Compiladores - Introdução

O texto apresentado neste documento é uma adaptação de:

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compiler Principles, Techniques and Tools, Second Edition, Addison-Wesley, 2006.

Linguagens de programação são notações para descrever cálculos para pessoas e para máquinas. O mundo como o conhecemos depende de linguagens de programação, porque todo o software em execução em todos os computadores foi escrito em alguma linguagem de programação. Mas, antes que um programa possa ser processado, ele primeiro deve ser traduzido em uma forma em que pode ser executado por um computador. Os sistemas de software que fazem essa tradução são chamados de compiladores.

1. Processadores de Linguagens

Simplificando, um **compilador** é um programa que pode ler um programa em uma linguagem - a linguagem fonte - e traduzi-lo em um programa equivalente em outra linguagem – a linguagem objeto. Uma representação desse processo pode ser visto na Fig. 1.1. Um papel importante do compilador é relatar quaisquer erros no programa de origem que detecte durante o processo de tradução.

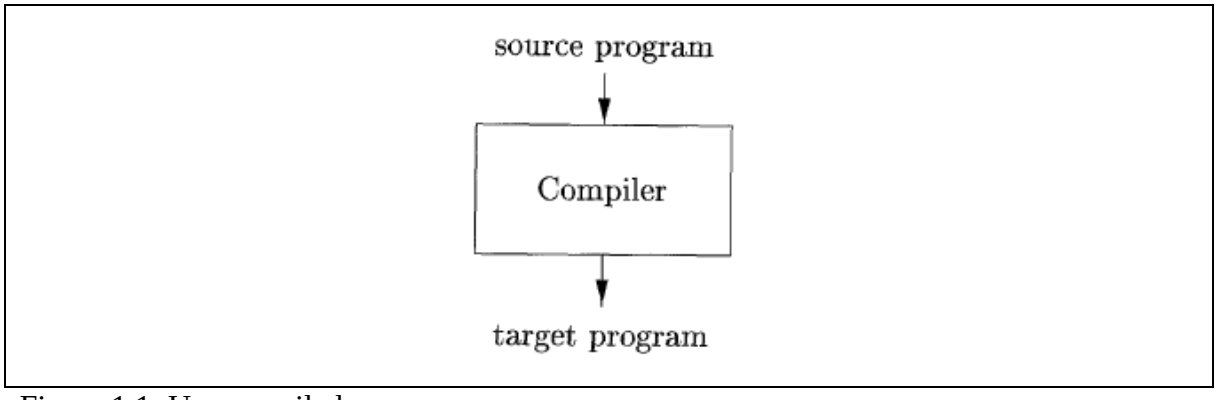

Figura 1.1: Um compilador.

Se o programa objeto é um programa executável codificado em linguagem de máquina, ele pode ser executado por um usuário para processar dados de entrada e produzir saídas, como representado na Fig. 1.2.

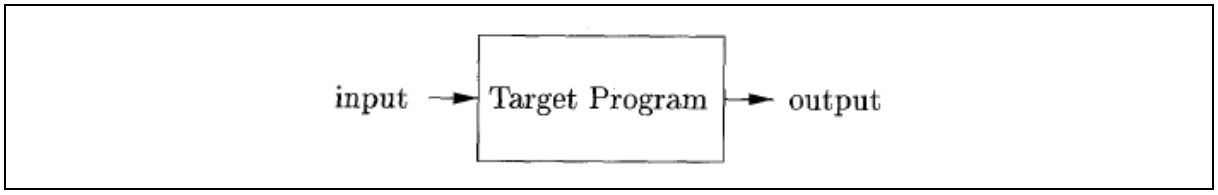

Figura 1.2: Executando o programa objeto.

Um **interpretador** é outro tipo comum de processador de linguagem. Em vez de produzir um programa objeto como uma tradução, um interpretador diretamente simula a executar diretamente as operações especificadas no programa de origem nas entradas fornecidas pelo usuário, conforme mostrado na Fig. 1.3.

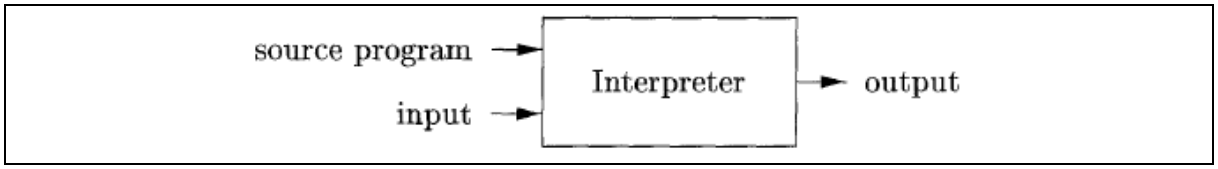

Figura 1.3: Um interpretador.

O programa objeto em linguagem de máquina produzido por um compilador é geralmente muito mais rápido do que um interpretador no mapeamento de entradas para saídas. Um interpretador, entretanto, geralmente pode fornecer diagnósticos de erro melhores do que um compilador, porque executa o programa fonte instrução por instrução.

Os processadores da linguagem Java combinam compilação e interpretação, conforme mostrado na Fig. 1.4. Um programa-fonte Java pode primeiro ser compilado em uma forma intermediária chamada *bytecodes*. Os bytecodes são então interpretados por uma máquina virtual. Um benefício desse arranjo é que os bytecodes compilados em uma máquina podem ser interpretados em outra máquina, facilitando até mesmo sua transferência por uma rede.

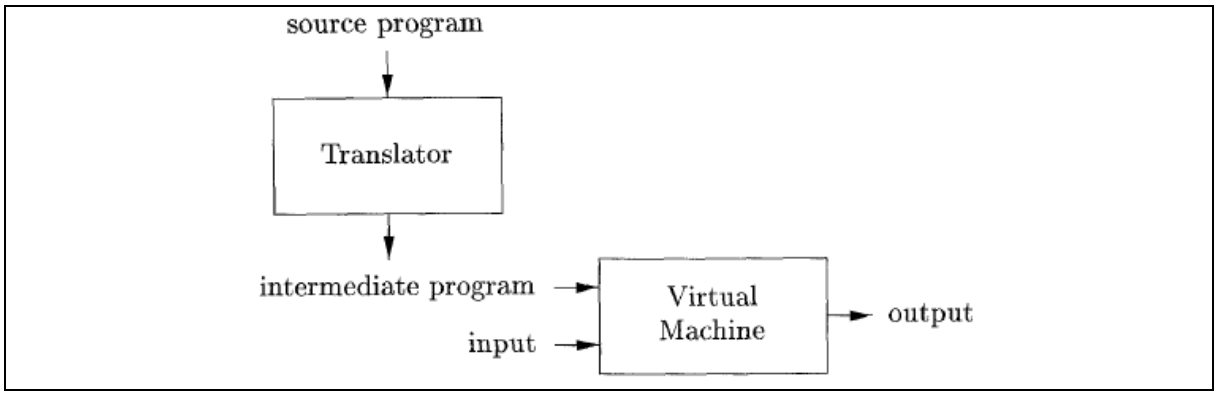

Figura 1.4: Um compilador híbrido.

Para obter um processamento mais rápido de entradas em saídas, alguns compiladores Java, chamados de compiladores *just-in-time*, traduzem os bytecodes em linguagem de máquina imediatamente antes de executarem o programa intermediário para processar a entrada.

2. A Estrututra de um Compilador

Até este ponto, tratamos um compilador como uma única caixa que mapeia um programa fonte em um programa objeto semanticamente equivalente. Se abrirmos um pouco essa caixa, veremos que esse mapeamento tem duas partes: *análise* e *síntese*.

A parte de *análise* divide o programa de origem em partes constituintes e impõe uma estrutura gramatical a elas. Em seguida, usa essa estrutura para criar uma representação intermediária do programa fonte. Se a parte de análise detectar que o programa de origem está sintaticamente malformado ou semanticamente incorreto, ela deve fornecer mensagens informativas, para que o usuário possa executar uma ação corretiva. A parte de análise também coleta e armazena informações sobre o programa fonte em uma estrutura de dados chamada tabela de símbolos, que é passada junto com a representação intermediária para a parte de síntese. A parte da análise é freqüentemente chamada de *front end* do compilador.

A parte de *síntese* constrói o programa objeto desejado a partir da representação intermediária e as informações na tabela de símbolos. A parte da síntese é freqüentemente chamada de *back end* do compilador

Se examinarmos o processo de compilação com mais detalhes, veremos que ele opera como uma sequência de fases, cada uma das quais transforma uma representação do programa fonte para outra representação. Uma decomposição típica de um compilador em fases é mostrada na Fig. 1.5. Na prática, várias fases podem ser agrupadas, e as representações intermediárias entre as fases agrupadas não precisam ser construídas explicitamente. A tabela de símbolos, que armazena informações sobre o programa fonte completo, é usada por todas as fases do compilador.

Alguns compiladores têm uma fase de otimização independente da máquina entre o front end e o back end. O objetivo desta fase de otimização é realizar transformações na representação intermediária, de modo que o back end possa produzir um programa objeto melhor do que teria produzido de outra forma de uma representação intermediária não otimizada. Como a otimização é opcional, uma ou outra das duas fases de otimização mostradas na Fig. 1.5 pode não ser processada.

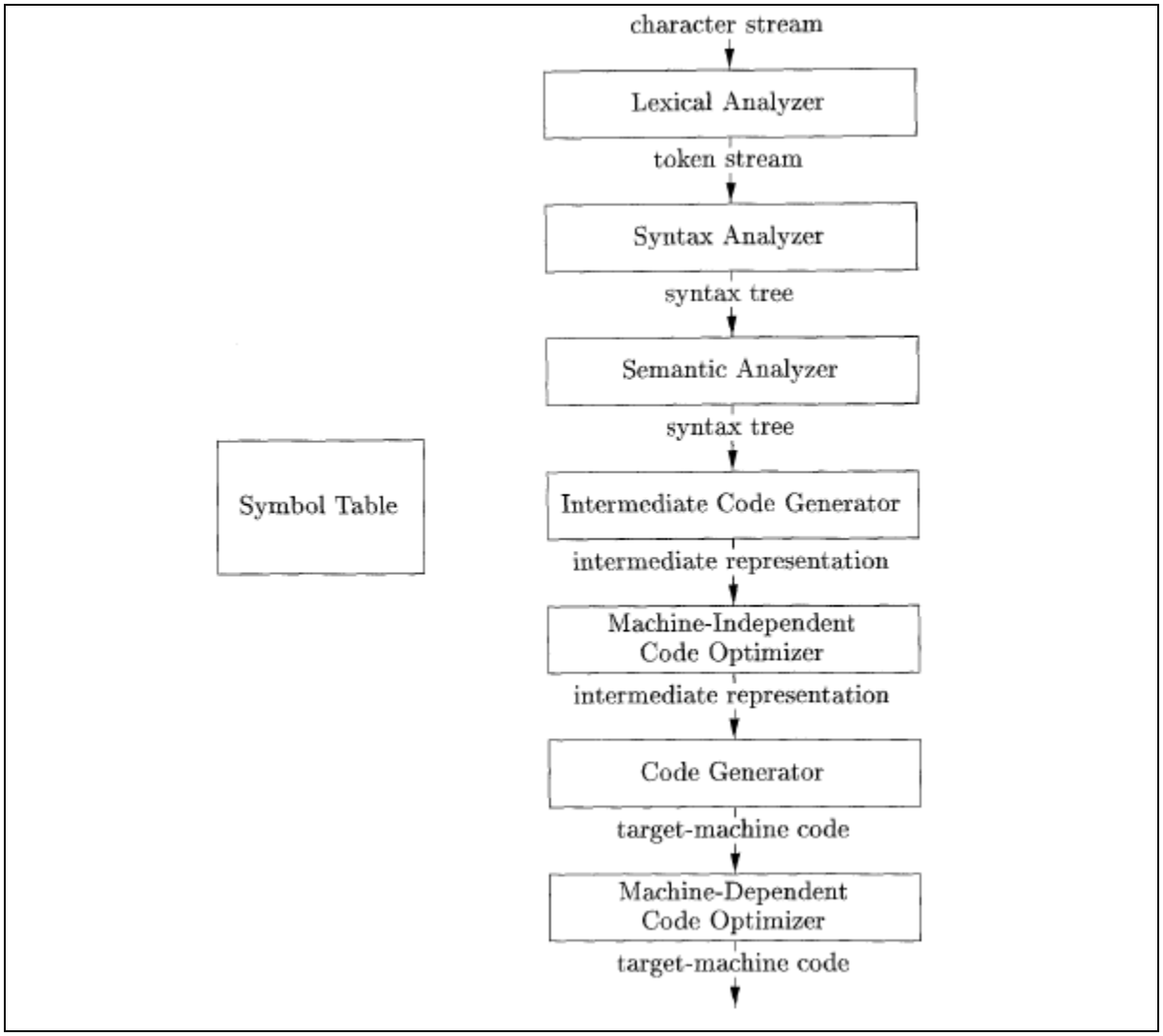

Figura 1.5: Fases de um compilador.

2.1. Análise Léxica

A primeira fase de um compilador é chamada de análise lexical. O analisador léxico lê a sequência de caracteres que constituem o programa fonte e agrupa os caracteres em sequências chamadas *lexemas*. Para cada lexema, o analisador léxico produz como saída um token na forma

<nome do token, valor do atributo>

que é passado para a fase subsequente, a análise sintática. O primeiro componente (nome do token) é um símbolo abstrato usado durante a análise sintática, e o segundo componente (valor de atributo) representa uma informação adicional (opcional).

Por exemplo, suponha que um programa fonte contenha a instrução de atribuição:

```
position = initial + rate * 60
```

Os caracteres nesta atribuição podem ser agrupados nos seguintes lexemas e mapeados nos seguintes tokens, passados para o analisador de sintaxe:

1. `position` é um lexema que mapeado em um token (id, 1), onde id é um símbolo abstrato que representa identificadores e 1 é um índice para uma tabela de símbolos onde a string “position” está armazenada na posição 1;
2. O símbolo de atribuição = é um lexema mapeado no token (=). Uma vez que este token não precisa de valor de atributo, omitimos o segundo
3. `initial` é um lexema que é mapeado no token (id, 2), com “initial” armazenado na posição 2 da tabela de símbolos,
4. + é um lexema mapeado no token (+).
5. `rate` é um lexema que é mapeado no token (id, 3), com “rate” armazenado na posição 3 da tabela de símbolos,
6. * é um lexema mapeado no token (*).
7. 60 é um lexema mapeado no token (number, 60).

A sequência de tokens

$$\langle \mathbf{id}, 1 \rangle \ \langle = \rangle \ \langle \mathbf{id}, 2 \rangle \ \langle + \rangle \ \langle \mathbf{id}, 3 \rangle \ \langle * \rangle \ \langle 60 \rangle$$

é representada na Fig. 1.6 como a saída produzida pela fase de análise léxica. Os espaços em branco que separam os lexemas são descartados pelo analisador léxico.

2.2. Análise Sintática

A segunda fase do compilador é a análise sintática. O analisador usa os primeiros componentes dos tokens produzidos pelo analisador léxico para criar uma representação intermediária em forma de árvore que representa a estrutura gramatical da sequência de tokens. Uma representação típica é uma árvore de sintaxe em que cada nó interior representa uma operação e os filhos do nó representam os argumentos da operação. Uma árvore de sintaxe para o fluxo de tokens do exemplo da Seção 2.1 é mostrada como a saída do analisador sintático na Fig. 1.6.

Essa árvore mostra a ordem em que as operações no comando de atribuição devem ser realizados. A árvore tem um nó interno rotulado * com (id, 3) como seu filho esquerdo e o inteiro 60 como filho direito. O nó (id, 3) representa o identificador `rate`. O nó rotulado * torna explícito que devemos primeiro multiplicar o valor de `rate` por 60. O nó marcado com + indica que devemos somar o resultado dessa multiplicação ao valor de `initial`. A raiz do árvore, rotulada =, indica que devemos armazenar o resultado desta adição no identificador `poition`. Esta ordem de operações é consistente com as convenções usuais da aritmética que nos dizem que a multiplicação tem precedência mais alta do que adição e, portanto, que a multiplicação deve ser realizada antes da adição.

2.3. Análise Semântica

O analisador semântico usa a árvore de sintaxe e as informações da tabela de símbolos para verificar o programa fonte quanto à consistência semântica com a definição da linguagem. Ela também coleta informações de tipo e as salva na árvore de sintaxe ou a tabela de símbolos, para uso subsequente durante a geração do código intermediário.

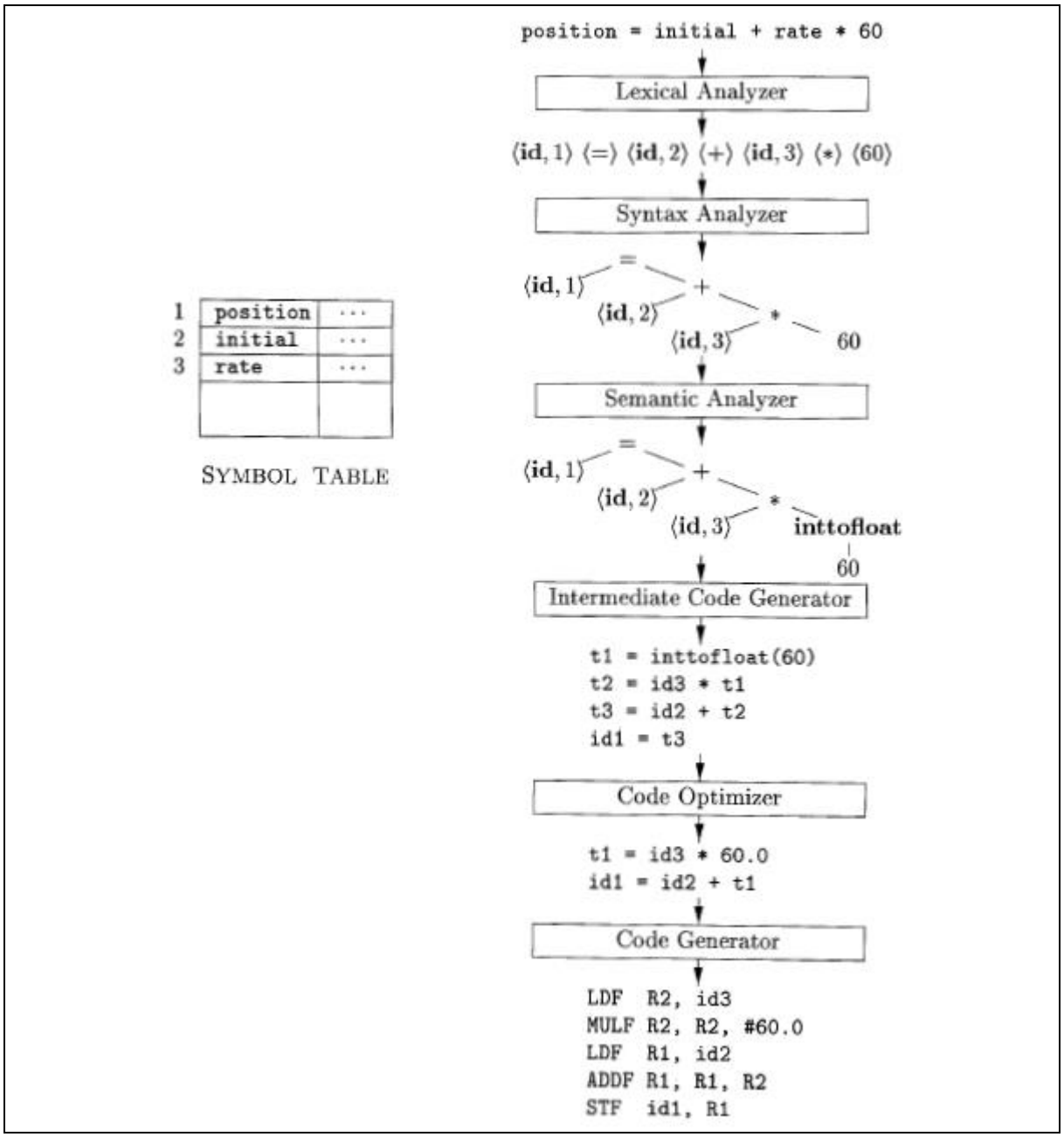

Figura 1.6: Tradução de um comando de atribuição.

Uma parte importante da análise semântica é a verificação de tipos, onde o compilador verifica se cada operador possui operandos correspondentes. Por exemplo, muitas deinição de linguages de programação requerem que um índice de array seja um inteiro; o compilador deve relatar um erro se um número de ponto flutuante for usado para indexar uma matriz.

A especificação da linguagem pode permitir algumas conversões de tipo chamadas coerções. Por exemplo, se um operador aritmético binário for aplicado em um número de ponto flutuante e um inteiro, o compilador pode converter o inteiro em um número de ponto flutuante.

2.4. Geração de Código Intermediária

No processo de tradução de um programa fonte em código objeto, um compilador pode construir uma ou mais representações intermediárias, que podem ter uma variedade de formas. Árvores de sintaxe são uma forma de representação intermediária; elas são comumente usadas durante a análise de sintaxe e semântica.

Após a análise de sintaxe e semântica do programa fonte, muitos compiladores geram uma representação intermediária explícita de baixo nível, que podemos pensar como um programa para uma máquina abstrata. Essa representação intermediária deve ter duas propriedades importantes: deve ser fácil de produzir e deve ser fácil de traduzir para a máquina de destino. Uma possível forma intermediária com essas caracteristicas é chamada código de três endereços, e consiste em uma sequência de instruções semelhantes a código assembly com três operandos por instrução. Cada operando pode atuar como um registrador. A saída do gerador de código intermediário na Fig. 1.6 consiste na sequência de código de três endereços

```
tl = intofloat(60)
t2 = id3 * tl
t3 = id2 + t2
id1 = t3
```

Existem vários pontos dignos de nota sobre as instruções para três endereços. Primeiro, cada instrução de atribuição de três endereços tem no máximo um operador no lado direito. Assim, essas instruções fixam a ordem em que as operações devem ser realizadas. Segundo, o compilador deve gerar um nome temporário para conter o valor calculado por uma instrução de três endereços. Terceiro, algumas "instruções de três endereços", como a primeira e o última na sequência acima, têm menos de três operandos.

2.5. Otimização de Código

A fase de otimização de código independente da máquina tenta melhorar o código intermediário para que o resultado seja um código objeto melhor. Normalmente, melhor significa mais rápido, mas outros objetivos podem ser desejados, como código mais curto ou código objeto que consuma menos energia.

Um algoritmo simples de geração de código intermediário seguido por otimização de código é uma maneira razoável de gerar um bom código objeto. O otimizador pode deduzir que a conversão de 60 de inteiro para ponto flutuante pode ser feita de uma vez em tempo de compilação, de modo que a operação inttofloat pode ser eliminada substituindo o inteiro 60 pelo número de ponto flutuante 60.0. Além disso, t3 é usado apenas uma vez para transmitir seu valor para id1 para que o otimizador possa obter uma sequência mais curta de instruções:

```
tl = id3 * 60.0
id1 = id2 + t1
```

2.6. Geração de Código

O gerador de código recebe como entrada uma representação intermediária do programa fonte e mapeia essa representação para a linguagem objeto. Se a linguagem objeto for código de máquina, registradores ou posições de memória são selecionados para cada uma das variáveis usadas pelo programa. Então, as instruções intermediárias são traduzidas em sequências de instruções de máquina que executam a mesma tarefa. Um aspecto crucial da geração de código é a atribuição criteriosa de registradores para conter variáveis.

Por exemplo, usando os registradores R1 e R2, o código intermediário obtido na Seção 2.5 pode ser traduzido para o código de máquina:

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  Rl, id2
ADDF Rl, Rl, R2
STF  idl , Rl
```

2.6. Gerência da Tabela de Símbolos

Uma função essencial de um compilador é registrar os nomes das variáveis usadas no programa fonte e coletar informações sobre vários atributos de cada nome. Esses atributos podem fornecer informações sobre o armazenamento alocado para um nome, seu tipo, seu escopo (onde no programa seu valor pode ser usado), e no caso de nomes de procedimentos, informações como o número e os tipos de seus argumentos, o método de passagem de cada argumento (por exemplo, por valor ou por referência) e o tipo retornado.

A tabela de símbolos é uma estrutura de dados contendo um registro para cada nome de variável, com campos para os atributos do nome. A estrutura de dados deve ser projetada para permitir que o compilador encontre o registro para cada nome rapidamente e para armazenar ou recuperar dados desse registro rapidamente.