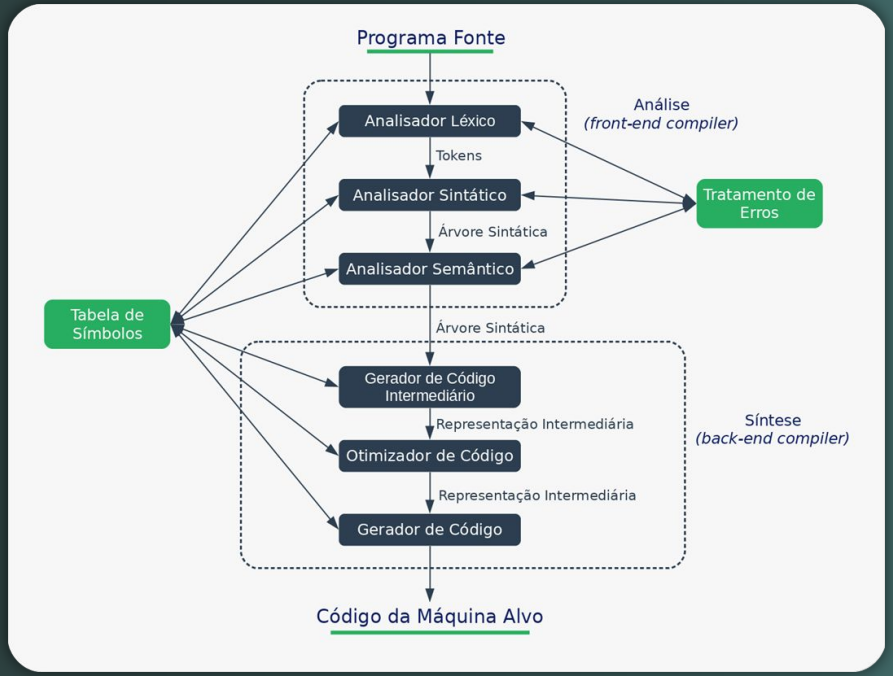
Linguagens de Programação

Compiladores: Análise Semântica

Davi Ventura C. Perdigão
Eric Henrique de C. Chaves

01 Introdução

A análise semântica é responsável por verificar aspectos relacionados ao significado das instruções, e nesse momento ocorre a validação de uma série de regras que não podem ser verificadas nas etapas anteriores, também trata a entrada sintática e transforma-a numa representação mais simples e mais adaptada à geração de código.



Não é possível representar com expressões regulares ou com uma gramática livre de contexto regras como: **todo identificador deve ser declarado antes de ser usado.** As validações que não podem ser executadas pelas etapas anteriores devem ser executadas durante a análise semântica a fim de garantir que o programa fonte esteja coerente e o mesmo possa ser convertido para linguagem de máquina. O analisador semântico utiliza a **árvore sintática** e a **tabela de símbolos** para fazer as análises semânticas, relacionando os identificadores com seus dependentes de acordo com a estrutura hierárquica.

É importante ressaltar que muitos dos erros semânticos tem origem de regras dependentes da linguagem de programação.

Um importante componente da análise semântica é a verificação de tipos, nela o compilador verifica se cada operador recebe os operandos permitidos e especificados na linguagem fonte.

Vamos utilizar como exemplo a regra de atribuição:
i := a + b;
O identificador i foi declarado?
O identificador i é uma variável?
Qual o escopo da variável i?
Qual é o tipo da variável i?
O tipo da variável i é compatível com os demais identificadores, operadores?

02

Funções e Tarefas

Funções de Análise Semântica:

1. **Verificação de tipo -**
garante que os tipos de dados sejam usados de maneira consistente com sua definição.
2. **Verificação de rótulos -**
um programa deve conter referências de rótulos.
3. **Verificação de controle de fluxo -**
mantém uma verificação de que as estruturas de controle são usadas de maneira adequada. Exemplo: nenhuma instrução de interrupção fora de um loop.

Análise de contexto com geração de código

Verificação de erros em frases sintaticamente corretas

A fase de geração de código intermediário permite a geração de instruções para uma máquina abstrata.

A fase de otimização analisa o código no formato intermediário e tenta melhorá-lo de tal forma que venha a resultar um código de máquina mais rápido em tempo de execução, usando as réguas que denotam a semântica da linguagem-fonte.

E por fim, a **fase de geração de código** tem como objetivo analisar o código já otimizado e gerar um código objeto definitivo para uma máquina alvo.

03

Inferência de Tipos

Sistemas 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, esse sistema é predominante em linguagens compiladas, pois essa informação é utilizada durante a compilação e simplifica o trabalho do compilador.

Estaticos

Linguagens como C, Java, Pascal obrigam o programador a definir os tipos das variáveis e retorno de funções, o compilador pode fazer várias checagens de tipo em tempo de compilação.

Dinâmico

Variáveis e retorno de funções não possuem declaração de tipos, como exemplo temos linguagens como Python , Perl e PHP.

Algumas linguagens utilizam um mecanismo muito interessante chamado inferência de tipos, que permite a uma variável assumir vários tipos durante o seu ciclo de vida, isso permite que ela possa ter vários valores. Linguagens de programação como Haskel tira proveito desse mecanismo. Nesses casos o compilador infere o tipo da variável em tempo de execução, esse tipo de mecanismo está diretamente relacionado ao mecanismo de Generics do Java e Delphi Language. A validação de tipos passa a ser realizada em tempo de execução.

Haskel

Delphi

Java

04

Principais Erros Semânticos

Escopo dos Identificadores -

O compilador deve garantir que variáveis e funções estejam declaradas em locais que podem ser acessados onde esses identificadores estão sendo utilizados.

Uma classe com funções declaradas como privadas e essas funções sendo utilizadas fora da classe:

```
class Foo {
   private $x;

   private function sum() {
        return $this->x = 1;
   }

}

Foo().sum()
```

A variável contador deve ser declarada em cada escopo onde está sendo utilizada:

```
def foo():
    cont = 0

    def bar():
        cont = 1

    def dop():
        cont = 3
    bar()
    dop()
    print(cont)
```

Compatibilidade de Tipos -

Verificar se os tipos de dados declarados nas variáveis e funções estão sendo utilizados e atribuídas corretamente, por exemplo: operações matemáticas devem ser realizadas com números, atribuições de valores para variáveis.

Variável declarada como int está recebendo um valor string:

```
var i : int;
var s : string;
s = "john";
i := f;
```

Em alguns casos pode ser efetuada a conversão de tipos, essa operação é conhecida como cast, e pode ser feito de forma explícita no código ou pelo próprio compilador.

Detectar Unicidade de nomes de identificadores -

Essa verificação é muito importante pois ele deve garantir que os identificadores sejam únicos, não havendo na tabela de símbolos uma entrada para o mesmo identificador, isso implica em:

- **Diferentes identificadores podem ter o mesmo nome;**
- **Identificadores pode estar em um escopo diferente;**
- **Sobrecarga de métodos deve gerar novas entradas na tabela de símbolos;**

```
var i : int;
var i : int64;

var foo int

func foo() {
    fmt.Println(math.pi)
}
```

Uso correto de comandos de controle de fluxo -

Comandos como **continue** e **break** executam saltos na execução de códigos, esses comandos devem ser utilizados em instruções que permitem os saltos.

Obrigado!

Alguma Pergunta?