Projeto de Compilador
E2 de **Análise Sintática**

Prof. Lucas Mello Schnorr
schnorr@inf.ufrgs.br

1 Introdução

A segunda etapa consiste em construir um analisador sintático utilizando a ferramenta de geração de reconhecedores `bison`. O arquivo `tokens.h` da etapa anterior desaparece, e deve ser substituído pelo arquivo `parser.y` (fornecido, mas que deve ser modificado para atender a esta especificação) com a declaração dos tokens. A função principal deve estar em um arquivo `main.c`, separado do arquivo `scanner.l` (léxico, da etapa 1) e do `parser.y` (sintático, por codificar na etapa 2). A solução desta etapa deve ser composta de arquivos tais como `scanner.l`, `parser.y`, e outros arquivos fontes que o grupo achar pertinente (deve ser compilados usando o `Makefile` que deve executar `flex` e `bison`). No final desta etapa o analisador sintático gerado pelo bison a partir da gramática deve verificar se a sentença fornecida – o programa de entrada a ser compilado – faz parte da linguagem ou não.

2 Funcionalidades Necessárias

2.1 Definir a gramática da linguagem

A gramática da linguagem deve ser definida a partir da descrição geral da Seção "A Linguagem", abaixo. As regras gramaticais devem ser incorporadas ao arquivo `parser.y`, arquivo este que conterá a gramática usando a sintaxe do bison.

2.2 Relatório de Erro Sintático

Caso a análise sintática termine de forma correta, o programa deve retornar zero. Na ausência de erros sintáticos, esse valor é automaticamente retornado pelo bison através da função `yyparse()`, chamada pela função `main` do programa. 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 um valor diferente de zero. Esses valores – zero ou diferente de zero – são utilizados durante a avaliação automática. Não há necessidade de implementar recuperação de erros, o compilador deve parar ao encontrar o primeiro erro sintático.

2.3 Remoção de conflitos gramaticais

Deve-se realizar a remoção de conflitos Reduce/Reduce e Shift/Reduce de todas as regras gramaticais. Estes conflitos podem ser tratados de duas formas. Reescrevendo a gramática de maneira a evitar os conflitos (recomendada) ou através do uso de configurações para o bison (veja a documentação sobre `%left`, `%right` ou `%nonassoc`). Os conflitos podem ser compreendidos através de uma análise cuidadosa do arquivo `parser.output` gerado automaticamente quando o bison é compilado com a opção `--report-file`. Sugere-se fortemente um processo construtivo da especificação em passos, verificando em cada passo a inexistência de conflitos. Por vezes, a remoção de conflitos pode ser feita somente através de uma revisão mais profunda de partes da gramática.

3 A Linguagem

Um programa na linguagem é composto por dois elementos, todos opcionais: um conjunto de declarações de variáveis globais e um conjunto de funções. Esses elementos podem aparecer intercaladamente e em qualquer ordem.

3.1 Declarações de Variáveis Globais

As variáveis são declaradas pelo tipo seguido de uma lista composta de pelo menos um nome de variável (identificador) separadas por vírgula. 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. Variáveis podem ser dos tipos primitivos `int`, `float`, `char`, `bool` e `string`. As declarações de variáveis globais são terminadas por ponto-e-vírgula, e não podem receber valores de inicialização.

3.2 Definição de Funções

Cada função é definida por um cabeçalho e um corpo, sendo que esta definição não é terminada por ponto-e-vírgula. O cabeçalho consiste no tipo do valor de retorno, que não pode ser vetor, 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.

3.3 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.

3.4 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, e chamadas de função.

Declaração de Variável

Consiste no tipo da variável precedido opcionalmente pela palavra reservada `static`, seguido de uma lista composta de pelo menos um nome de variável (identificador) separadas por vírgula. Os tipos podem ser aqueles descritos na seção sobre variáveis globais. 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.

Comando de Atribuição

Existe apenas uma forma de atribuição para identificadores. Identificadores podem receber valores assim (primeiro caso de um identificador simples; segundo caso de um identificador que é um vetor):

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

Comandos de Entrada e Saída

Para entrada de dados, o comando tem a palavra reservada `input`, seguida de um identificador. O comando de saída é identificado pela palavra reservada `output`, seguida de um identificador ou de um literal.

Chamada de Função

Uma chamada de função 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 os exemplos válidos abaixo. Os exemplos são dados com <<, mas as entradas são sintaticamente válidas também para >>. O `número` deve ser representado por um literal inteiro.

```
identificador << número
identificador[expressão] << número
```

Comando de Retorno, Break, Continue

Retorno é a palavra reservada `return` seguida de uma expressão. Os comandos `break` e `continue` são simples.

Comandos de Controle de Fluxo

A linguagem possui construções condicionais, iterativas e de seleção para controle estruturado de fluxo. As condicionais incluem o `if` com o `else` opcional, assim:

```
if (expressão) bloco
if (expressão) bloco else bloco
```

As construções iterativas são as seguintes no formato:

```
for (atrib: expressão: atrib) bloco
while (expressão) do bloco
```

Os dois marcadores `atrib` do comando `for` representa o comando de atribuição, único aceito nestas posições. Em todas as construções de controle de fluxo, o termo `bloco` indica um bloco de comandos. Este não tem ponto-e-vírgula nestas situações.

3.5 Expr. Aritméticas, Lógicas

As expressões podem ser de dois tipos: aritméticas e lógicas. As expressões aritméticas podem ter como operandos: (a) identificadores, opcionalmente seguidos de expressão inteira entre colchetes, para acesso a vetores; (b) literais numéricos como inteiro e ponto-flutuante; (c) chamada de função. As expressões aritméticas podem ser formadas recursivamente com operadores aritméticos, assim como permitem o uso de parênteses para forçar uma associatividade ou precedência diferente daquela tradicional. A associatividade é à esquerda.

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`. 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.

Os operadores são os seguintes:

- Unários (todos prefixados)
  - `+` sinal positivo explícito
  - `-` inverte o sinal
  - `!` negação lógica
  - `&` acesso ao endereço da variável
  - `*` acesso ao valor do ponteiro
  - `?` avalia uma expressão para `true` ou `false`
  - `#` acesso a um identificador como uma tabela hash
- Binários
  - `+` soma
  - `-` subtração
  - `*` multiplicação
  - `/` divisão
  - `%` resto da divisão inteira
  - `|` bitwise OR
  - `&` bitwise AND
  - `^` exponenciação
  - todos os comparadores relacionais
  - todos os operadores lógicos (`&&` para o e lógico, `||` para o ou lógico)
- Ternários
  - `?` seguido de `:`, conforme a sintaxe `expressão ? expressão : expressão`

As regras de associatividade e precedência de operadores matemáticos são aquelas tradicionais de linguagem de programação e da matemática. Recomenda-se que tais regras sejam já incorporadas na solução desta etapa, ou através de construções gramaticais ou através de comandos do bison específicos para isso (`%left`, `%right`). A solução via construções gramaticais é recomendada. Enfim, nos casos não cobertos por esta regra geral, temos as seguintes regras de associatividade:

- Associativos à direita
  - `&`, `*` (acesso ao valor do ponteiro), `#`

A Arquivo `main.c`

```
/*
Função principal para realização da análise sintática.

Este arquivo será posterioremente substituído, não acrescente nada.
*/
#include <stdio.h>
#include "parser.tab.h" //arquivo gerado com bison -d parser.y
extern int yylex_destroy(void);

int main (int argc, char **argv)
{
  int ret = yyparse();
  yylex_destroy();
  return ret;
}
```

B Arquivo `parser.y` inicial

```
%{
int yylex(void);
void yyerror (char const *s);
%}

%token TK_PR_INT
%token TK_PR_FLOAT
%token TK_PR_BOOL
%token TK_PR_CHAR
%token TK_PR_STRING
%token TK_PR_IF
%token TK_PR_THEN
%token TK_PR_ELSE
%token TK_PR_WHILE
%token TK_PR_DO
%token TK_PR_INPUT
%token TK_PR_OUTPUT
%token TK_PR_RETURN
%token TK_PR_CONST
%token TK_PR_STATIC
%token TK_PR_FOREACH
%token TK_PR_FOR
%token TK_PR_SWITCH
%token TK_PR_CASE
%token TK_PR_BREAK
%token TK_PR_CONTINUE
%token TK_PR_CLASS
%token TK_PR_PRIVATE
%token TK_PR_PUBLIC
%token TK_PR_PROTECTED
%token TK_PR_END
%token TK_PR_DEFAULT
%token TK_OC_LE
%token TK_OC_GE
%token TK_OC_EQ
%token TK_OC_NE
%token TK_OC_AND
%token TK_OC_OR
%token TK_OC_SL
%token TK_OC_SR
%token TK_LIT_INT
%token TK_LIT_FLOAT
%token TK_LIT_FALSE
%token TK_LIT_TRUE
%token TK_LIT_CHAR
```

```
%token TK_LIT_STRING
%token TK_IDENTIFICADOR
%token TOKEN_ERRO

%%

programa:

%%
```