declaração de novo tipo **não** pode ser um tipo de usuário, ou seja, um tipo que foi declarado com `class`.

4 Regras Gerais

Veja no moodle as regras gerais de entrega.