Projeto de Compilador
E3 de **Árvore Sintática Abstrata** (AST)

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

1 Introdução

A terceira etapa do projeto de compilador para a linguagem consiste na criação da árvore sintática abstrata (*Abstract Syntax Tree* – AST) baseada no programa de entrada. A árvore deve ser obrigatoriamente criada na medida que as regras semânticas são executadas no final das produções. Um ponteiro para a raiz da árvore deve ainda existir após o retorno da função `yyparse`.

2 Funcionalidades Necessárias

2.1 Associação de valor ao `token` (`yylval`)

Nesta etapa, deve-se associar um valor para os `tokens` que farão parte da AST (identificadores e literais). Devemos fazer tal associação no analisador léxico (alterações no arquivo `scanner.l`), atribuindo um valor para a variável global yylval. Esta variável deve ser configurada com a diretiva `%union` no `parser.y`. Assumindo que o campo desta union seja `valor_lexico`, a associação do valor em si deverá então ser feita através de uma atribuição para a variável `yylval.valor_lexico` no código do scanner. O tipo do campo `valor_lexico` (e por consequência o valor que será retido) deve ser uma estrutura de dados (`struct`) que contém os seguintes campos: (a) número da linha onde apareceu o lexema; (b) tipo do token (identificador ou literal); (c) valor do token. O valor do token é uma cadeia de caracteres (duplicada com `strdup` a partir de `yytext`) para todos os tipos de tokens. Somente tokens identificadores e literais devem possuir um valor léxico a ser empregado na AST.

2.2 Estrutura de dados em árvore

Implementar uma estrutura de dados para representar uma árvore em memória, com funções habituais tais como criação de nó, remoção, alteração e impressão recursiva da árvore através de um percurso em profundidade. Qualquer outra função que o grupo achar pertinente pode também ser implementada. Salienta-se o fato de que cada nó pode ter um número arbitrário de filhos, que também serão nós da árvore.

2.3 Ações *bison* para construção da AST

Colocar ações semânticas **no final das regras de produção** descritas no arquivo para o `bison`, as quais criam ou propagam os nós da árvore, montando-a na medida que a análise sintática é realizada. Como a análise sintática é ascendente, a árvore será criada de baixo para cima, no momento das reduções do *parser*. A maior parte das ações será composta de chamadas para o procedimento de criação de um nó da árvore, e associação desta com seus filhos na árvore de derivação que já foram criados. Ao final do processo de análise sintática, um ponteiro para a estrutura de dados que guarda a raiz da árvore deve ser salvo na variável global `arvore`. A raiz da árvore é o nó que representa a primeira função do arquivo de entrada. Devem fazer parte da AST:

1. Listas de funções, onde cada função tem dois filhos, um que é o seu primeiro comando e outro que é a próxima função;
2. Listas de comandos, onde cada comando tem *pelo menos* um filho, que é o próximo comando;
3. Listas de expressões, onde cada expressão tem *pelo menos* um filho, que é a próxima expressão, naqueles comandos onde isso se faz necessário, tais como na chamada de função;
4. Todos os comandos simples da linguagem, salvo o bloco de comando. O comando de atribuição deve ter pelo menos dois filhos, um que é o identificador e outro que é o valor da expressão. O comando chamada de função tem pelo menos um filho, que é a primeira expressão na lista de seus argumentos. O comando return tem um filho, que é uma expressão. O comando if com else opcional deve ter pelo menos três filhos, um para a expressão, outro para o primeiro comando quando verdade, e o último – opcional – para o segundo comando quando falso. O comando while deve ter pelo menos dois filhos, um para expressão e outro para o primeiro comando do laço.
5. Todas as expressões obedecem as regras de associatividade e precedência já estabelecidas na E2, incluindo identificadores e literais. Os operadores unários devem ter pelo menos um filho, os operadores binários devem ter pelo menos dois filhos.

Acima explicita-se o "pelo menos"pois os diversos nós da árvore podem aparecer em listas, sendo necessário mais um filho que indica qual o próximo elemento da lista, conforme detalhado acima.

2.4 Exportar a árvore em formato específico

Implementar a função `exporta` (veja no anexo `main.c` abaixo). Esta função deverá percorrer a árvore gerada, a partir da raiz e de maneira recursiva, imprimindo todos os nós (vértices) e todas as relações entre os nós (arestas). A impressão deve acontecer na saída padrão (stdout, tipicamente com uso de `printf`). Um nó deve ser identificado pelo seu endereço de memória (impresso com o padrão `%p` da libc). Esse formato de arquivo de texto puro possui dois tipos de linhas: 1/ que identificam arestas; 2/ que identificam o rótulo (*label*) do nó. Estes dois tipos podem aparecer misturadas, em qualquer ordem. Todos os nós devem possuir um nome. Não devem haver nós desconectados.

2.4.1 Identificação de arestas

Veja o exemplo de saída abaixo, onde o nó `0x8235900` tem somente um filho `0x82358e8`, que por sua vez tem dois filhos (`0x8235890` e `0x82358d0`):

```
0x8235900, 0x82358e8
0x82358e8, 0x8235890
0x82358e8, 0x82358d0
```

2.4.2 Identificação dos *labels*

Todos os nós devem ser nomeados, usando uma linha por nó, da seguinte forma: o identificador do nó (endereço de memória impresso com o padrão `%p` da lib) seguido de espaço e abre colchetes, `label=` e o nome entre aspas duplas, terminando-se por fecha colchetes e ponto-e-vírgula. Veja o exemplo:

```
0x8235900 [label="minha_funcao"];
0x82358e8 [label="="];
0x8235890 [label="minha_var"];
0x82358d0 [label="c"];
```

O nome que deve ser utilizado no campo `label` deve seguir o seguinte regramento. Para funções, deve-se utilizar seu identificador (o nome da função). Para o comando de atribuição, o nome deve ser `=` (o operador igual). Para a chamada de função, o nome deve ser `call` seguinte do nome da função chamada, separado por espaço. Para o comando de retorno deve ser utilizado o lexema correspondente. Para os comandos de controle de fluxo, deve-se utilizar o nome `if` para o comando if com else opcional, e `while` para o comando while. Para as expressões aritméticas, devem ser utilizados os próprios operadores unários ou binários como nomes. Para as expressões lógicas, deve-se utilizar `&` para o e lógico e `|` para o ou lógico. Enfim, para os identificadores e literais, utiliza-se o próprio lexema.

2.5 Remoção de conflitos/ajustes gramaticais

Todos os conflitos *Reduce-Reduce* e *Shift-Reduce* devem ser removidos, caso estes se tornem presentes com eventuais modificações feitas na gramática.

A Arquivo `main.c`

A função principal da E3 aparece abaixo. A variável global `arvore` de tipo `void*` é um ponteiro para a estrutura de dados que contém a raiz da árvore de derivação do programa. A função `exporta`, cujo protótipo é dado, deve ser implementada de maneira recursiva para exportar a AST na saída padrão.

```
#include <stdio.h>
extern int yyparse(void);
extern int yylex_destroy(void);
void *arvore = NULL;
void exporta (void *arvore);
int main (int argc, char **argv)
{
  int ret = yyparse();
  exporta (arvore);
  yylex_destroy();
  return ret;
}
```

Utilize o comando `extern void *arvore` nos outros arquivos que fazem parte da implementação (como no `parser.y`) para ter acesso a variável global `arvore` declarada no arquivo `main.c`.

B Avaliação objetiva

No processo de avaliação automática, será considerada como raiz o primeiro nó que não tenha um pai. A ordem dos filhos de um nó da árvore não importa na avaliação objetiva (mas importa em etapas subsequentes). O programa será executado da seguinte forma no processo de avaliação automática:

```
./etapa3 < entrada > saida
```

O conteúdo de `saida` contém a árvore da solução. Uma vez reconstituído, tal estrutura da solução será comparada com a AST de referência. Cada teste unitário será avaliado como correto caso a árvore criada seja estruturalmente idêntica aquela de referência, com a mesma quantidade de nós, arestas e nomes de nós (a ordem dos nós filhos não importa).