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 ser mantida em memória após o fim da análise sintática, ou seja, após a função `yyparse` retornar.

2 Funcionalidades Necessárias

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

Nesta etapa, deve-se associar um valor para os `tokens`. 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`, usando o nome `valor_lexico` no campo que será dedicado aos valores do subconjunto de tokens. A associação do valor em si deverá então ser feita através de uma atribuição para a variável `yylval.valor_lexico`.

Campo Valor Léxico: O tipo do campo `valor_lexico` (e por consequência o valor que será retido) deve ser uma estrutura de dados que contém os seguintes campos: (a) número da linha onde apareceu o lexema; (b) tipo do token (caractere especial, palavra reservada, operador composto, identificador ou literal); (c) valor do token.

Valor do Token: O valor do token deve ser uma cadeia de caracteres (duplicada com `strdup` a partir de `yytext`) para tokens do tipo caractere especial, operador composto, palavra reservada e identificador. Os tokens do tipo literal devem recever um tratamento especial, pois o *valor do token* deve ser convertido para o tipo apropriado (inteiro `int`, ponto-flutuante `float`, caractere `char`, booleano para `bool` ou `int`). A conversão é feita com funções tais como `atoi` e `atof`. O tipo de token caractere literal não deve conter aspas simples no campo valor. Uma forma usual de implementar o *valor do token* para literais é utilizar dois campos: um *tipo de literal* e o valor associado a ele através de uma construção `union` da linguagem C.

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 e a declaração de variáveis sem inicialização. O comando de inicialização de variável e de atribuição deve ter pelo menos dois filhos, um que é o identificador (ou identificador com indexação de arranjo – veja abaixo) e outro que é o valor da inicialização. 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. ~~Os comandos de break e continue não tem filhos~~. 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 primeiro 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 ariméticas e lógicas devem obedecer 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. O indexador de arranjo `[]` deve ter dois filhos, o identificador do

arranjo e uma árvore com as expressões de indexação. Esta árvore deve se inspirar de uma árvore de derivação criada com uma regra de lista de expressões com recursão à esquerda. Cada nó da árvore deve ter dois filhos, um que indica a dimensão anterior e outro para a expressão que define o índice da dimensão. O nó da árvore que indica a primeira dimensão deve ter apenas um filho, que é a expressão da primeira dimensão.

Explicita-se o "pelo menos"pois os 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). Um exemplo de saída CSV válida é o seguinte, 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
```

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 declaração de variável com inicialização, o nome deve ser <= (o operador composto menor igual). Para o comando de atribuição, o nome deve ser = (o operador igual). Para o indexador de vetor, o nome deve ser `[]` (abre e fecha colchetes) e para os nós que guardam as expressões das dimensões, o nome deve ser `^` (o sinal de circunflexo). 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, deve-se utilizar o próprio lexema, sem as aspas simples quando for um literal caractere.

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.

2.6 Gerenciar corretamente a memória

Implementar a função `libera` (veja no anexo `main.c` abaixo), que deve liberar a memória de maneira recursiva (de baixo para cima). O programa `valgrind` será utilizado para averiguar a ausência de vazamentos de memória.

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. A função `libera`, cujo protótipo também é dado, deve ser implementada para liberar toda a memória que foi alocada para manter a árvore.

```
/*
  Função principal para realização da E3.
  Não modifique este arquivo.
*/
#include <stdio.h>
extern int yyparse(void);
extern int yylex_destroy(void);

void *arvore = NULL;
void exporta (void *arvore);
void libera (void *arvore);

int main (int argc, char **argv)
{
  int ret = yyparse();
  exporta (arvore);
  libera(arvore);
  arvore = NULL;
  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 or-

dem dos filhos de um nó da árvore não importa na avaliação objetiva (mas importa para sua solução em etapas seguintes). 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 em memória, a estrutura da árvore da solução será comparada com aquela 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 memória alocada dinamicamente deve ser bem gerenciada. Neste sentido, um teste automático consistirá em lançar a ferramenta `valgrind` para verificar se toda a memória alocada dinamicamente foi integralmente liberada antes do término do programa. Ao lançar o compilador com uma determinada entrada, assim:

```
valgrind ./etapa3 < uma_entrada_correta
```

Será observada esta saída do `valgrind`:

```
==26684==    definitely lost: 0 bytes in 0 blocks
==26684==    indirectly lost: 0 bytes in 0 blocks
==26684==      possibly lost: 0 bytes in 0 blocks
==26684==    still reachable: 0 bytes in 0 blocks
==26684==         suppressed: 0 bytes in 0 blocks
```

Somente passará o teste em que todos os valores de bytes e blocos forem zero. Qualquer valor diferente de `0 bytes in 0 blocks` será interpretado como não liberação correta da memória alocada dinamicamente, e o teste será avaliado para falha.

C Sobre a Árvore Sintática Abstrata

A árvore sintática abstrata, do inglês *Abstract Syntax Tree* (AST), é uma árvore n-ária onde os nós folha representam os tokens presentes no programa fonte, os nós intermediários são utilizados para criar uma hierarquia que condiz com as regras sintáticas, e a raiz representa o programa inteiro, ou a primeira função do programa. Essa árvore se inspira nas derivações do analisador sintático, tornando mais fáceis as etapas posteriores de verificação semântica e síntese de código.

A árvore é abstrata (quando comparada a árvore de derivação gramatical completa) porque não detalha todas as derivações gramaticais para uma entrada dada. Tipicamente são omitidas derivações intermediárias onde um símbolo não terminal gera somente um outro símbolo terminal, tokens que são palavras reservadas, e todos os símbolos de sincronismo ou identificação do código, os quais estão implícitos na estrutura hierárquica criada. São mantidos somente os nós fundamentais para manter a semântica da entrada. A estrutura do nível de detalhamento de uma AST pode depender das escolhas de projeto de um compilador.

Os nós da árvore são frequentemente de `tipos` relacionados aos símbolos não terminais, ou a nós que representam operações diferentes, no caso das expressões. É importante notar que normalmente as declarações de tipos e variáveis são omitidas da AST, pois estas construções linguísticas não geram código, salvo nas situações onde as variáveis declaradas devem ser inicializadas.