Compilador Na Linguagem TPP - semântica

Gabriel David Sacca[1]

[1] Departamento de Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 87301 – Campo Mourão – PR – Brazil

{gabrielsacca}@alunos.utfpr.edu.br

1. Análise semântica

Após a separação dos Tokens já concluída, e a verificação de todas as regras sintáticas também, a execução da analise semântica é o próximo passo, nela se faz necessário a correção de qualquer regra da linguagem que não é relatada na sintática como por exemplo declarações de funções com faltas de parâmetros e afins. Nesta sessão será discutida como toda a analise foi feita, desde a geração da tabela de símbolos e sua importância, até passando por todas as regras da linguagem TPP. A semântica é dividida em 3 grandes áreas, sendo elas a geração da tabela de símbolos o encontro de erros semânticos e a poda da arvore que serão analisados nas sub sessões sequentes.

1.1. Tabela de símbolos

A tabela de símbolos é um recurso muito importante para facilitar a geração da analise semântica, ela deve guardar qualquer informação referente a variáveis e a funções do código executado para assim facilitar a identificação de erros ou falhas no processo. Neste projeto a tabela de símbolos é tratada como um vetor de classes, onde cada instancia desta classe é equivalente a uma tupla da tabela de símbolos, os atributos dessa classe e seus significados são os seguintes:

- **Tipo**: Este atributo é responsável por separar o que é uma variável e o que é uma função nesta tabela, e isto é importante para algumas funcionalidades futuras que o analisador semântico irá necessitar como identificar variáveis que possuam algum valor;
- **Nome**: Este por sua vez identifica o nome da variável ou da função a ser guardada, ele é muito importante nas varias buscas que serão realizadas pela tabela para o bom funcionamento do analisador;
- **TipoValor**: já o tipoValor identifica qual o tipo armazenado dentro da variável ou, se for uma função, o tipo de retorno da mesma, ele é muito importante para tratar de erros como coerção de valores ou atribuições indevidas;
- **Parâmetros**: Os parâmetros existem exclusivamente para tuplas do tipo função, ele guarda os tipos e parâmetros de variáveis dessa função, e se a mesma não possuir nenhum o valor padrão é None. Os parâmetros serão uteis para identificar utilizações de variáveis e separação de escopos;
- **Dimensões**: Por outro lado as dimensões são exclusivas das variáveis e elas dizem se uma variável é um vetor ou uma matriz, se ela for uma matriz o atributo dimensões irá possuir um vetor com uma posição que contem o tamanho dessa dimensão, já se for uma matriz será um vetor com duas posições onde cada posição guarda o tamanho especifico da sua dimensão.
- **Escopo**: Este parâmetro salva o escopo da variável ou função analisada, ele serve para identificar duplicatas de variáveis ou funções;

- **Declarada**: A declarada é um atributo do tipo Boolean que caso seja True significa que uma variável já foi declarada, e caso seja False ela ainda deve ser declarada. Por padrão uma variável é criada com declarada True, pois ela é criada no momento da declaração;
- **Inicializada**: A inicializada segue os mesmos princípios da declarada, todavia ela é acionada apenas quando um valor é dado a uma variável, seja este valor em uma atribuição direta ou uma escrita do console;
- **Utilizada**: E por ultimo a utilizada que funciona como as duas anteriores, porem ela muda para o estado True quando utilizamos de alguma forma a variável, seja em retornos, consoles, atribuições e outros.

Com esta tabela e estes atributos o controle se torna muito mais eficiente e pratico, pois da para se ter uma noção de tempo de execução, controle de nomenclaturas repetidas, estados e outros já citados anteriormente.

A ideia para criar a tabela é simples, uma única função de nome criarTabela é responsável por preencher com regras de se encontrar um no com nome ":", ela cria uma nova tupla de tipo variavel, com o tipo do filho mais a esquerda e nome de filho mais a direita, e se achar a palavra "cabecalho"executa a mesma função mudando o tipo por funcao.

1.2. semântica TPP

Para encontrar os erros primeiro é necessário investigar quais são as regras semânticas da linguagem TPP, e ela possui algumas, sendo seus principais tópicos:

Funções e Procedimentos, neste tópico estão todas as regras que englobam funções e procedimentos da linguagem, dentre elas temos regras como, todo o código deve possuir uma função de nome principal, funções devem ter tipo e retorno condizentes com as declaradas, declarações devem possuir a mesma quantidade de parâmetros do que as chamadas de funções, funções devem ser declaradas antes de serem chamadas e outras.

Erros relacionados as Variáveis, que podem ser dos mais diversos como, variáveis com escopo errados, duplicatas de variáveis, tipos de atribuição errados, inversões de declarações e atribuição de retorno de uma função a uma variável. Além destes também temos um subtópico das variáveis que são os erros de coerção, este tipo de erro acontece quando uma variável é declarada de um tipo porém um dado de outro tipo é inserido nela, quando isto ocorre uma coerção é necessária, ou seja, o tipo que irá estar dentro da variável irá ser truncado ou terá seu escopo aumentado como exemplo podemos imaginar o seguinte escopo:

```
inteiro principal()
    inteiro: N
    N := 1.5
    retorna(0)
fim
```

Neste caso pode ser observado que a variável N é do tipo inteiro, todavia foi atribuída com um valor do tipo flutuante, e neste caso a seguinte mensagem deve ser exibida:

```
Coerção implícita do valor de N.
```

E por ultimo temos também os erros de arranjos, eles são acionados quando um vetor ou matriz são definidos com tipo flutuante como parâmetro, ou quando possuem valores de entradas maiores do que os declarados.

Possuindo então conhecimento sobre todos os possíveis tipos de erros que podem ocorrer na linguagem TPP, se faz necessário a codificação para que eles ocorram quando necessário. Para qualquer um dos erros a mesma lógica foi empregada, primeiramente a arvore foi percorrida tentando identificar um nome comum a todos que davam aquele erro, após encontrar este token a arvore era percorrida e com o auxilio da tabela de símbolos o erro era identificado e guardado em um vetor de erros, ao final de toda a execução se este vetor de erros possuir alguma coisa com nome erro, o código não esta apto para a geração de códigos, se ao final ele tiver coisas e forem apenas Warnings, ou se ele não tiver nada, então o código é passado para geração de códigos que será discutida nas próximas sessões.

Um exemplo de função de identificação de erros é a função chamadaPrincipal, que possui como intuito apenas verificar se em todo o código existe uma função com nome principal, e ela pode ser observado a seguir:

```
def chamadaPrincipal(no):
    for children in no.children:
        chamadaPrincipal(children)

    if(no.name == "chamada_funcao" and
    no.children[0].name == "principal"):
        error = {
            "erro": True,
            "tipo": "Erro:",
            "conteudo": "Chamada para a função principal
            não permitida",
            "flag": True
        }
        vetError.append(error)
```

nela podemos ver claramente cada uma das etapas de se encontrar um erro, a primeira parte é um for com chamada recursiva, ele é responsável por percorrer a arvore sempre passando nos filhos do nó estudado a baixo deste for temos uma condição de achada de Token, se meu Token for igual a chamada_funcao e este Token tiver um filho de nome principal, então houve uma chamada impropria para a função principal. Para relatar isto um uma variável error é criada, contendo todas as especificações necessárias daquele erro como, se é considerado um erro ou não, seu nome, o conteúdo a ser mostrado e uma Flag de controle. Ao final de tudo o erro é encapsulado e adicionado ao vetor de erros, que irá ser processado mais a frente.

1.3. Arvore Sintática Abstrata

E a ultima etapa do processo semântico é a poda da arvore gerada pela etapa sintática. Esta poda necessita ser realizada retirando todos os nós desnecessários e guardando apenas os que forem ser utilizados na semântica e geração de código. Sabendo disso podemos analizar a figura 1.

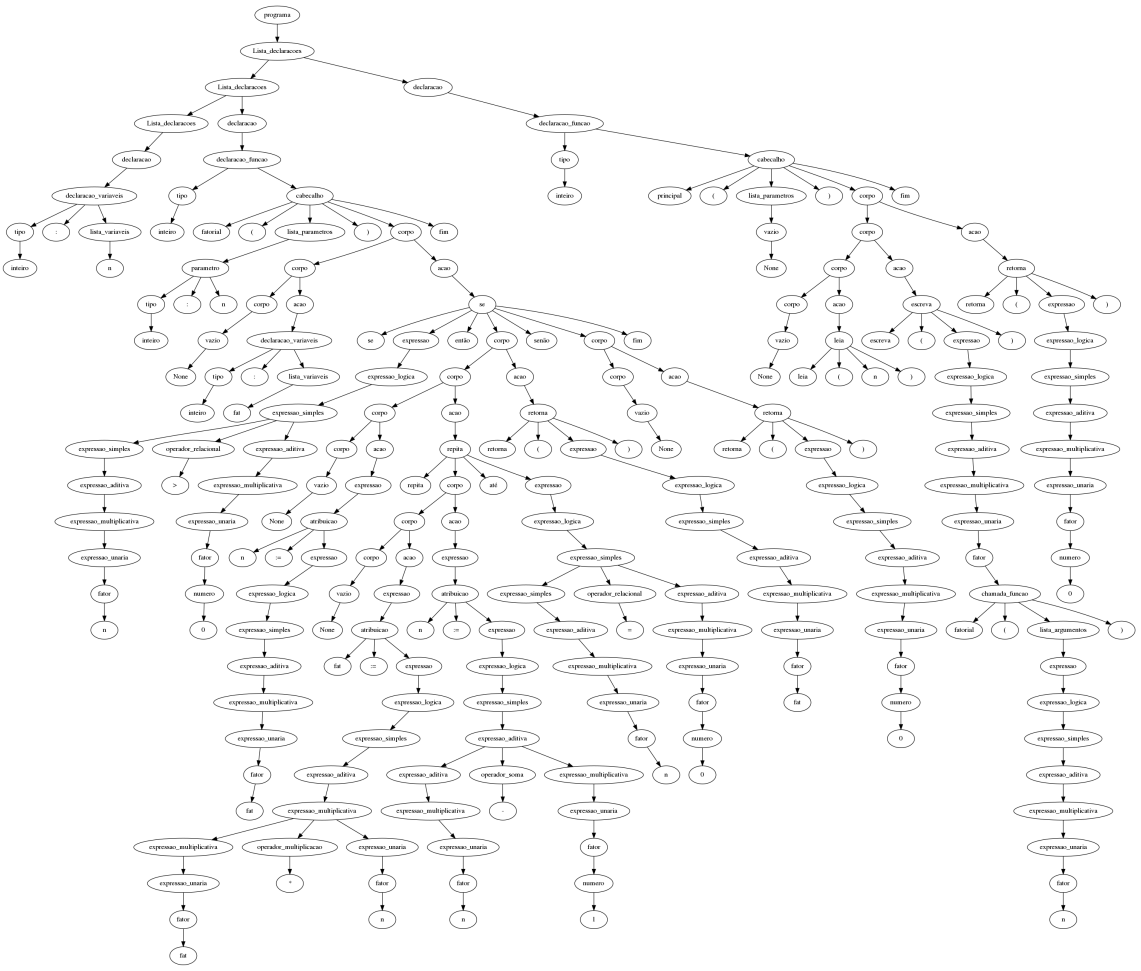

Figura 1. arvore de decisões gerada sem poda

Nela podemos observar uma extrema repetição de expressões, corpos, acoes, operadores e outras coisas que são de extrema importância na geração da arvore, porém que perdem o sentido quando necessitamos gerar um analisador semântico ou gerar código, pelo contrario, estas etapas intermediarias apenas atrasam todo o processo de verificação e cobertura da arvore. Então esta série de desvantagens que possuir uma arvore gerada muito densa e com informações desnecessárias faz necessária a poda da mesma para que além de tornar a passagem por ela mais rápido, irá tornar-la mais legível.

para realizar a poda, foram utilizados os mesmos princípios já descritos anteriormente neste relatório, primeiro percorre a arvore, buscando nós Tokens, e depois de encontra-lo retire ele passando seus filhos para seus pais, ou execute a ação desejada. O resultado desta busca e retirada será parecido com o da figura 2 que é a mesma arvore a figura 1 porém podada.

Nela é possível observar logo de cara que está bem menor que a original, as principais mudanças são que agora os nós folhas onde o programa realmente fica armazenado estão com muito mais destaque do que os anteriores, além também de que alguns nós precisavam de caminhos extensos e difíceis de serem percorridos, agora com a arvore reduzida esse caminho já não se faz mais necessário.

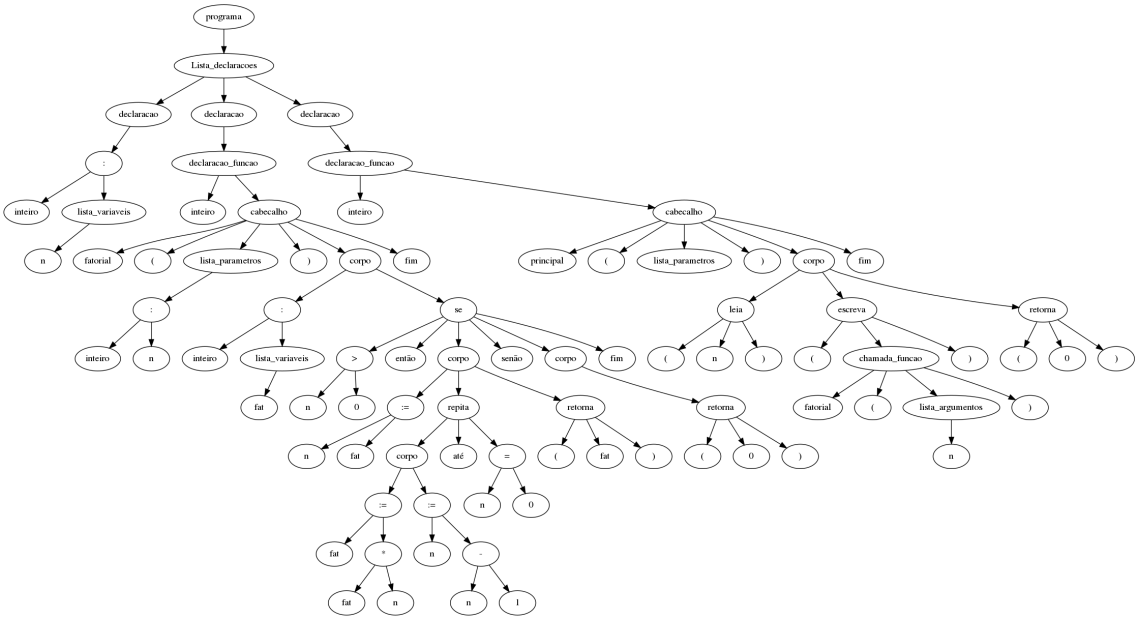

Figura 2. arvore de decisões gerada com poda

2. Conclusão

Desta forma com a semântica finalizada, a geração de código se torna possível, pois agora com a tabela de símbolos, a arvore gerada e qualquer erro eliminado, ira ser utilizado o LLVM para gerar código de assembly que poderá ser repassado para qualquer linguagem.

Referências

Beazley, D. M. Ply (python lex-yacc) documentation.