Aluno: José Carlos Tobias da Silva
RA: 2016.1.08.007

Documentação de Implementação

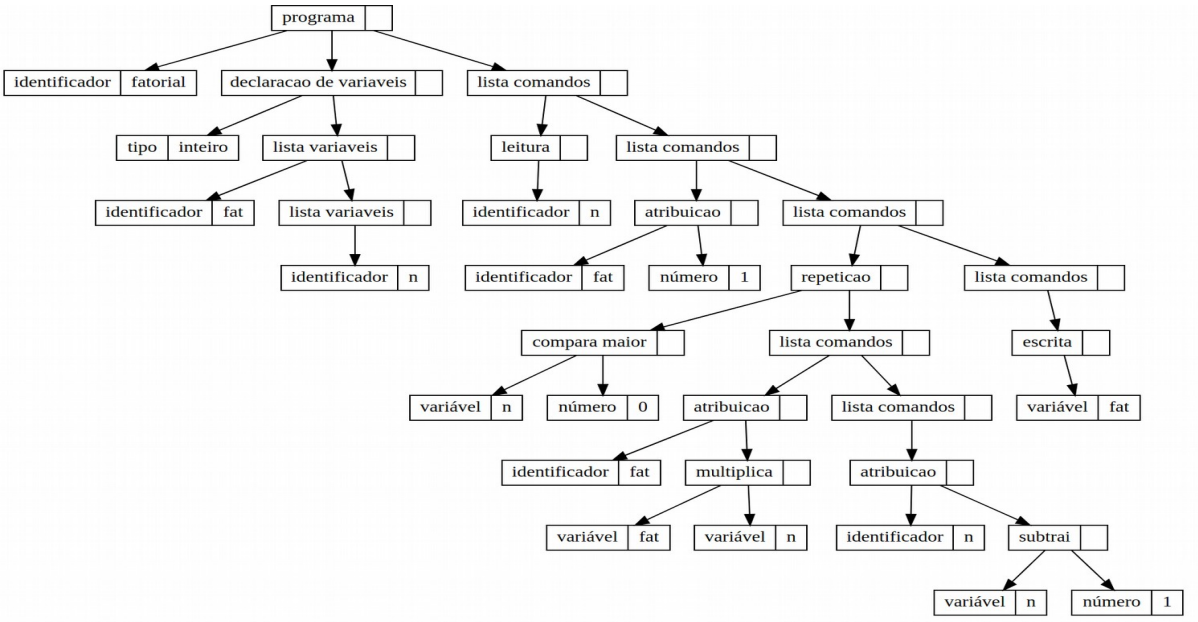
Parte 1 – Geração da Árvore Sintática

Nesta parte do trabalho, devemos montar a árvore sintática que permite que diversas ações sejam executadas sobre o programa percorrendo-se a árvore. Ao se executar a leitura do código então, o programa deve montar a árvore correspondente para que as ações futuras possam ser tomadas.

Como exemplo para a montagem da árvore, dado que o analisador sintático está a percorrer o programa, temos então um código como:

```
programa fatorial
    inteiro n fat
inicio
    leia n
    fat ← 1
    enquanto n > 0 faca
        fat ← fat * n
        n ← n – 1
    fimenquanto
    escreva fat
fimprograma
```

Este programa pode ser traduzido em uma árvore como abaixo



Esta árvore é produzida enquanto a análise sintática do código é feita, sendo assim, cada estrutura reconhecida pela análise é traduzida também em uma estrutura de nós da árvore. Como por exemplo no programa acima, a estrutura de repetição é montada de maneira que esta subárvore de raíz com nó “repeticao” é uma estrutura reconhecida em análise sintática e agora também reconhecida como uma estrutura na árvore.

É de se observar que alguns nós da árvore não possuem tradução direta na linguagem e nem no jeito em que os *tokens* são formatados. Tokens como o do tipo de declaração de lista de variáveis, para a declaração de variáveis e para variáveis foram criados para que estes nós tenham formatação exclusiva na sua saída para o arquivo do grafo, permitindo que a estrutura seja montada de maneira formatada corretamente.

A montagem dessa árvore é feita durante as chamadas das regras do analisador sintático de maneira recursiva, como por exemplo em uma regra do tipo de repetição, temos:

repeticao: T_ENQTO expressao T_FACA lista_comandos T_FIMENQTO

Executamos no momento dessa ação a montagem de nós e chamadas para a estrutura em código C, fazendo então o trecho de código ficar como em:

```
repeticao: T_ENQTO expressao T_FACA lista_comandos T_FIMENQTO {
        pt = criarNo(T_ENQTO, "repeticao");
        adcionarFilho(pt, $4);
        adcionarFilho(pt, $2);
        $$ = pt;
}
;
```

Os símbolos $4 e $2 são espécies de estruturas criadas pela ferramenta Bison para estruturar a análise sintática permitindo que ações sejam tomadas durante a mesma. Antes, no arquivo léxico, a linha "*#define YYSTYPE ptno*" faz com que a estrutura utilizada sejam nós da árvore, sendo assim, $4 e $2 são já nós da árvore prontos para serem formataddos e utilizados.

Fica então fácil de perceber a montagem da estrutura da subárvore de raíz "repeticao" e seus filhos.

Parte 2 – Geração de Código MIPS

Nesta parte já temos a disposição a árvore sintática do programa, e devemos então percorrer a mesma executando ações que permitam transformar ações explicitadas nessa árvore em um programa MIPS. O programa MIPS gerado é real e executável, como pode ser verificado através do simulador MARS (https://courses.missouristate.edu/KenVollmar/MARS/) que executa o código com perfeição.

Para tradução do programa para linguagem MIPS, deve ser feito um percurso na árvore sintática já montada na parte 1, e então as mesmas devem ser transcritas para código MIPS. Como por exemplo, na parte do código em que se lê:

*fat ← fat * n*

Deve então ser feita a tradução para MIPS como em:

```
lw $a0 fat
sw $a0 0($sp)
addiu $sp $sp -4
lw $a0 n
lw $t1 4($sp)
addiu $sp $sp 4
mult $t1 $a0
mflo $a0
```

sw $a0, fat

Observe que as operações utilizam estruturas de empilhamento e desempilhamento de valores, fazendo com que se possa trabalhar de maneira mais correta com os valores utilizando uma abstração de pilhas.

Tal procedimento é feito para todos os nós necessários para a escrita do programa. Existem porém, nós como o do tipo de lista de variáveis, que não têm tradução imediata para código MIPS, mas que devem ser utilizado para se executar outras ações semânticas sobre o código, como por exemplo para inserção de variáveis tipadas na tabela de símbolos.

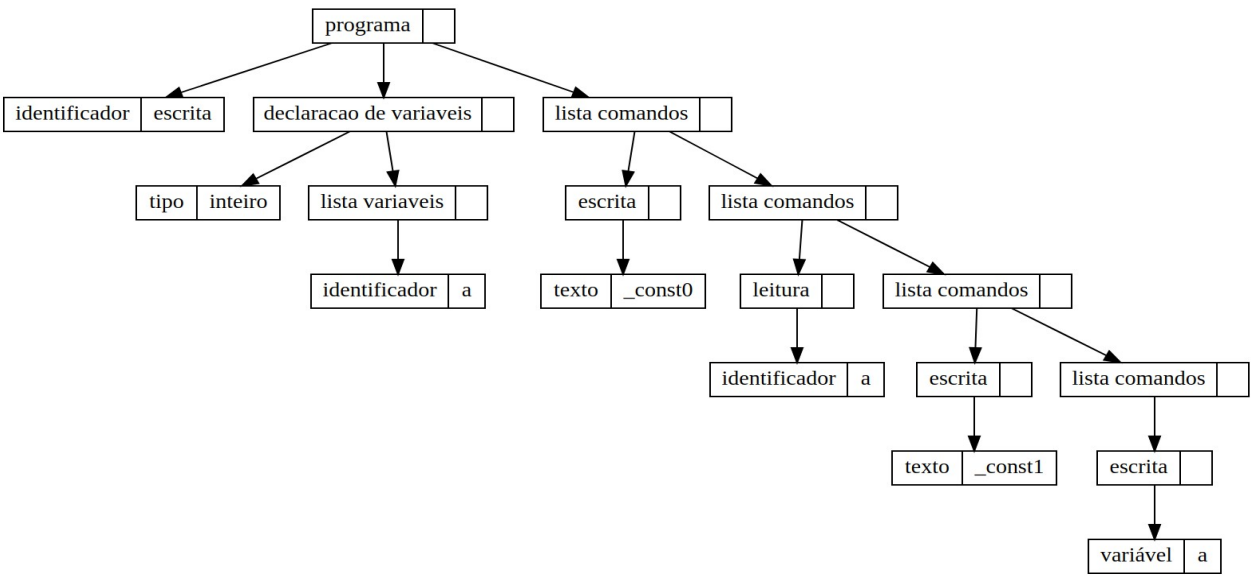
Parte 2.1 – Criação de Literais

A adição de literais na linguagem foi feita utilizando-se de um esquema parecido com o que é utilizado em linguagem C, porém obviamente menos robusto como pode ser visto na documentação oficial do Flex (http://www.delorie.com/gnu/docs/flex/flex_11.html). É utilizado um esquema parecido com o de leitura (ou não leitura no caso) de comentários, que se aproveita da estrutura em forma de autômato criada pelo Flex, fazendo com que estados possam ser criados para se executar ações que têm início e fim.

As cadeias de literais são então lidas na análise léxica, passadas para a análise sintática e guardadas indexadamente, em um vetor de strings, para uso. Por exemplo, com um programa do tipo:

```
programa escrita
        inteiro a
inicio
        escreva “digite um valor para a: “
        leia a
        escreva “o valor digitado para a foi: “
        escreva a
fimprograma
```

Temos a seguinte árvore sintática:



Com o seguinte código MIPS:

```
.data
        _esp: .asciiz " "
        _ent: .asciiz "\n"
        _const0: .asciiz "digite um valor para a: "
        _const1: .asciiz "o valor digitado para a foi: "
        a: .word 1
.text
        .globl main
main:   nop
        la $a0 _const0
        li $v0, 4
        syscall
        li $v0, 5
        syscall
        sw $v0, a
        la $a0 _const1
        li $v0, 4
        syscall
        lw $a0 a
        li $v0, 1
        syscall
fim:    nop
```

É assim então feita a leitura e escrita de literais no programa, utilizando o esquema de *“quoted strings”* como em linguagem C e indexação para retorno da string a ser escrita no aquivo de código MIPS.

Parte 2.2 – Checagem de Tipos

A checagem de tipos deve ser feita com o objetivo de se assegurar que os tipos das operações condizem com sua finalidade ou local. Por exemplo, não devemos armazenar um tipo lógico em uma variável numérica, ou um valor numérico em variável lógica, assim como para desvios no código se deve ser uma condição lógica e não um retorno numérico, pelo menos nessa linguagem que está a ser trabalhada.

Uma pequena modificação na estrutura do elemento da tabela de símbolos foi feita, permitindo que o tipo de uma determinada variável seja guardada junto da mesma na tabela de símbolos. Isso faz com que seja fácil de se buscar em operações posteriores o tipo de determinado identificador, seja para atribuição ou operações que o envolvam.

Foi então requisitado para se fazer a checagem de tipos para atribuições, repetições e seleções. Temos então que:

- **Para atribuições:** devemos verificar se a variável que está a receber o resultado de uma expressão é do mesmo tipo do retorno desta expressão.
- **Para repetições e seleções:** devemos verificar se as expressões que compõe a condição possuem valor lógico.

Na linguagem em questão temos operações que são feitas especificamente sobre números e devem retornar valores numéricos sendo as operações básicas aritméticas (soma, subtração, divisão e multiplicação), e temos operações que são feitas sobre tipo numérico que retornam valor lógico

(operações de comparação numérica maior, menor e igual) e operações sobre valores lógicos que retornam também valor lógico (conjunção, disjunção e negação lógica).

Assim sendo, para se garantir o tipo das operações, basta se checar o tipo de todos os envolvidos nas mesmas e as operações feitas entre eles, além de seu local de retorno. Isso pode ser feita utilizando-se de uma pilah semântica.