Compiladores

Análise Semântica

Introdução

Como já visto, a primeira etapa do *front-end* de um tradutor é a análise léxica, na qual o *scanner* acessa o código fonte e gera *tokens* a serem usados como entrada pelo *parser*. A partir desses *tokens*, o *parser* gera uma árvore sintática, a ser usada como entrada na terceira e última etapa de análise: a análise semântica. Estas notas de aula abordam as ações do compilador a partir deste ponto.

A Figura 1 resume a explicação do parágrafo anterior com um trecho de código como exemplo.

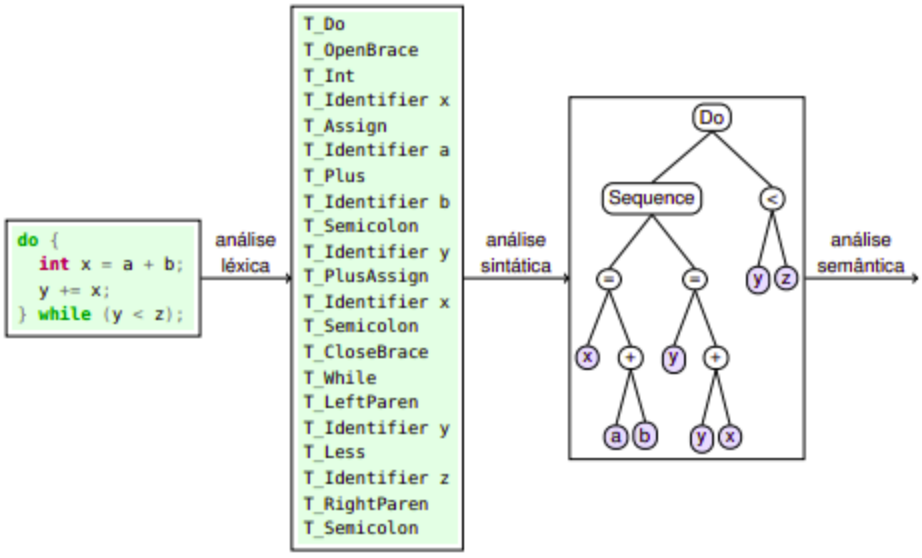

Figura 1. Transformação do código fonte em árvore sintática. Fonte: Malaquias, 2012.

Análise Semântica

A Análise Semântica é a última análise realizada pelo compilador (última etapa do *front-end*). O analisador semântico usa a árvore sintática e as informações na tabela de símbolos para verificar a consistência semântica do programa fonte com a definição da linguagem (AHO *et al*., 2007).

Nesta etapa, portanto, se verifica o sentido (significado) do código já analisado sintaticamente, presente na árvore sintática. Além disso, informações de tipo são coletadas e salvas na árvore de sintaxe ou na tabela de símbolos, para uso posterior durante a geração de código intermediário (AHO *et al*., 2007). Isso significa que a saída da etapa de análise semântica é uma árvore sintática corretamente tipada em relação às regras de tipo da linguagem, como mostra a Figura 2.

Posteriormente, a árvore sintática tipada é usada como entrada para a geração de código intermediário, que ainda pode ser otimizado antes da geração de código objeto (que também pode ser otimizado), como mostra a Figura 3.

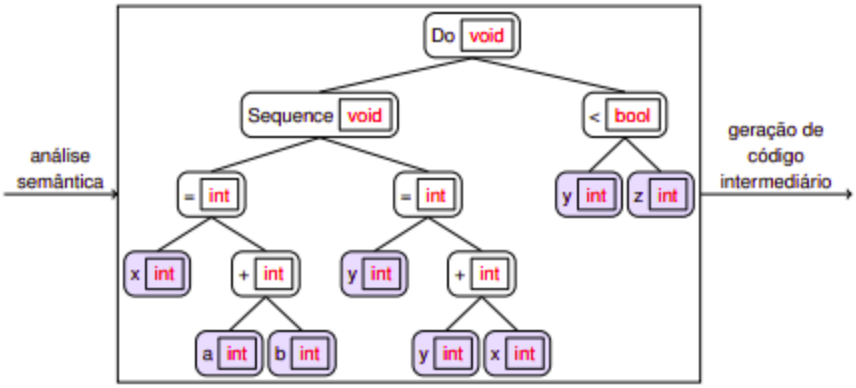

Figura 2. Árvore sintática tipada. Fonte: Malaquias, 2012.

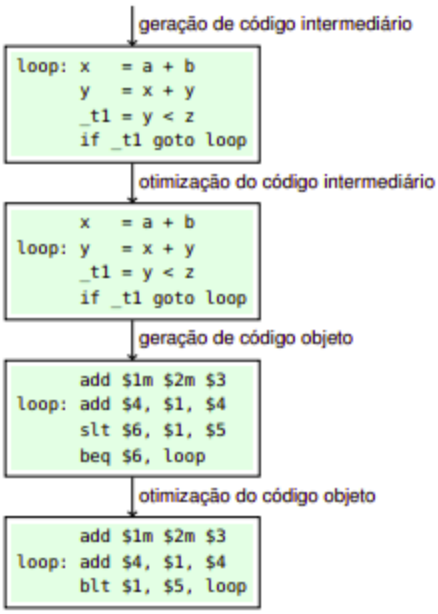

Figura 3. Ações do compilador após a análise semântica. Fonte: Malaquias, 2012.

A árvore sintática tipada pode ser anotada com algumas regras de correção de tipos. Um exemplo de uma regra de tipo, utilizando a semântica operacional:

$$\frac{\texttt{B:BOOL EXP1:COMMAND EXP2:COMMAND}}{\texttt{IF B THEN EXP1 ELSE EXP2 : COMMAND}}$$

Uma linguagem tipada define regras de análise de tipos para cada categoria de expressões que podem aparecer nos programas.

Exemplos:

$$\frac{\texttt{B1:BOOL B2:BOOL}}{\texttt{B1\^{}B2:BOOL}} \qquad \frac{}{\texttt{TRUE:BOOL}} \qquad \frac{\texttt{E1:INT E2:INT}}{\texttt{E1+E1:INT}}$$

Os compiladores geralmente usam a semântica operacional[1] na análise semântica. A semântica operacional

- fornece uma descrição precisa e formal do comportamento de uma linguagem de programação;
- descreve como as expressões e instruções da linguagem devem ser avaliadas e executadas;
- pode ser realizada de forma estruturada (*small-step*) ou natural (*big-step*).

SAIBA MAIS

Seria necessária uma disciplina para avançar nos aspectos semânticos de teoria das linguagens de programação, incluindo o aprofundamento nas semânticas operacional, denotacional, axiomática e de programação concorrente.

Recomenda-se as leituras de **Cavalcanti (s.d.)** e **Pellegrini (2019)** para avançar os estudos neste assunto.

O analisador semântico analisa a consistência dos tipos (*type checking*), por exemplo, verificando se um operador recebeu operandos do tipo correto. Considerando a linguagem C,

- `"A" - "B"` não é uma operação válida semanticamente.
- `3.09 - 3.1415` é uma operação válida semanticamente.
- `arr["x"]` não é uma operação válida semanticamente.

A análise semântica é uma **análise sensível ao contexto**, diferentemente das análises léxica e sintática, que são livres de contexto.

[1] De acordo com Louden (2013), semântica operacional é uma abordagem para definir o significado de uma linguagem de programação através da especificação de um conjunto de regras de transição que descrevem como as expressões e instruções da linguagem são avaliadas e executadas em uma máquina abstrata.

Erros

O erro no código a seguir, escrito em C, é léxico, sintático ou semântico?

```
#include <stdio.h>
void main() {
    printf("%d", "a" * 6);
}
```

O erro no código a seguir, escrito em C, é léxico, sintático ou semântico?

```
#include <stdio.h>
void main() {
    int a = 4, b = 2;
    a / b = 10;
}
```

O compilador, nesta etapa, identifica os erros não capturados nas etapas anteriores. Ele deve relatar erros semânticos como

- Erros de tipo: parâmetro informado de tipo diferente do parâmetro declarado;
- Usar um número de ponto flutuante para indexar uma matriz.
- Variável utilizada e não declarada;
- Variável redeclarada;
- Atributo privado utilizado fora do escopo da classe;
- Erro de verificação de fluxo de controle;

```
#include <stdio.h>
void foo() {
  if (1)
    break;
  else
    continue;
}
```

```
void main() { /* (...) */ }
```

Neste exemplo, adaptado de Ricarte (2003), as mensagens de erro informam que o comando `break` não está em um laço de repetição ou `switch`, e o `continue` não está em um laço de repetição.

Conflito de tipos para variáveis de mesmo nome declaradas duas vezes:

```
#include <stdio.h>
void main() {
    int i;
    char i;
    /* (...) */
}
```

Redeclaração de variáveis:

```
#include <stdio.h>
void main() {
    int i;
    int i;
    /* (...) */
}
```

CURIOSIDADE

O código abaixo funciona?

```
#include <stdio.h>
int valor = 13;
int valor;
void main() {
  printf("%d", valor);
```

```
}
```

E trocando a segunda declaração de variável para "`int valor = 14;`"?

Uma variável local não pode ser redeclarada. Porém, isto funciona para a variável global porque o compilador trata esta situação como uma *declaração de encaminhamento* (*forward declaration*) e as variáveis globais são armazenadas na seção de dados. A seção de dados permanece inalterada durante a execução do programa. As variáveis locais são armazenadas na pilha, assim como os parâmetros. Observa-se ainda que, ao atribuir o valor `14` à variável global, ela não funciona.

Não há diferença porque o compilador acaba declarando primeiro `int valor;` de qualquer forma. O compilador faz este e vários outros ajustes, por exemplo, alinhamento de memória, quando isso for conveniente.

Uma declaração de encaminhamento acontece quando a variável global é declarada e ainda não se conhece seus valores. Mesmo assim ela pode ser referenciada. Isto também funciona com estruturas (`struct`), por exemplo:

```
#include <stdio.h>
struct a {
    struct b *bPtr;
    int c;
};
struct b {
    struct a *aPtr;
    float d;
};
void main() {
  printf("1");
}
```

Outro exemplo de erro semântico:

```
#include <stdio.h>
int main() {
    int arr[10] = {0}, i = 0;
    while (i <= 10) {
        printf("%i: %i\n", i, arr[i]);
        i++;
    }
    return 0;
}
```

Coerção

A especificação da linguagem pode permitir algumas conversões de tipo chamadas ***coerções***. As coerções são as conversões implícitas e esperadas pelo programador. Conforme Aho *et al*. (2007), um exemplo é o operador aritmético binário poder ser aplicado a um par de inteiros ou a um par de números de ponto flutuante. Se o operador for aplicado a um número de ponto flutuante e um inteiro, o compilador pode converter ou coagir o inteiro em um número de ponto flutuante.

Exemplo em C:

```
#include <stdio.h>
void main() {
    int a = 4;
    float b = 2.0;
    printf("%f", a + b);
}
```

A coerção aparece no código acima, pois `a` tem tipo inteiro e `b` tem tipo numérico de ponto flutuante. Como o comando `printf` formata a saída como `float`, mas recebe uma operação de soma entre `int` e `float`, o verificador de tipos no analisador semântico descobre que o operador é aplicado a um `float` e a um `int` e, nesse caso, o inteiro pode ser convertido em

um número de ponto flutuante.

Conforme Ricarte (2003), a seguinte seqüência de regras determina a realização automática de coerção em expressões aritméticas na linguagem C:

- `char` e `short` são convertidos para `int`, `float` para `double`;
- se um dos operandos é do tipo `double`, o outro é convertido para `double` e o resultado também é um `double`;
- se um dos operandos é `long`, o outro é convertido para `long` e o resultado também é um `long`;
- se um dos operandos é `unsigned`, o outro é convertido para `unsigned` e o resultado também é um `unsigned`;
- senão, todos os operandos são `int` e o resultado é `int`.

Conclusão

A análise semântica verifica a consistência e a correção do código fonte de acordo com as regras da linguagem. A partir da árvore sintática, o analisador semântico assegura que todas as construções do programa fazem sentido semântico. Ele verifica tipos, garante que operações sejam realizadas entre tipos compatíveis e identifica erros como variáveis não declaradas ou redeclarações, que não são capturados durante as análises léxica e sintática. Além disso, a análise semântica realiza o armazenamento de informações de tipo, essenciais para a geração de código intermediário.

A análise semântica é sensível ao contexto, ao contrário das etapas anteriores. Com isso, permite uma verificação mais profunda do comportamento esperado para o programa. A árvore sintática tipada, resultado desta etapa, serve como entrada para a próxima fase do compilador: a geração de código intermediário.

O processo de coerção de tipos é outro aspecto importante da análise semântica, permitindo conversões implícitas que garantem que operações entre diferentes tipos de dados sejam realizadas corretamente. Isso exemplifica como o compilador realiza ajustes para a correta execução do programa. A análise semântica, portanto, é mais uma etapa fundamental para

validar a coerência do código fonte.

Exercícios

1) Usar variáveis não declaradas é um erro sintático ou semântico?

2) (POSCOMP 2018) Sobre tipos de dados, é correto afirmar que:
   a) Tipos booleanos são valores que são mantidos fixos pelo compilador.
   b) O double é um tipo inteiro duplo com menor precisão do que o tipo inteiro.
   c) A faixa de valores dos tipos inteiros tem somente dois elementos: um para verdadeiro e outro para falso.
   d) Uma conversão de tipos implícita consiste em uma modificação do tipo de dados executado, automaticamente, pelo compilador.
   e) Vetores, matrizes e ponteiros são exemplos de tipos de dados primitivos (básicos).

Respostas

1) Semântico.
2) d.

Referências

AHO, Alfred V. et al. Compilers: principles, techniques, & tools. Pearson Education India, 2007.

RICARTE, Ivan Luiz Marques. 2003. Análise semântica. https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node71.html. Acesso em: 01 jun. 2024.

MALAQUIAS, José Romildo. Construção de Compiladores - Capítulo 1: Introdução. 2012. Disponível em: http://www.decom.ufop.br/romildo/2012-2/bcc328/slides/01-introducao.pdf. Acesso em: 01 jun. 2024.

Louden, Kenneth C. Compiladores: princípios e práticas. Porto Alegre: Bookman, 2013.