Projeto de Compilador
Etapa 3 : Árvore Sintática Abstrata

Stéfano Drimon Kurz Mór e Gabriel Bronzatti Moro

25 de Setembro de 2016

A terceira etapa do trabalho de implementação de um compilador para a linguagem Saga consiste na criação da árvore sintática abstrata (*Abstract Syntax Tree* – AST) baseada no programa de entrada, escrito em Saga, e considerando as convenções estabelecidas na Seção 3.2. 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 dos tutores); segundo, por uma comparação automática da árvore gerada com aquela esperada para um determinado programa fonte.

1 Correção da Gramática da E2

É importante, para a E3 e subsequentes, que não reste dúvida ou ambiguidade na escrita da gramática implementada na E2. Existem, no entanto, pelo menos duas interpretações possíveis para o uso de ponto-e-vírgula e blocos observadas na especificação da E2. (Os que alcançaram 100% de acerto nos testes da E2 nada precisam fazer e podem pular essa parte.) A partir desse momento é necessário, então, ajustar a gramática para que os programas Saga tenham o seguinte comportamento:

1. O ponto-e-vírgula deve ser **opcional** na **última** instrução de um bloco. Por exemplo, `{a = 5 ; b = 6}` ou `{a = 5}` são construções válidas.

2. O ponto-e-vírgula deve ser **opcional** após o **fechamento** de um bloco caso ele esteja dentro de outro bloco. Por exemplo, `{ {a = 5 ; b = 6} ; }` ou `{ {a = 5 ; b = 6} }` são construções válidas.

É importante ressaltar que, dependendo da gramática utilizada, é possível que outras alterações sejam necessárias para cumprir requisitos dessa etapa e das subsequentes. Cabe aos alunos identificar essas mudanças e implementá-las.

Dúvidas podem se tiradas com os tutores a qualquer momento.

2 Funcionalidades Necessárias

2.1 Criar a Árvore Sintática Abstrata

Criar a árvore sintática abstrata para uma entrada qualquer escrita em Saga, 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 3.2 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.

2.2 Remoção de Conflitos/Ajustes Gramaticais

A solução apresentada pelo grupo para a remoção de conflitos *Reduce-Reduce* e *Shift-Reduce* 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 3.2. 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 3.2.

2.3 Implementar programas em Saga

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

3 Descrição da Árvore

A árvore sintática abstrata, do inglês Abstract Syntax Tree (AST), 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. É importante notar que declarações de tipos e variáveis não figuram na AST, pois não geram código.

3.1 Nó da AST

Cada nó da AST tem um tipo associado, e este deve ser um dos tipos declarados no arquivo `cc_ast.h` disponibilizado. Quando o nó da AST for um dos tipos:

AST_IDENTIFICADOR AST_LITERAL 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 código da estrutura em árvore já está disponível e deve ser usado (`src/cc_ast.c` com protótipos em `include/cc_ast.h`) O apêndice 3.2 detalha o que deve ter para cada tipo de nó da AST.

3.2 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. As subseções seguintes tem nomes de acordo com os comandos do tipo `#define` no arquivo `cc_ast.h`. Em todas as subseções seguintes, considere a seguinte regra de generalização para um determinado nó da árvore e seus possíveis tipos.

Comando

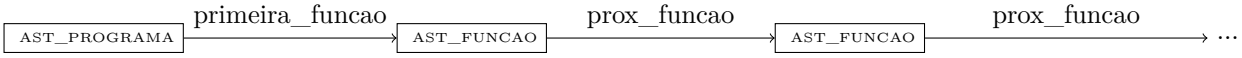
AST_IF_ELSE	AST_DO_WHILE	AST_WHILE_DO	AST_ATRIBUICAO
AST_RETURN	AST_BLOCO	AST_CHAMADA_DE_FUNCAO	

Condição e Expressão

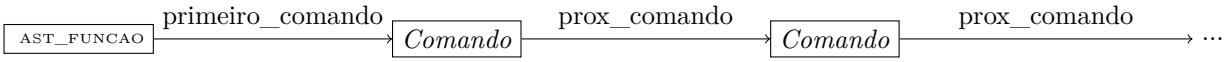
AST_IDENTIFICADOR	AST_LITERAL	AST_ARIM_SOMA
AST_ARIM_SUBTRACAO	AST_ARIM_MULTIPLICACAO	AST_ARIM_DIVISAO
AST_ARIM_INVERSAO	AST_LOGICO_E	AST_LOGICO_OU
AST_LOGICO_COMP_DIF	AST_LOGICO_COMP_IGUAL	AST_LOGICO_COMP_LE
AST_LOGICO_COMP_GE	AST_LOGICO_COMP_L	AST_LOGICO_COMP_G
AST_LOGICO_COMP_NEGACAO	AST_VETOR_INDEXADO	AST_CHAMADA_DE_FUNCAO

3.2.1 Programa e Função

1. AST_PROGRAMA

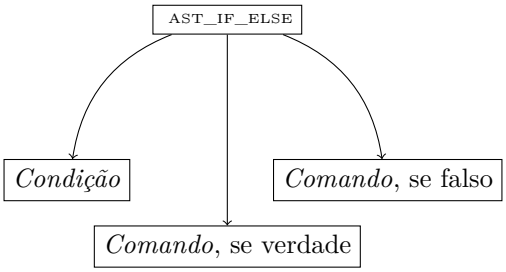


2. AST_FUNCAO

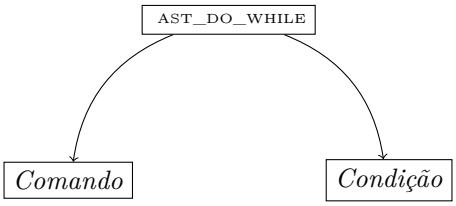


3.2.2 Comandos

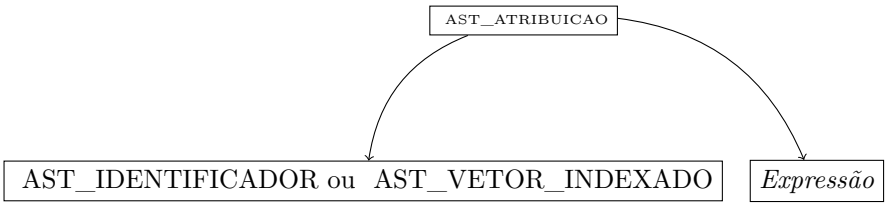
1. AST_IF_ELSE (com o `else` sendo opcional)



2. AST_DO_WHILE e AST_WHILE_DO



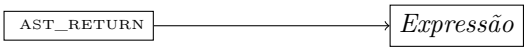
3. AST_ATRIBUICAO



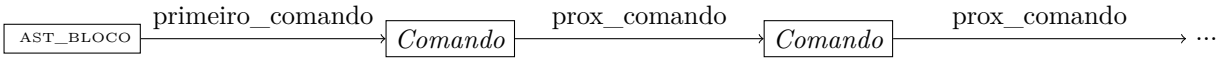
   (a) Declaração com inicialização
   Declarações de variáveis em geral não aparecem na AST. No caso específico onde uma declaração de variável tem uma inicialização de valor, esta deve aparecer na AST pelo fato que é passível de gerar código. Sendo assim, a árvore deve ser semelhante aquela para AST_ATRIBUICAO.

   (b) Atribuição para campos de um tipo definido pelo usuário
   Nos casos onde temos `identificador!campo = expressão`, a AST correspondente deve ser idêntica a AST_ATRIBUICAO, com um nó adicional filho (do tipo AST_IDENTIFICADOR) para identificador o campo.

4. AST_RETURN



5. AST_BLOCO (recursivo)



3.2.3 Condição, Expressão

1. AST_IDENTIFICADOR e AST_LITERAL

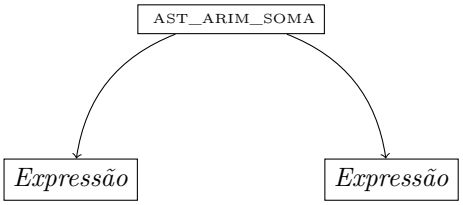
   Os nós do tipo AST_IDENTIFICADOR e AST_LITERAL não têm filhos que são nós da AST. No entanto, eles devem ter obrigatoriamente um ponteiro para a entrada na tabela de símbolos.

2. Expressões Aritméticas Binárias

   Os nós do tipo:

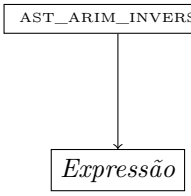
   - AST_ARIM_SOMA
   - AST_ARIM_SUBTRACAO
   - AST_ARIM_MULTIPLICACAO
   - AST_ARIM_DIVISAO

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



3. Expressão Aritmética Unária

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

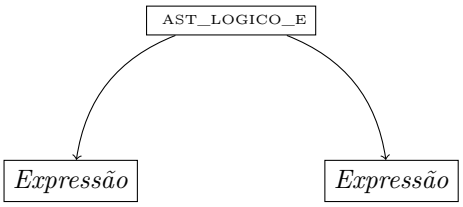


4. Expressões Lógicas Binárias

   Os nós do tipo:

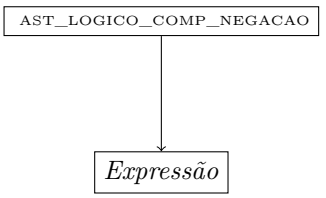
   - AST_LOGICO_E
   - AST_LOGICO_OU
   - AST_LOGICO_COMP_DIF
   - AST_LOGICO_COMP_IGUAL
   - AST_LOGICO_COMP_LE
   - AST_LOGICO_COMP_GE
   - AST_LOGICO_COMP_L
   - AST_LOGICO_COMP_G

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

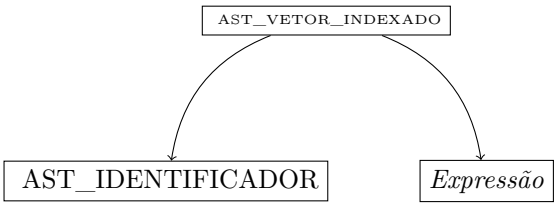


5. Expressão Lógica Unária

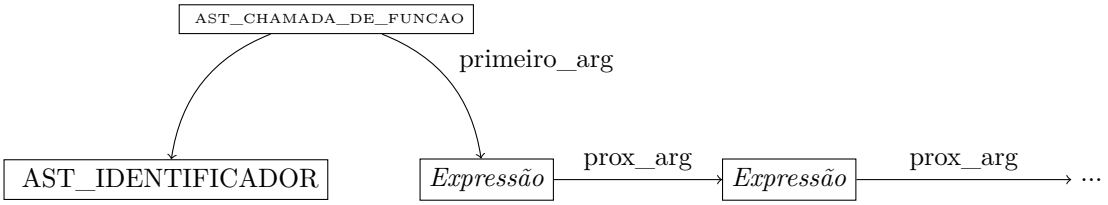
   O nó do tipo AST_LOGICO_COMP_NEGACAO tem somente um filho, como mostrado abaixo.



6. AST_VETOR_INDEXADO



7. AST_CHAMADA_DE_FUNCAO



3.2.4 Outras Construções Presentes na Sintaxe

A construção da AST para os comandos não listados acima mas que fazem parte da sintaxe são opcionais.

4 Casos Omissos

Casos não previstos serão discutidos com os tutores. Abaixo os casos omissos já detectados e cujo interpretação já foi definida.

5 Regras Gerais

Veja no moodle as regras gerais de entrega.