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 Usar o módulo `asd` para construção da AST

É disponibilizado o módulo `asd`, composto de dois arquivos (`asd.c` e `asd.h`), que deve ser incorporado no projeto para que seja realizada a construção da árvore. As funções públicas do módulo encontram-se listadas e documentadas no arquivo `asd.h`. A implementação das funções encontra-se no arquivo `asd.c`. Altere o processo de compilação do seu projeto para incluí-lo.
Todos os nós da AST devem possuir um nome, que serve como um rótulo, um *label*. O nome que deve ser utilizado 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 lexema do token `TK_PR_WITH`. Para o comando de atribuição, o nome deve ser o lexema do token `TK_PR_IS`. 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 ao token `TK_PR_RETURN`. Para os comandos de controle de fluxo, deve-se utilizar como nome o lexema do token `TK_PR_IR` para o comando if com else opcional, e o lexema do token `TK_PR_WHILE` para o comando while. Para as expressões aritméticas, devem ser utilizados os próprios operadores unários ou binários como nomes. Para as expressões lógicas e relacionais, deve-se utilizar o lexema correspondente. Enfim, para os identificadores e literais, utiliza-se o próprio lexema.

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 das folhas 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 (`asd_new`), e associação desta com seus filhos que já foram criados (`asd_add_child`). 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 comandos. 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 Verificação de alocação de memória

Adicione a *flag* `-fsanitize=address` na variável `CFLAGS` do seu `Makefile`. Em seguida, tenha certeza que essa flag é passada para o `gcc` na compilação de todos os arquivos do seu projeto. No final da execução do seu compilador, o mesmo reportará a existência de vazamento de memória (*memory leak*). Garanta que seu compilador não reporte nenhum vazamento de memória, fazendo o gerenciamente adequado de memória.

2.5 Remoção de conflitos/ajustes gramaticais

Conflitos *Reduce-Reduce* e *Shift-Reduce* devem ser removidos, caso existam após modificações gramaticais.

A Arquivos `asd.c` e `asd.h`

Acesse-os diretamente no moodle.

B Arquivo `main.c`

A função principal da E3 aparece abaixo. A variável global `arvore` de tipo `asd_tree_t*` é um ponteiro para a estrutura de dados que deverá conter a raiz da AST do programa fornecido em entrada. A função `asd_print_graphviz` exporta a AST no formato DOT. A função `asd_free` libera a memória associada a AST de maneira recursiva. Utilize o comando `extern asd_tree_t *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`.

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

C Avaliação objetiva

No processo de avaliação automática, será considerada como raiz o primeiro nó do grafo exportado 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.dot
```

O conteúdo de `saida.dot` contém a árvore da solução no formato DOT. 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. Reforça-se que a ordem dos nós filhos não importa.