Especificação da Etapa 3 do Projeto de Compilador

Criação da Árvore Sintática Abstrata

A terceira etapa do trabalho de implementação de um compilador para a **Linguagem Amarela** consiste na criação da árvore sintática abstrata (AST) baseada no programa de entrada, escrito em **Amarela**, e considerando as convenções estabelecidas na Seção 3. A árvore deve ser criada a medida que as regras semânticas são executadas e deve ser mantida em memória mesmo após o fim da análise sintática (ou seja, quando `yyparse` retornar). A avaliação deste trabalho será feita de duas formas: primeiro, através de uma análise subjetiva visual da árvore, através da geração de um arquivo em formato `dot` definido pelo pacote GraphViz (funções serão fornecidas para tal através do repositório git do professor); segundo, por uma comparação automática da árvore gerada com aquela esperada para um determinado programa fonte.

1 Funcionalidades Necessárias

A solução do grupo para a análise sintática deve ter as seguintes funcionalidades:

1.1 Implementação de uma estrutura de dados em árvore

Deve ser definido um novo tipo de dado para uma estrutura de dados em árvore. Cada nó desta árvore tem uma série de informações relacionadas a esta etapa do trabalho (veja a Seção 2.1 para detalhes). Dentre estas, salienta-se o fato de que cada nó deve ter um número arbitrário de filhos que também são nós da árvore. O nome do tipo de dado para o nó da árvore deve ser `comp_tree_t`. Este novo tipo de dado deve vir acompanhado de funções tradicionais tais como criação, remoção, alteração, e qualquer outra função que o grupo achar pertinente implementar.

1.2 Criar a árvore sintática abstrata

Criar a árvore sintática abstrata para uma entrada qualquer escrita em **Amarela**, instrumentando a gramática com ações semânticas ao lado das regras de produção descritas no arquivo `parser.y` para a criação dos nós da árvore e conexão entre eles (veja a Seção 3 para detalhes sobre os nós da árvore). A árvore deve permanecer em memória após o fim da análise sintática, ou seja, acessível na função `main_finalize` do programa.

1.3 Remoção de conflitos e ajustes gramaticais

A solução apresentada pelo grupo para a remoção de conflitos *Reduce/Reduce*[1] e *Shift/Reduce*[2] da etapa anterior, realizada através dos comandos `%left`, `%right` ou `%nonassoc`) do bison pode fazer com que a árvore sintática gerada nesta etapa seja diferente daquela esperada e detalhada na Seção 3. Um outro motivo para estas diferenças pode advir da gramática ser muito diferente, com produções que não permitam a geração apropriada da árvore sintática tal qual ela é descrita nesta especificação. Caso estas situações ocorram, o grupo deve realizar novos ajustes gramaticais e acertar a ordem dos comandos citados acima que removem conflitos. De qualquer forma, a solução desta etapa deve ser livre de conflitos informados pelo bison e deve se adequar a especificação AST da Seção 3.

1.4 Gerar a árvore em formato *dot* para análise gráfica

Gerar o arquivo em format *dot* para análise gráfica e avaliação utilizando as funções fornecidas pelo professor para que o grupo possa visualizar a árvore sintática abstrata gerada. Essas funções estão no repositório, nos arquivos `gv.c` e `gv.h`, devidamente documentados. Somente as funções `gv_declare` e `gv_connect` precisam ser utilizadas pelo grupo. A árvore será impressa na saída padrão do programa, podendo ser redirecionada para arquivo.

[1] `http://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html`
[2] `http://www.gnu.org/software/bison/manual/html_node/Shift_002fReduce.html`

1.5 Implementar dois programas utilizando Amarela

Dois programas utilizando a sintaxe da linguagem **Amarela** devem ser implementados e disponibilizados juntamente com a solução desta etapa. O grupo tem a liberdade de escolher qualquer algoritmo para ser implementado.

2 Descrição da Árvore

A árvore sintática abstrata, ou AST (*Abstract Syntax Tree*), é uma árvore n-ária onde os nós intermediários representam símbolos não terminais, os nós folha representam tokens presentes no programa fonte, e a raiz representa o programa corretamente analisado. Essa árvore registra as derivações reconhecidas pelo analisador sintático, e torna mais fáceis as etapas posteriores de verificação e síntese, já que permite consultas em qualquer ordem. A árvore é abstrata porque não precisa representar detalhadamente todas as derivações. Tipicamente serão 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 reconhecida. Os nós da árvores serão de tipos relacionados aos símbolos não terminais, ou a nós que representam operações diferentes, no caso das expressões. **Declarações de variáveis, tanto globais quanto locais, não figuram na AST.**

2.1 Nó da AST

Cada nó da AST tem um tipo associado, e este deve ser um dos tipos declarados no arquivo `iks_ast.h` disponibilizado (veja seção 4). Quando o nó da AST é do tipo IKS_AST_IDENTIFICADOR, IKS_AST_LITERAL, ou ainda IKS_AST_FUNCAO, ele deve conter obrigatoriamente um ponteiro para a entrada correspondente na tabela de símbolos. Além disso, cada nó da AST deve ter uma estrutura que aponte para os seus filhos. O apêndice 3 detalha o que deve ter para cada tipo de nó da AST.

3 Descrição detalhada dos nós da AST

Esta seção apresenta graficamente como deve ficar cada nó da AST considerando as suas características, principalmente a quantidade de nós filhos. **Casos omissos devem ser resolvidos através do fórum do projeto.** As subseções seguintes tem nomes de acordo com os `define`s no arquivo `iks_ast.h` disponibilizado pelo professor. Em todas as subções seguintes, considere a seguinte convenção:

- Onde aparecer *Comando*, considere que o nó da árvore pode ser qualquer um dos seguintes tipos:
  - IKS_AST_IF_ELSE
  - IKS_AST_DO_WHILE
  - IKS_AST_WHILE_DO
  - IKS_AST_INPUT
  - IKS_AST_OUTPUT
  - IKS_AST_ATRIBUICAO
  - IKS_AST_RETURN
  - IKS_AST_BLOCO
  - IKS_AST_CHAMADA_DE_FUNCAO

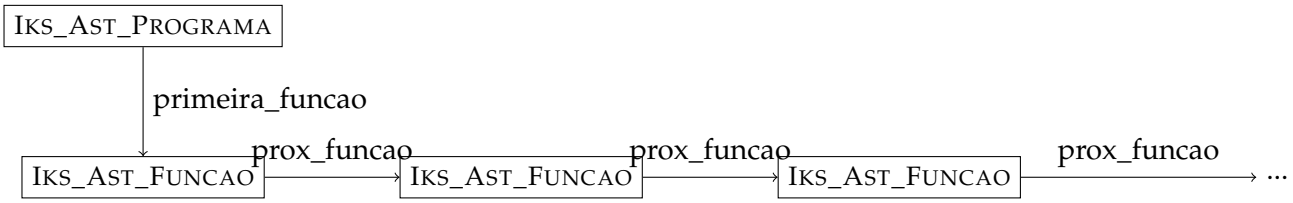
- Onde aparecer *Condição*, considere que o nó da árvore pode ser qualquer um dos seguintes tipos:
  - IKS_AST_IDENTIFICADOR
  - IKS_AST_LITERAL
  - IKS_AST_ARIM_SOMA
  - IKS_AST_ARIM_SUBTRACAO
  - IKS_AST_ARIM_MULTIPLICACAO
  - IKS_AST_ARIM_DIVISAO
  - IKS_AST_ARIM_INVERSAO
  - IKS_AST_LOGICO_E
  - IKS_AST_LOGICO_OU
  - IKS_AST_LOGICO_COMP_DIF
  - IKS_AST_LOGICO_COMP_IGUAL
  - IKS_AST_LOGICO_COMP_LE
  - IKS_AST_LOGICO_COMP_GE
  - IKS_AST_LOGICO_COMP_L
  - IKS_AST_LOGICO_COMP_G
  - IKS_AST_LOGICO_COMP_NEGACAO
  - IKS_AST_VETOR_INDEXADO
  - IKS_AST_CHAMADA_DE_FUNCAO

- Onde aparecer *Saída*, considere que o nó da árvore pode ser qualquer um dos seguintes tipos, levando em conta que IKS_AST_LITERAL só pode ser uma string:

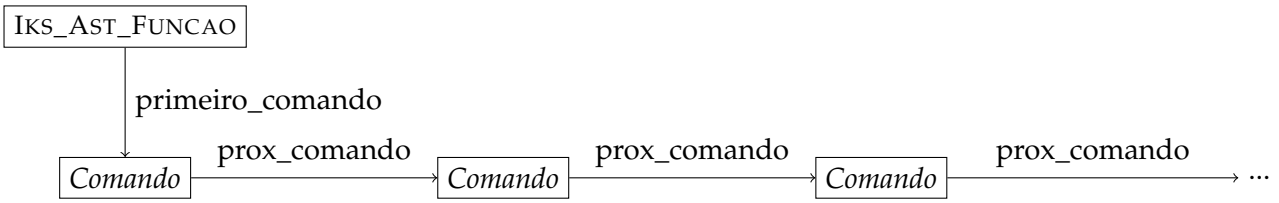
- IKS_AST_IDENTIFICADOR
- IKS_AST_LITERAL
- IKS_AST_ARIM_SOMA
- IKS_AST_ARIM_SUBTRACAO
- IKS_AST_ARIM_MULTIPLICACAO
- IKS_AST_ARIM_DIVISAO
- IKS_AST_ARIM_INVERSAO
- IKS_AST_LOGICO_E
- IKS_AST_LOGICO_OU
- IKS_AST_LOGICO_COMP_DIF
- IKS_AST_LOGICO_COMP_IGUAL
- IKS_AST_LOGICO_COMP_LE
- IKS_AST_LOGICO_COMP_GE
- IKS_AST_LOGICO_COMP_L
- IKS_AST_LOGICO_COMP_G
- IKS_AST_LOGICO_COMP_NEGACAO
- IKS_AST_VETOR_INDEXADO
- IKS_AST_CHAMADA_DE_FUNCAO

- Onde aparecer *Expressão*, considere que o nó da árvore pode ser qualquer um dos seguintes tipos:
  - IKS_AST_IDENTIFICADOR
  - IKS_AST_LITERAL
  - IKS_AST_ARIM_SOMA
  - IKS_AST_ARIM_SUBTRACAO
  - IKS_AST_ARIM_MULTIPLICACAO
  - IKS_AST_ARIM_DIVISAO
  - IKS_AST_ARIM_INVERSAO
  - IKS_AST_LOGICO_E
  - IKS_AST_LOGICO_OU
  - IKS_AST_LOGICO_COMP_DIF
  - IKS_AST_LOGICO_COMP_IGUAL
  - IKS_AST_LOGICO_COMP_LE
  - IKS_AST_LOGICO_COMP_GE
  - IKS_AST_LOGICO_COMP_L
  - IKS_AST_LOGICO_COMP_G
  - IKS_AST_LOGICO_COMP_NEGACAO
  - IKS_AST_VETOR_INDEXADO
  - IKS_AST_CHAMADA_DE_FUNCAO

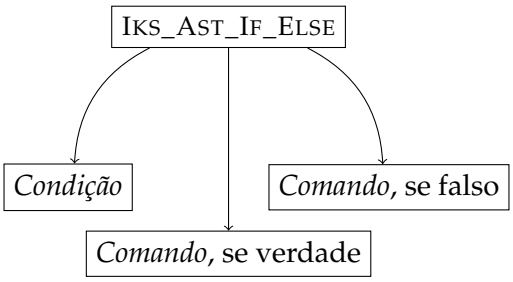
3.1 IKS_AST_PROGRAMA



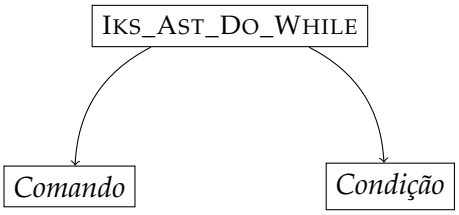
3.2 IKS_AST_FUNCAO



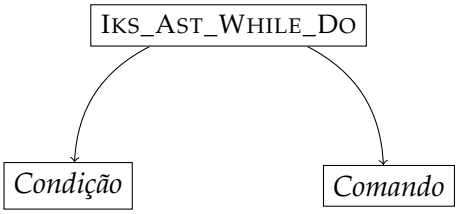
3.3 IKS_AST_IF_ELSE



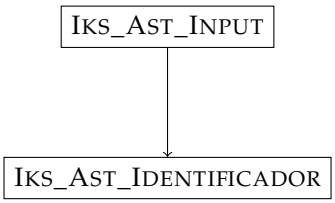
3.4 IKS_AST_DO_WHILE



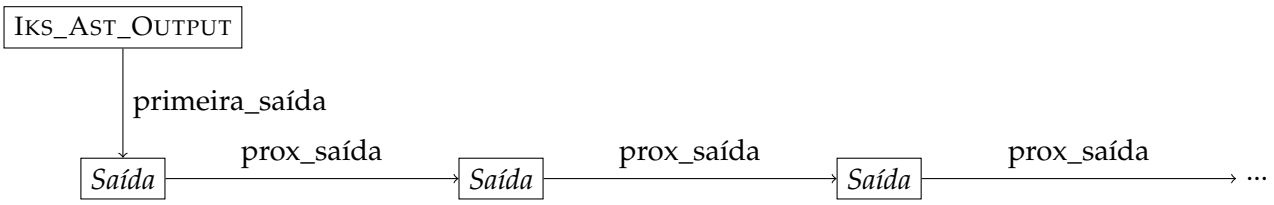
3.5 IKS_AST_WHILE_DO



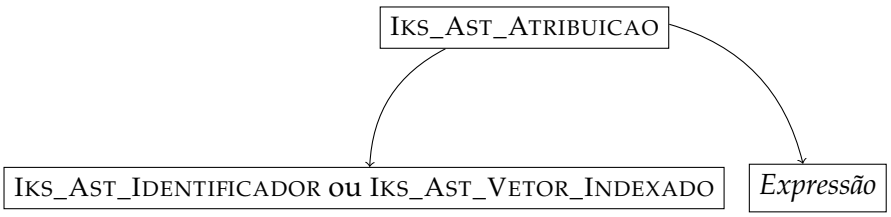
3.6 IKS_AST_INPUT



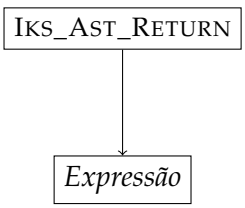
3.7 IKS_AST_OUTPUT



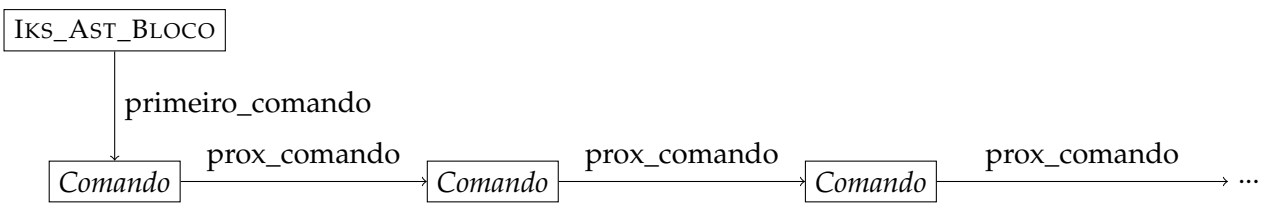
3.8 IKS_AST_ATRIBUICAO



3.9 IKS_AST_RETURN



3.10 IKS_AST_BLOCO (recursivo)



3.11 IKS_AST_IDENTIFICADOR e IKS_AST_LITERAL

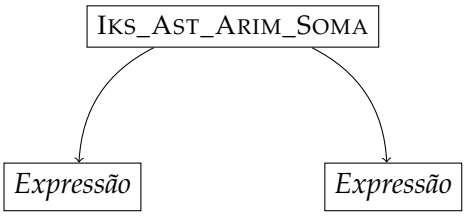
Os nós do tipo IKS_AST_IDENTIFICADOR e IKS_AST_LITERAL não têm filhos que são nós da AST. No entanto, eles devem ter obrigatoriamente um ponteiro para a entrada na tabela de símbolos.

3.12 Expressões aritméticas com dois operandos

Os nós do tipo

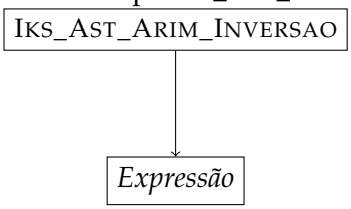
- IKS_AST_ARIM_SOMA
- IKS_AST_ARIM_SUBTRACAO
- IKS_AST_ARIM_MULTIPLICACAO e
- IKS_AST_ARIM_DIVISAO

têm dois filhos, como mostrado abaixo (utilizando neste exemplo o nó do tipo IKS_AST_ARIM_SOMA).



3.13 Expressão aritmética unária

O nó do tipo IKS_AST_ARIM_INVERSAO tem somente um filho, como mostrado abaixo.

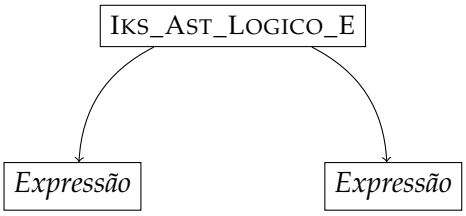


3.14 Expressões lógicas com dois operandos

Os nós do tipo

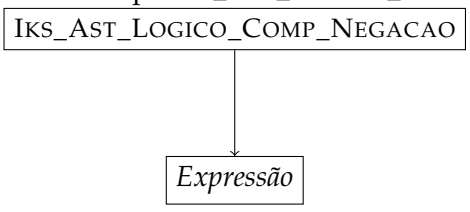
- IKS_AST_LOGICO_E
- IKS_AST_LOGICO_OU
- IKS_AST_LOGICO_COMP_DIF
- IKS_AST_LOGICO_COMP_IGUAL
- IKS_AST_LOGICO_COMP_LE
- IKS_AST_LOGICO_COMP_GE
- IKS_AST_LOGICO_COMP_L e
- IKS_AST_LOGICO_COMP_G

têm dois filhos, como mostrado abaixo (utilizando neste exemplo o nó do tipo IKS_AST_LOGICO_E).

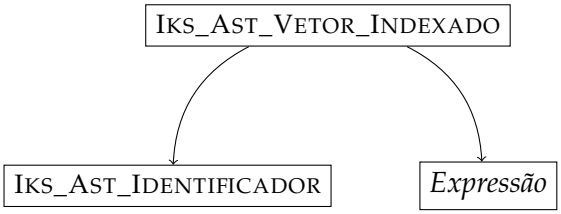


3.15 Expressão lógica unária

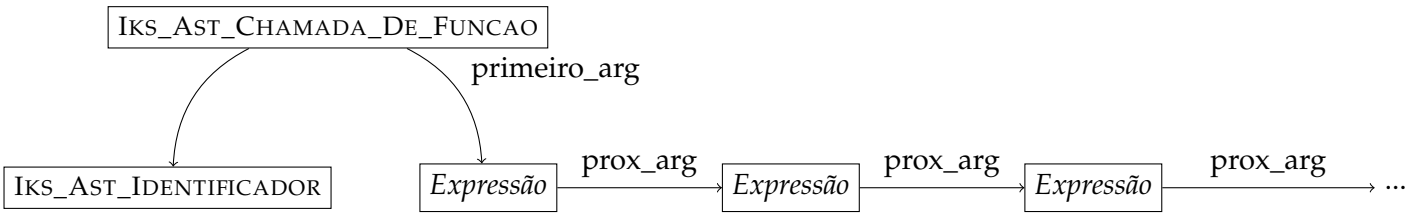
O nó do tipo IKS_AST_LOGICO_COMP_NEGACAO tem somente um filho, como mostrado abaixo.



3.16 IKS_AST_VETOR_INDEXADO



3.17 IKS_AST_CHAMADA_DE_FUNCAO



4 Requisitos Obrigatórios

Os requisitos obrigatórios são os mesmos da seção equivalente na especificação da etapa 2 do projeto de compilador. O ambiente de compilação e execução desta etapa podem ser definidos da seguinte forma:

```
$ git pull origin master
$ cmake -DETAPA_1=OFF -DETAPA_2=OFF -DETAPA_3=ON ..
$ make
```

Para os testes da avaliação automática, basta executar o comando:

```
ctest -R e3
```

Caso os testes falhem, utilize a opção `-V` com o comando `ctest`.

5 Dúvidas

Crie um novo tópico no fórum de discussão do projeto de compilador no moodle.

Bom trabalho!