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 (devem 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 valor zero é retornado pela função `yyparse()`. Caso contrário, esta função retorna um valor diferente de zero (tipicamente um). O valor de retorno da função `yyparse()` deve ser utilizado como valor de retorno da função principal, de maneira que seja possível identificar externamente o resultado da análise sintática. Caso o valor que será retornado seja diferente de zero, o programa deve 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. Isso é realizado na implementação da função `yyerror()` que deve ser realizada pelo grupo. Não encerre o programa de maneira não estruturada (evite chamar `exit`). O compilador deve parar ao encontrar o primeiro erro sintático. Use `%define parse.error verbose` no cabeçalho do arquivo `parser.y` para obter uma mensagem de erro com mais detalhamento.

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 devem ser resolvidos através da reescrita da gramática de maneira a evitá-los. Os conflitos podem ser melhor identificados e compreendidos através de uma análise cuidadosa do arquivo `parser.output` gerado automaticamente quando o bison é executado 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 exige uma revisão mais profunda da gramática construída a partir da especificação abaixo. A linguagem descrita abaixo não é ambígua.

3 A Linguagem

Um programa na linguagem é composto por uma lista opcional de elementos. Os elementos da lista são separados pelo operador vírgula e a lista é terminada pelo operador ponto-e-vírgula. Cada elemento dessa lista é ou uma definição de função ou uma declaração de variável.

Definição de Função: Ela possui um cabeçalho e um corpo. O cabeçalho consiste no token `TK_ID` seguido do token `TK_SETA` seguido ou do token `TK_DECIMAL` ou do token `TK_INTEIRO`, seguido por uma lista opcional de parâmetros seguido do token `TK_ATRIB`. A lista de parâmetros, quando presente, consiste no token opcional `TK_COM` seguido de uma lista, separada por vírgula, de parâmetros. Cada parâmetro consiste no token `TK_ID` seguido do token `TK_ATRIB` seguido ou do token `TK_INTEIRO` ou do token `TK_DECIMAL`. O corpo de uma função é um bloco de comandos (veja abaixo).

Declaração de variável: Esta declaração é idêntica ao comando simples de declaração de variável

(veja abaixo), sendo que a única e importante diferença é que esse elemento não pode receber valores de inicialização.

3.1 Comandos Simples

Os comandos simples da linguagem podem ser: bloco de comandos, declaração de variável, comando de atribuição, chamada de função, comando de retorno, e construções de fluxo de controle.

Bloco de Comandos: Definido entre colchetes, e consiste em uma sequência, possivelmente vazia, de comandos simples. Um bloco de comandos é considerado como um comando único simples e pode ser utilizado em qualquer construção que aceite um comando simples.

Declaração de Variável: Consiste no token `TK_VAR` seguido do token `TK_ID`, que é por sua vez seguido do token `TK_ATRIB` e enfim seguido do tipo. O tipo pode ser ou o token `TK_DECIMAL` ou o token `TK_INTEIRO`. Uma variável pode ser opcionalmente inicializada caso sua declaração seja seguida do token `TK_COM` e de um literal. Um literal pode ser ou o token `TK_LI_INTEIRO` ou o token `TK_LI_DECIMAL`.

Comando de Atribuição: O comando de atribuição consiste em um token `TK_ID`, seguido do token `TK_ATRIB` e enfim seguido por uma expressão.

Chamada de Função: Uma chamada de função consiste no token `TK_ID`, seguida de argumentos entre parênteses, sendo que cada argumento é separado do outro por vírgula. Um argumento é uma expressão. Uma chamada de função pode existir sem argumentos.

Comando de Retorno: Trata-se do token `TK_RETORNA` seguido de uma expressão, seguido do token `TK_ATRIB` e terminado ou pelo token `TK_DECIMAL` ou pelo token `TK_INTEIRO`.

Comandos de Controle de Fluxo: A linguagem possui uma construção condicional e uma construção iterativa para controle estruturado de fluxo. A condicional consiste no token `TK_SE` seguido de uma expressão entre parênteses e então por um bloco de comandos obrigatório. Após este bloco, podemos opcionalmente ter o token `TK_SENAO` que, quando aparece, é seguido obrigatoriamente por um bloco de comandos. Temos apenas uma construção de repetição que é o token `TK_ENQUANTO` seguido de uma expressão entre parênteses e de um bloco de comandos.

3.2 Expressão

Expressões envolvem operandos e operadores, sendo este opcional. Os **operandos** podem ser identificadores, literais e chamada de função ou outras expressões, podendo portanto 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 (portanto implemente recursão à esquerda nas regras gramaticais). Os **operadores** são os seguintes:

- Unários prefixados
  - `+` número positivo
  - `-` inverte o sinal
  - `!` negação lógica
- Binários infixados
  - `+` 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, abaixo temos uma tabela com uma visão somente com os operadores de nossa linguagem. Recomenda-se fortemente que tais regras sejam incorporadas na solução desta etapa através de construções gramaticais (evitando totalmente o emprego das diretivas `%left` `%right` do bison).

Precedência	Op.	Aridade
0	Chamada de Função	
	`TK_ID`	
	`TK_LI_INTEIRO`	
	`TK_LI_DECIMAL`	
	`(` expressão `)`	
1	`+`	Unária
	`-`	Unária
	`!`	Unária
2	`*`	Binária
	`/`	Binária
	`%`	Binária
3	`+`	Binária
	`-`	Binária
4	`<`	Binária
	`>`	Binária
	`TK_OC_LE`	Binária
	`TK_OC_GE`	Binária
5	`TK_OC_EQ`	Binária
	`TK_OC_NE`	Binária
6	`&`	Binária
7	`|`	Binária