

## Especificação da Etapa 2 do Projeto de Compilador

# Análise Sintática e Preenchimento da Tabela de Símbolos

O trabalho consiste no projeto e implementação de um compilador funcional para uma linguagem de programação que a partir de agora chamaremos de **Linguagem IKS**<sup>1</sup>. Na segunda etapa do trabalho é preciso fazer um analisador sintático utilizando a ferramenta de geração de reconhecedores **bison** e completar o preenchimento da tabela de símbolos encontrados, associando os valores e tipos corretos aos *tokens*.

### 1 Funcionalidades Necessárias

A sua análise sintática deve realizar as seguintes tarefas:

1. **Realização da análise sintática.** O programa principal deve chamar uma única vez a rotina `yyparse` para reconhecer programas que fazem parte da linguagem. Se concluída com sucesso sem nenhum erro sintático, o analisador deve ser terminado com a chamada `exit(IKS_SYNTAX_SUCESSO);`.
2. **Relatório de erros sintáticos.** Caso a entrada não seja reconhecida, deve-se imprimir uma mensagem informando a linha do código da entrada que gerou o erro sintático. Após a emissão desta mensagem de erro, o analisador sintático deve ser terminado com a chamada `exit(IKS_SYNTAX_ERRO);`.
3. **Enriquecimento da tabela de símbolos.** Os nós armazenados na tabela de símbolos devem distinguir entre os tipos de símbolos armazenados (identificadores, literais flutuantes, inteiros, etc); os valores devem ser convertidos do lexema para um campo de tipo apropriado, e o nó deve ser associado ao *token* retornado através da atribuição para `yylval.symbol`.

### 2 Descrição Geral da Linguagem IKS

Um programa na linguagem **IKS** é composto por um conjunto opcional de declarações de variáveis globais e um conjunto opcional de funções, que podem aparecer intercaladamente e em qualquer ordem. Todas as declarações globais são **terminadas** por ponto-e-vírgula. Cada função é descrita por um cabeçalho seguido do seu corpo. Os comandos podem ser de atribuição, controle de fluxo ou os comandos **input**, **output** e **return**.

#### 2.1 Declarações de Variáveis Globais

As variáveis são declaradas pelo seu tipo, dois-pontos e são terminadas pelo seu nome. 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. Todas as variáveis globais serão inicializadas com o valor zero. Variáveis podem ser dos tipos **int**, **float**, **char**, **bool** e **string**.

#### 2.2 Definição de Funções

Cada função é definida por um cabeçalho, uma lista de declarações locais e um corpo. O **cabeçalho** consiste no tipo do valor de retorno, seguido de dois-pontos, seguido pelo nome da função e terminado por uma lista. Esta lista, possivelmente vazia, entre parênteses, de parâmetros de entrada, separados por vírgula, onde cada parâmetro é definido pelo tipo, dois pontos e seu nome, e não podem ser do tipo vetor. A **lista de declarações** locais é um grupo de declarações de variáveis no mesmo formato das declarações globais, onde cada uma consiste no tipo da variável, dois-pontos, o nome da variável e o terminador ponto-e-vírgula. As declarações locais, ao contrário das globais, não permitem vetores. O **corpo** da função é composto por um bloco, como definido a seguir. *A função não deve ser terminada por ponto-e-vírgula.*

#### 2.3 Bloco de Comandos

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

<sup>1</sup>IKS é o fonema da letra X em francês

## 2.4 Comandos Simples

Os comandos simples da linguagem podem ser: 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. 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 de entrada 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 de saída é identificado pela palavra reservada **output**, seguida de uma lista de elementos separados por vírgulas, onde cada elemento pode ser uma **string** ou uma expressão aritmética a ser impressa. O comando de retorno 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. Para facilitar a escrita de programas aceitando o caractere de ponto-e-vírgula como terminador, e não apenas separador, a linguagem deve aceitar também o comando vazio.

## 2.5 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 em código ASCII. 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 do tipo **caractere**. 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, feita pelo seu nome, seguido de argumentos entre parênteses e separados por vírgula.

## 2.6 Comandos de Fluxo de Controle

Para o controle de fluxo, a linguagem **IKS** possui as seguintes construções:

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

## 3 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 **bison**). 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 IKS_SIMBOLO_INDEFINIDO      0  
#define IKS_SIMBOLO_LITERAL_INT     1  
#define IKS_SIMBOLO_LITERAL_FLOAT   2  
#define IKS_SIMBOLO_LITERAL_CHAR    3  
#define IKS_SIMBOLO_LITERAL_STRING  4  
#define IKS_SIMBOLO_LITERAL_BOOL    5  
#define IKS_SIMBOLO_IDENTIFICADOR   6
```

## 4 Controle e Organização da Solução

A função `main` deve estar em um arquivo chamado `main.c`. Outros arquivos fontes são encorajados de forma a manter a modularidade do código fonte. A entrada para o *bison* deve estar em um arquivo com o nome `parser.y`. A entrada para o *flex* deve estar em um arquivo com o nome `scanner.l`.

### 4.1 Git e Cmake

A solução desta etapa do projeto de compiladores deve ser feita sobre a etapa 0. Cada ação de commit deve vir com mensagens significativas explicando a mudança feita. Todos os membros do grupo devem ter feito ações de commit, pelo fato deste trabalho ser colaborativo. Estas duas ações – mensagens de commit e quem fez o commit – serão obtidas pelo professor através do comando `git log` na raiz do repositório solução do grupo. Os arquivos adicionais necessários para esta etapa podem ser obtidos através do seguinte comando (para atualizar o repositório já clonado na etapa 0):

```
$ git pull origin master
```

Note que o arquivo `parser.y`, que deverá ser fortemente modificado para atender aos requisitos deste trabalho, está praticamente vazio. A solução do grupo deve partir deste código inicial, juntamente com o resultado da etapa 1 do projeto de compilador do mesmo grupo. Novos arquivos de código fonte podem ser adicionados, modificando o arquivo `CMakeLists.txt`, para que ele seja incluído no processo de compilação do analisador sintático.

## 5 Atualizações e Dicas

Verifique regularmente o Moodle da disciplina e o final deste documento para informar-se de alguma eventual atualização que se faça necessária ou dicas sobre estratégias que o ajudem a resolver problemas particulares. Em caso de dúvida, não hesite em consultar o professor.