Trabalho de pesquisa

Felipe Theodoro
11718401 - felipe.lopes401@al.faj.br

Juan Domingues
11717541 - juan.domingos541@al.faj.br

Pedro Correa
11718563 - pedro.figueiredo563@al.faj.br

Rafael Vedovatto
11718375 - rafael.vedovatto375@al.faj.br

Ygor Gazola
11718340 - ygor.gazola340@al.faj.br

22 de agosto de 2020

1 Análise Léxica

É a primeira fase do processo de compilação, no qual tem o objetivo de identificar unidades léxicas que compõem o programa. O analisador léxico lê todos os caracteres do programa fonte e verifica se eles pertencem ao alfabeto da linguagem. Caso não exista, devea ser gerado um erro léxico. É nesta parte também em que ele remove todos os espaços em branco e comentários.

Após essa primeira etapa de verificação, ele deverá quebrar o texto em lexemas, este que é uma sequência de caracteres que são reconhecidos pelo compilador e que são esperados por ele, um exemplo são as palavras reservadas. Nesta etapa as expressões regulares possuem um mecanismo bastante importante para ajudar no reconhecimento e padronização dos lexemas.

Após o compilador reconhecer e mapear os lexemas, ele passa para a parte dos tokens, que são os lexemas indentificados de forma que seja mais facil a compreensão das palavras reservadas e variáveis que devem ser declaradas.

Os tokens podem ser divididos em dois grupos:

- **Tokens simples**: são tokens que não têm valor associado pois a classe do token já a descreve. Exemplo: palavras reservadas, operadores, delimitadores.
- **Tokens com argumento**: são tokens que têm valor associado e corresponde a elementos da linguagem definidos pelo programador. Exemplo: variáveis, contantes numéricas.

Um exemplo da estrutura de token seria o seguinte:
<nome-token, valor-atributo>

Onde o **nome do token** corresponde a classificação definida pelo compilador da linguagem, por exemplo: número, variáveis, constantes. E o **valor do atributo** corresponde a um valor qualquer que pode ser atribuído ao token, por exemplo o valor de entrada ou o valor recebido na hora da execução.

1.1 Exemplo de análise léxica

Para esclarecer melhor a ideia do que seria um token e como é o comportamento dele, vamos utilizá-lo no seguinte trecho de código:

*total = entrada * saida () + 2*

Assim, nesta declaração temos as classificações:

- *<id, 15>*: apontador 15 da tabela de símbolos e classe do token *id*.
- *<=, >*: operador de atribuição, sem necessidade de um valor para o atributo.
- *<id, 20>*: apontador 20 da tabela de símbolos e classe do token *id*.
- *<*, >*: Operador de multiplicação, sem necessidade de um valor para o atributo.

- *<id, 30>*: apontador 30 da tabela de símbolos e classe do token *id*.
- *<+, >*: operador de soma, sem necessidade de um valor para o atributo.
- *<(, >*: delimitador de função.
- *<), >*: delimitador de função.
- *<numero, 2>*: token numérico, com valor para o atributo 2 indicado o valor do número.

1.2 Erros léxicos

Esta etapa de análise léxica é muito inocente para identificar alguns erros de compilação que podem existir, um exemplo seria:

fi (a == '123')

Onde o analisador não ia identificar que a palavra *fi* deveria ser declarada *if*. Essa validação somente é possível ser feita na análise sintética.

2 Análise Sintática

Nesta parte é responsável por determinar se o programa de entrada representado pelo fluxo de tokens possui as sentenças válidas para a linguagem de programação.

Esse modelo pode ser definido utilizando gramáticas livres de contexto que representam uma gramática formal e pode ser escrita através de algoritmos que fazem a derivação de todas as possíveis construções da linguagem. Essas derivações tem como objetivo determinar se o fluxo de palavras fazem sentido na sintaxe da linguagem de programação.

2.1 Gramática Livre de Contexto - GLC

Em geral, quase todas as linguagens de programação pertencem a uma categoria chamada de Linguagens Livres de Contexto, estes que são a base para a construção de analisadores sintáticos. Eles são utilizados para especificar as regras sintáticas de uma linguagem de programação, uma linguagem regular pode ser reconhecida por um automato finito determinístico e não determinístico, já uma GLC pode ser reconhecida por um automato de pilha.

Uma gramática descreve naturalmente como é deve ser realizado as construções no programa. Um exemplo que podemos utilizar é este *if* na linguagem Pascal:

if (expression) then declaration else declaration ;

Essa mesma forma em GLC é expressada da seguinte maneira:

declaration →if (expression) then declaration else declaration ;

A definição formal de uma GLC pode ser representada através dos seguintes componentes:

G = (N, T, P, S)

Onde:

- N - Conjunto finito de símbolos não terminais.
- T - Conjunto finito de símbolos terminais.
- P - Conjunto de regras de produções.
- S - Símbolo inicial da gramática.

2.2 Derivações

Durante este etapa é aplicado a regra de produção para substituir cada símbolo não terminal por um símbolo terminal, permitindo identificar se certa cadeias de caracteres pertence a linguagem. O resultado deste processo é a árvore de derivação.

Tipos de derivação:

- Top-Dowm: Examina os símbolos terminais da esquerda para a direita, formando uma árvore sintática de cima para baixo.

- Bottom-Up: Examina os símbolos terminais da direita para a esquerda, formando uma árvore sintética de baixo para cima.

A utilização de qual algoritmo deve ser utilizado para realizar esta tarefa não possui tanta importancia, pois sempre no final deverá ter o mesmo resultado, caso a árvore final seja diferente temos uma ambiguidade.

2.3 Ambiguidade

Certas gramaticas permitem que a mesma sentença possua mais de uma árvore de derivação, tornando a gramática inadequada para a linguagem de programação pois o compilador não consegue definir a estrutura do programa fonte. Duas derivações podem gerar uma única árvore sintática, mas duas árvores sintáticas não podem ser geradas por uma derivação.

Uma ambiguidade pode ser evitada de duas formas:

1. Reescreven a gramática afim de remover a ambiguidade, podendo torná-la mais complexa.

2. Definir ordens de prioridade durante a derivação.

2.4 Geradores de analisadores sintáticos

Igual ao comportamento de construir analisadores léxicos, os analisadores sintáticos podem ser construídos através de ferramentas que auxiliam neste trabalho.

Basta um arquivo de configurações sintáticas e através de comandos é possível gerar um analisador sintático. A saída é um arquivo de código com a implementação do analisador sintático.

3 Análise Semântica

Esta etapa é responsável por verificar aspectos relacionados ao significado das instruções e também a validação sobre uma série de regras que não podem ser verificadas nas etapas anteriores.

As validações que não são realizadas durante as etapas anteriores são realizadas durante a análise semântica a fim de garantir que o programa fonte esteja coerente e o mesmo possa ser convertido para a linguagem de máquina. A análise semântica percorre a árvore sintática e relaciona os identificadores com sues dependentes de acordo com a estrutura hierárquica.

Esta etapa também é responsável pela captura de informações sobre o programa fonte para que as fases subsequentes gerem o código objeto, outro fator importante desta fase é a validação de tipos, nela o compilador verifica se cada operador recebe os operandos permitidos e especificados na linguagem fonte.

Os tipos de dados são muito importantes nessa etapa da compilação, pois eles são os responsáveis por dar características para as linguagens de programação. Com base nos tipos, o analisado semântico pode definir quais valores podem ser manipulados, essa ação é conhecida como *type checking*.

3.1 Inferência de tipos

O sistema de tipos de dados podem ser divididos em dois grupos: sistemas dinâmicos e sistemas estáticos. Muitas das linguagens utilizam o sistema estático pois essa informação é utilizada durante a compilação e simplifica o trabalho do compilador. Esse sistema é muito predominante em linguagens compiladas.

- **Sistema de tipo estático**: obrigam o programador definir os tipos das variáveis e retorno de funções, exemplos de linguagens: C, Java, Pascal.
- **Sistema de tipo dinâmico**: variáveis e retorno de funções não possuem declaração de tipos, exemplos de linguagens: Python, PHP, Javascript.

Algumas linguagens utilizam um mecanismo muito interessante chamado inferncia de tipos, que permite a uma variável assumir vários tipos durante o seu ciclo de vida, permitindo que ela consiga ter vários valores. Nesses casos o compilador infere o tipo da variável em tempo de execução, esse mecanismo esta diretamente relacionado ao mecanismo de *Generics* do Java. Linguagens de programação como o Haskel utilizam esse método.