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 fazer parte do `parser.y`, arquivo este que conterá a gramática usando a sintaxe do bison.

2.2 Relatório de Erro Sintático

Se a análise sintática tem sucesso, o programa retorna zero. Esse é o comportamento padrão da função `yyparse()`, chamada pela função `main` do programa. Caso contrário, deve-se retornar um valor diferente de zero e 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. 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 estar intercalados e em qualquer ordem.

3.1 Declarações de Variáveis Globais

As variáveis globais são declaradas pelo tipo seguido de uma lista composta de pelo menos um nome de variável (identificador) separadas por vírgula. O tipo pode ser `int`, `float`, `bool` e `char`. Um nome de variável pode ser considerado um arranjo multidimensional, caso o nome seja sucedido do caractere ~~dois-pontos~~ abre-colchetes [e de uma sequência de literais inteiros, separados pelo caractere circunflexo, e terminados por fecha-colchetes]. ~~Estas declarações~~ As declarações de variáveis globais são terminadas por ponto-e-vírgula.

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 seguido pelo nome da função e terminado por uma lista. O tipo pode ser `int`, `float`, `bool` e `char`. 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. 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, operação de retorno, um bloco de comandos, e chamadas de função.

Declaração de Variável: Consiste no tipo da variável 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. Uma variável local pode ser opcionalmente inicializada caso sua declaração seja seguida do operador composto `<=` e de um literal.

Comando de Atribuição: O comando de atribuição consiste em um identificador seguido pelo caractere de igualdade seguido por uma expressão. Um identificador pode ser um arranjo multidimensional. Neste caso, os índices são especificados, após o identificador, através de uma lista de expressões, iniciada com abre-colchetes [e terminada com fecha-colchetes], ~~onde a lista está separada do nome do arranjo por um caractere dois-pontos~~ e cada elemento da lista é separado pelo caractere circunflexo.

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.

Comando de Retorno: Trata-se do token `return` seguido de uma expressão.

Comandos de Controle de Fluxo: A linguagem possui construções condicionais, iterativas e de seleção para controle estruturado de fluxo. As condicionais incluem o token `if` seguido de uma expressão entre parênteses, seguido pelo token `then` seguido então por um bloco de comandos obrigatório. O `else`, sendo opcional, é seguido de um bloco de comandos, obrigatório caso o else seja empregado. Temos apenas uma construção de repetição que é o token `while` seguido de uma expressão entre parênteses e de um bloco de comandos.

3.5 Expressões

Expressões tem operandos e operadores. Os **operandos** podem ser (a) identificadores, opcionalmente seguidos de um abre-colchetes [com uma lista de expressões separadas por circunflexo e por fim fecha-colchetes] ~~opcionalmente seguidos de dois-pontos e de uma lista de expressões separadas por circunflexo~~; (b) literais; (c) chamada de função. As expressões podem ser formadas recursivamente pelo emprego de operadores. Elas também permitem o uso de parênteses para forçar uma associatividade ou precedência diferente daquela tradicional. A associatividade é à esquerda (use recursão à esquerda nas regras gramaticais). Os **operadores** são os seguintes:

- Unários prefixados
  - `-` inverte o sinal
  - `!` negação lógica
- Binários
  - `+` soma
  - `-` subtração
  - `*` multiplicação
  - `/` divisão
  - `%` resto da divisão inteira
  - operadores compostos

As regras de associatividade e precedência de operadores matemáticos são aquelas tradicionais de linguagem de programação e da matemática. Pode-se usar esta referência da Linguagem C. Para facilitar. Recomenda-se fortemente que tais regras sejam incorporadas na solução desta etapa, através de construções gramaticais, evitando as diretivas `%left` `%right` como mencionando no manual do bison.

Tabela de precedência.

Precedência	Op.
1	`-`
	`!`
2	`*`
	`/`
	`%`
3	`+`
	`-`
4	`<`
	`>`
	`TK_OC_LE`
	`TK_OC_GE`
5	`==`
	`!=`
6	`TK_OC_AND`
7	`TK_OC_OR`