Análise Semântica

Eduardo Ferreira dos Santos

Ciência da Computação
Centro Universitário de Brasília – UniCEUB

Maio, 2017

Sumário

Fases

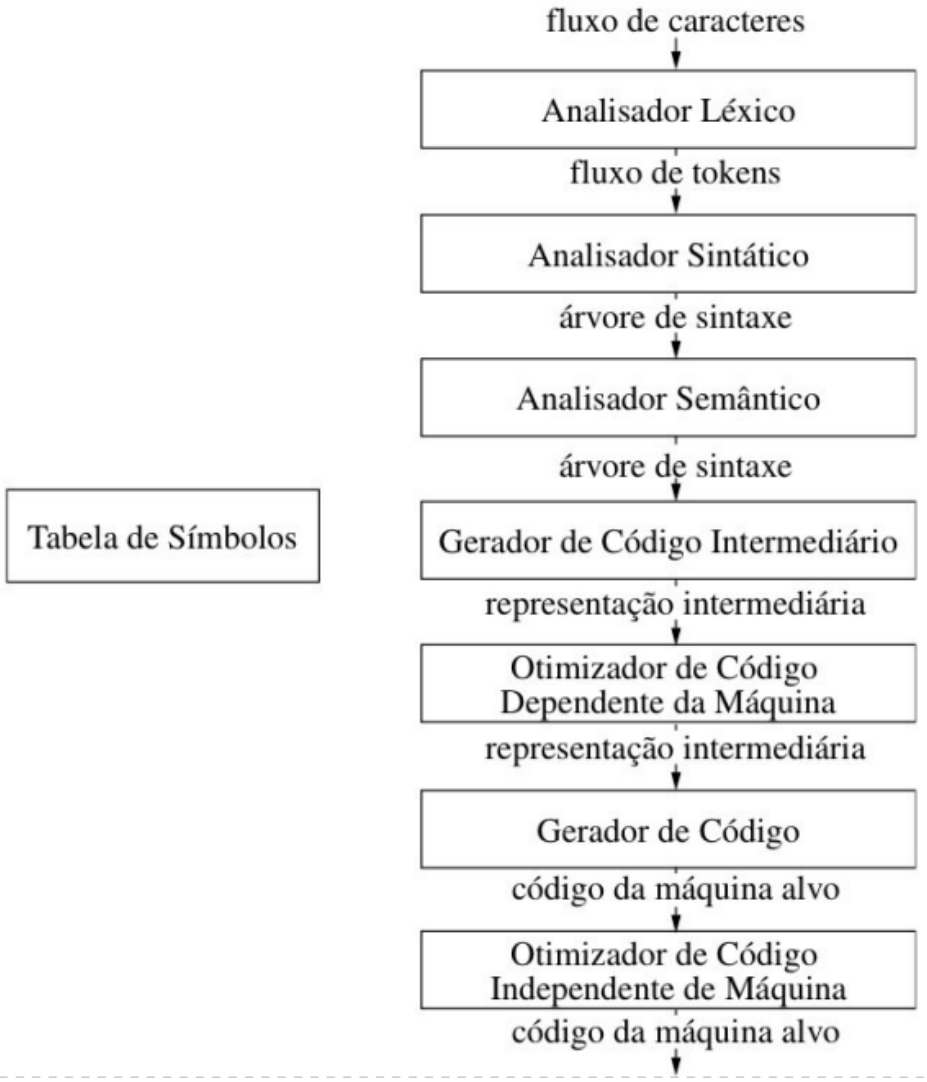

Figura 1.1: Fases do compilador [Aho et al., 2007]

Árvore de sintaxe

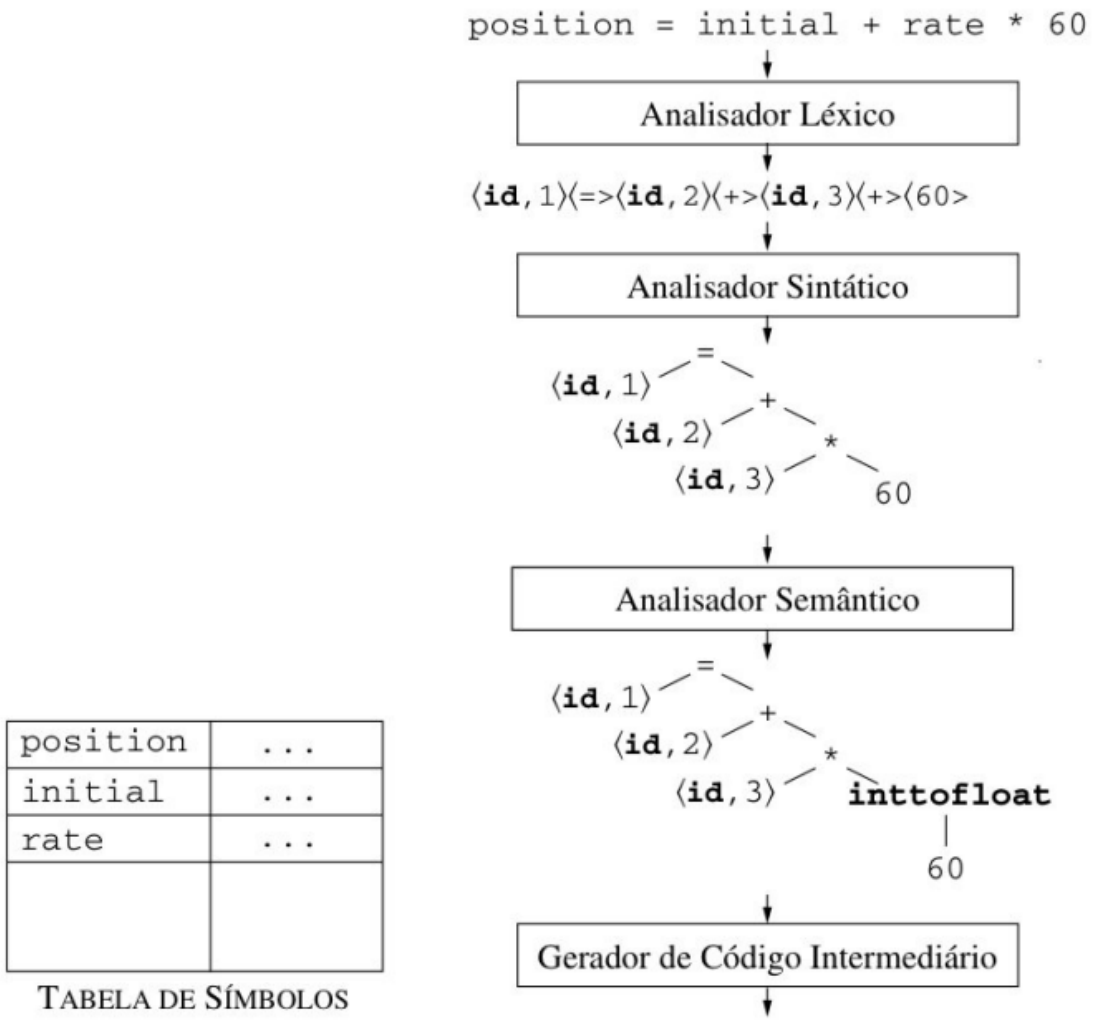

Figura 1.2: Modelo de tradução e atribuição [Aho et al., 2007]

Introdução

- O papel da análise semântica é produzir uma lista de tarefas;
- Para produzir a lista de tarefas o compilador vai precisar da lista de símbolos;
- Nesse momento também se faz a verificação de tipos;
- Objetivo: facilitar a tradução da lista de tarefas em linguagem de máquina.

Etapas da compilação

Análise léxica Detecta entradas com caracteres inválidos;

Análise sintática Produz a árvore de parsing e verifica erros de formação da árvore;

Análise semântica Última fase do "*front end*", detecta os erros que ainda podem existir.

Justificativa

- Alguns erros não são detectados pelo *parsing*;
- A análise semântica pode realizar várias verificações [Schwarz et al., 2016]:
  1. Todos os identificadores estão declarados;
  2. Os tipos são consistentes;
  3. Relações de herança;
  4. As classes são unicamente identificadas;
  5. Os métodos em uma classe são definidos apenas uma vez;
  6. As palavras reservadas não estão sendo utilizadas de maneira equivocada;
  7. ...
- Os requisitos semânticos dependem da linguagem.

Visão geral de Decaf

- Princípios de *design*:
  - Implementação rápida;
  - Linguagem operativa;
  - Fortemente tipada;
  - Utilização de:
    1. Tipos de dados primitivos;
    2. Tipos estruturados: *Classes* e *arrays* unidimensionais;
    3. Atribuição;
    4. Condicionais: `if/else`;
    5. Iteradores: `while`;
    6. Métodos e chamadas de método;
    7. Aritmética: `+, -, *, /, %`;
    8. Operadores relacionais: `<, >, <=, >=, ==, !=`;
    9. Operadores booleanos: `&&, ||, !`;
    10. Entrada/saída em inteiros: `print, read`.
  - Pela simplicidade, alguns outros conceitos são deixados de fora.

Exemplo simples

- Os programas em *Decaf* possuem uma estrutura parecida com a linguagem Java;
  - Uma classe especial chamada `Program` com um método especial chamado `main`;
- `classe` = marcador de início de programa;
- `main` = função principal chamada na execução do código.

Listing 1: Exemplo de programa vazio em Decaf

```
class Program
{
    void main()
    {
    }
}
```

Chamadas em Decaf

- As funções são chamadas utilizando a instrução `callout`;
- Serve apenas para chamadas de funções internas (bibliotecas) da linguagem.

Listing 2: Chamada da função print para um Hello World em Decaf [Amarasinghe and Rinard, 2010]

```
class Program
{
    void main()
    {
        callout("printf", "Hello, World.\n");
    }
}
```

Expressões

- A linguagem permite a utilização de expressões

Listing 3: Expressões e operadores em Decaf [Amarasinghe and Rinard, 2010]

```
class Program
{
    void main()
    {
        int a, b, c;
        int d, e;

        a = 10;
        b = 20;
        c = 30;

        d = (a + b);
        e = (c * 3);

        e = d * e - 100;

        callout("printf", "%d %d\n", d, e);
    }
}
```

Funções

- É possível definir funções e misturar com chamadas do tipo `callout`;
- Decaf também permite chamadas recursivas.

Listing 4: Recursão, funções e `callout` [Schwarz et al., 2016]

```
class Program {
    int factorial(int n)
    {
        if (n <= 1) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }

    void main()
    {
        int res;

        res = factorial(10);

        callout("printf", "10! = %d (3628800)\n", res);
    }
}
```

Escopo

- Definição: verifica a compatibilidade entre a declaração e a utilização dos identificadores;
- Importante etapa de análise estática na maior parte das linguagens;
- O escopo do identificador é a parte do programa onde o identificador está acessível;
- O mesmo identificador pode mudar de significado a depender da parte do programa;
- O identificador pode ter escopo restrito.

Estático versus dinâmico

- A maior parte das linguagens possui escopo estático:
  - O escopo depende apenas do texto do programa e não da execução;
  - C, C++ e Javascript possuem escopo estático;
- Outras linguagens possuem escopo dinâmico [Schwarz et al., 2016]:
  - Lisp, SNOBOL;
  - Lisp agora possui escopo dinâmico;
  - O escopo depende da execução do programa.

Escopo estático

Listing 5: O valor de a se refere à definição mais próxima

```
class Program
{
    int a = 1;
    int b, c;

    void main()
    {
        a = 10;
        b = 5;

        c = a + b;

        callout("printf", "%d + %d = %d (15)\n", a, b, c)
            ;
    }
}
```

Escopo dinâmico

Listing 6: O valor de c é calculado no momento da execução

```
class Program
{
    int a = 1;
    int b, c;

    void main()
    {
        a = 10;
        b = 5;

        c = a + b;

        callout("printf", "%d + %d = %d (15)\n", a, b, c)
            ;
    }
}
```

Exemplo comparativo

Static scoping

```
1  const int b = 5;
2  int foo()
3  {
4     int a = b + 5;
5     return a;
6  }
7
8  int bar()
9  {
10    int b = 2;
11    return foo();
12 }
13
14 int main()
15 {
16    foo(); // returns 10
17    bar(); // returns 10
18    return 0;
19 }
```

Dynamic scoping

```
1  const int b = 5;
2  int foo()
3  {
4     int a = b + 5;
5     return a;
6  }
7
8  int bar()
9  {
10    int b = 2;
11    return foo();
12 }
13
14 int main()
15 {
16    foo(); // returns 10
17    bar(); // returns 7
18    return 0;
19 }
```

Figura 3.1: Exemplo de escopo estático versus dinâmico na linguagem C [1]

[1] Fonte: https://msujaws.wordpress.com/2011/05/03/static-vs-dynamic-scoping/

Escopo em Decaf

- As regras de escopo em Decaf são **simples** e **restritivas**:
  - Uma variável deve ser declarada antes de ser utilizada;
  - Um método só pode ser chamado pelo código que aparece após sua declaração;
  - Métodos recursivos são permitidos.

Escopos locais e globais

- Pelo menos dois escopos podem ser acessados a qualquer momento em Decaf:

  Escopo global Nomes dos campos ⟨*fields*⟩ e métodos introduzido na classe especial `Program`

  Escopo do método Nomes de variáveis e parâmetros formais introduzidos na declaração do método.

- Algumas regras específicas:
  - Escopos locais existem em cada bloco ⟨*block*⟩ do código;
  - Podem existir depois de `if` e `for`, ou em qualquer lugar onde ⟨*statement*⟩ é legal;
  - Escopo de método pode "esconder" valores declarados no escopo global.

Exemplo de uso

Listing 7: Utilização de escopo global e de método

```
class Program {
    int c = 1;
    int add(int a, int b)
    {
        return (a + b);
    }

    void main() {
        int d = 2;
        int e;
        e = add(c, d);
        callout("printf", "A soma é %d", e);
    }
}
```

Tabela de símbolos

- Considere a seguinte expressão em Decaf: `x = e = 0;`
- Ideia [Schwarz et al., 2016]:
  - Antes de processar e adicione a definição de x ao conjunto de definições atual, sobrescrevendo qualquer atribuição anterior de x;
  - Navegue na árvore de parsing recursivamente;
  - Depois de processar e, remova a definição de x e restaure o valor anterior.
- A tabela de símbolos é uma estrutura de dados que armazena as atribuições de valor mais atuais para os identificadores.

Implementação simples

- Estrutura de dados utilizada: pilha;
- Operações:

  add_symbol(x) Executa uma operação de *push* na pilha para x, com todas as informações associadas;

  find_symbol(x) Busca na pilha, de cima para baixo, o valor de x. Retorna `NULL` se não for encontrado;

  remove_symbol() Executa uma operação de *pop* na pilha.

Limitações

- A tabela de símbolos funciona bem para o operador $=$:
  - Os símbolos são adicionados individualmente de maneira sequencial;
  - As declarações estão perfeitamente aninhadas.
- E se a ordem das atribuições for diferente?

Tabela de símbolos mais completa

enter_scope() Inicia um novo escopo aninhado;

find_symbol(x) Encontra o valor atual de x (ou `NULL`);

add_symbol(x) Adiciona o símbolo x na tabela;

check_scope(x) Verdadeira se x estiver definido no escopo atual;

exit_scope() Executa uma operação de *pop* na pilha.

Definições de método

- Os nomes de método podem ser utilizados antes de ser definidos;
- Não é possível verificar os nomes dos métodos:
  - Utilizando uma tabela de símbolos;
  - Em uma única passagem pela árvore.
- Solução:

  Passo 1 Reúna todos os nomes de método;

  Passo 2 Faça a checagem.
- A análise semântica requer múltiplas passagens pela árvore.

Tipos

- O que é um tipo?
- A noção varia de acordo com a linguagem;
- Consenso:
  - Um conjunto de valores;
  - Um conjunto de operações com esses valores.
- As classes são uma instância da definição moderna de tipo [Schwarz et al., 2016].

Por que tipos?

Listing 8: Quais os tipos de dado para $r1 $r2 e $r3? [Schwarz et al., 2016]

```
add $r1, $r2, $r3
```

Tipos e operações

- Cada tipo de dado possui um conjunto válido de operações;
  - Não faz sentido executar uma operação de adição entre um ponteiro e um inteiro na linguagem C;
  - Faz sentido a adição de dois inteiros;
  - Contudo, ambos executam a mesma operação em assembly.
- O sistema de tipos da linguagem especifica os tipos e suas operações permitidas;
- O objetivo da verificação de tipos é garantir que as operações sejam realizadas com os tipos de dado corretos.
  - É a única etapa que realiza interpretação de valores.;
  - Para o computador, tudo se resume a 0 e 1.

Verificação de tipos

- Existem três tipos de linguagens:

  Statically typed Verificação estática de tipos. Quase toda a validação de tipos é feita no momento da compilação (C, Java, Cobol);

  Dynamically typed Verificação dinâmica de tipos. Quase toda a validação de tipos é feita no momento da execução do programa (Scheme, Python);

  Untyped Não faz verificação de tipos (código de máquina).

Necessidade de verificação

- Existem visões concorrentes em relação às verificações estática e dinâmica;
- Os defensores da verificação estática dizem:
  - A verificação estática encontra muitos problemas no momento da compilação;
  - Evita o *overhead* da verificação de tipo em tempo de execução.
- Os defensores da verificação dinâmica dizem:
  - Os sistemas de verificação estática são muito restritivos;
  - Não é possível realizar a prototipagem rápida em sistemas de verificação estática.

Conclusões sobre tipos

- Na prática, a maior parte do código é escrita em linguagens com verificação estática de tipo com algum tipo de mecanismo de "escape";
  - Ex.: Unsafe casts in C, Java
- É discutível saber se essa decisão representa o melhor dos dois mundos.

Verificação e inferência

- A verificação de tipos é o processo de verificar programas onde a tipagem é forte;
- A inferência de tipos é o processo de descobrir a informação de tipo que está faltando;
- Os processos são diferentes mas os termos são intercambiáveis.
- Utilização de regras de inferência.

Regras de inferência

- Vimos dois exemplos de notação formal especificando partes do compilador:
  - Expressões regulares;
  - Gramáticas livres de contexto.
- Para a verificação de tipos, o formalismo necessário é a utilização de regras lógicas de inferência.

Justificativa

- Forma das regras de inferência:

  SE a hipótese é verdadeira, ENTÃO a conclusão é verdadeira

- A verificação de tipos utiliza a mesma forma:

  SE E_1 e E_2 são de um tipo, ENTÃO E_3 é do mesmo tipo

- Regras de inferência são uma maneira reduzida de escrever expressões do tipo IF-THEN.

Criando regras de inferência

- Inicie com uma notação simplificada e adicione novas funcionalidades à medida que forem necessárias.
- Elementos:
  - Símbolo ^ significa and;
  - Símbolo $\Rightarrow$ significa if-then;
  - $x : T$ significa x tem tipo T.

Criando regras de inferência II

SE e_1 é do tipo *Int* and e_2 é do tipo *Int*, ENTÃO $e_1 + e_2$ é do tipo *Int*

Criando regras de inferência II

SE e_1 é do tipo *Int* and e_2 é do tipo *Int*, ENTÃO $e_1 + e_2$ é do tipo *Int*

(e_1 tem tipo *Int*^e_2 tem tipo *Int*) $\Rightarrow$ $e_1 + e_2$ tem tipo *Int*

Criando regras de inferência II

SE e_1 é do tipo *Int* and e_2 é do tipo *Int*, ENTÃO $e_1 + e_2$ é do tipo *Int*

$(e_1$ tem tipo $Int \hat{} e_2$ tem tipo $Int) \Rightarrow e_1 + e_2$ tem tipo *Int*

$(e_1 : Int \hat{} e_2 : Int) \Rightarrow e_1 + e_2 : Int$

Criando regras de inferência III

A expressão

$$(e_1 : Int \hat{} e_2 : Int) \Rightarrow e_1 + e_2 : Int$$

é um caso especial de:

$$Hypothesis_1 \hat{} ... \hat{} Hypothesis_n \Rightarrow Conclusion$$

Trata-se então de uma regra de inferência.

Notação

- Por tradição as regras de inferência são escritas no seguinte formato:

$$\frac{Hypothesis_1 \hat{} ... \hat{} Hypothesis_n}{Conclusion}$$

- Na linguagem Cool as regras de tipo possuem hipóteses e conclusões:

$$\vdash e : T$$

- O símbolo $\vdash$ significa está provado que.

Duas regras

$$\frac{\text{i é um inteiro}}{\vdash i : Int} \quad \text{(Int)}$$

$$\frac{\vdash e_1 : Int \quad \vdash e_2 : Int}{\vdash e_1 + e_2 : Int} \quad \text{(Add)}$$

- As regras fornecem *templates* para descrever os tipos inteiro e expressões de soma $(+)$;
- Preenchendo os *templates* é possível produzir definições completas de tipo para as expressões.

Exemplos

$$\frac{\dfrac{\text{1 é um inteiro}}{\vdash 1 : Int} \quad \dfrac{\text{2 é um inteiro}}{\vdash 2 : Int}}{\vdash 1 + 2 : Int} \qquad (1+2)$$

$$\frac{}{\vdash false : Bool} \qquad (Bool)$$

$$\frac{\text{s é uma constante do tipo string}}{\vdash s : string} \qquad (String)$$

Problema

- Qual é o tipo de uma variável de referência?

$$\frac{\text{x é um identificador}}{\vdash x :?} \quad \text{(Var)}$$

- A regra estrutural local não contém informação suficiente para fornecer um tipo a x;
- Como resolver?

Solução

- Solução: adicionar informações às regras de inferência;
- Um ambiente de tipos (*type enviroment*) fornece os tipos para variáveis livres:
  - Um ambiente de tipos é uma função de ObjectIdentifiers para Types;
  - Uma variável está livre numa expressão se não for definida na expressão.

OBRIGADO!!!
PERGUNTAS???


Aho, A., Lam, M., Sethi, R., and Ullman, J. (2007).
Compiladores–Princıpios Técnicas e Ferramentas.
Pearson, 2a. edition.


Amarasinghe, S. and Rinard, M. (2010).
Computer language engineering.
Disponível em `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/` Acessado em 02/08/2016.


Schwarz, K., Papadakis, H., and Mittal, R. (2016).
Compilers.
Disponível em `http://web.stanford.edu/class/cs143/` Acessado em 30/09/2016.