Compilador TIGER
usando JAVA
Manual do Usuário

Relatório Técnico do Laboratório de
Linguagens de Programação - LLP011/99

Gustavo Brandão Escalda
Mariza A. S. Bigonha

Resumo

Este documento é uma descrição suscinta do projeto de desenvolvimento de um compilador para a linguagem TIGER usando a linguagem JAVA como ferramenta de desenvolvimento. O produto desenvolvido tem por objetivo disponibilizar para o aluno do curso de Compiladores um compilador completo, de modo a auxiliar na fixação dos conhecimentos da área.

A linguagem TIGER é uma pequena linguagem com funções aninhadas, registros com apontadores implícitos, arranjos, variáveis inteiras e *strings*, além de algumas construções de controle estruturadas.

O projeto consiste na construção modularizada de um compilador para TIGER, passando pelas etapas de análise léxica, sintática, semântica e tradução para código intermediário, utilizando as ferramentas JLex e CUP para auxílio na construção dos analisadores léxico e sintático, respectivamente.

Sumário

1 Introdução

Este texto relata os resultados obtidos pela construção de um compilador para a linguagem TIGER [2] em um ambiente JAVA.

O objetivo deste projeto é disponibilizar para os alunos de compiladores um compilador completo. Este projeto produziu também como subproduto a formação do aluno em um ambiente de programação de grande interesse hoje não só na área de Linguagens como em outras áreas da Computação.

Devido ao fato de a linguagem JAVA ser relativamente recente, o curso de Compiladores da UFMG ainda não dispunha de um compilador desenvolvido neste ambiente, e, com a construção do mesmo, poder-se-á utilizá-lo como instrumento de ensino aos futuros alunos de Compiladores. A idéia é a cada etapa do compilador substituir o módulo desenvolvido pelo aluno pelo módulo correspondente na ferramenta.

Para que isto aconteça, o paradigma de programação utilizado foi o orientado por objeto [4]. A escolha por este paradigma se explica pela necessidade de se desenvolver a ferramenta o mais modular possível, o que, na orientação por objeto, vem a ser uma premissa básica.

Com este objetivo organizamos a implementação de TIGER em 7 etapas, as quais listamos a seguir:

- Análise Léxica
- Análise Sintática
- Tabela de Símbolos
- Sintaxe Abstrata
- Análise Semântica
- Geração de Código Intermediário

A linguagem TIGER é uma pequena linguagem com funções aninhadas, registros com apontadores implícitos, arranjos, variáveis inteiras e *strings*, além de algumas construções de controle estruturadas.

Este texto está organizado da seguinte forma: a Seção 2 apresenta a linguagem TIGER para a qual desenvolvemos o compilador.

A Seção 3 descreve suscintamente a linguagem JAVA, alvo do desenvolvimento de nosso projeto.

Na Seção 4 é descrita a Máquina Virtual JAVA.

A Seção 5 descreve a implementação do compilador apresentando as decisões de projeto, ferramentas utilizadas, etc..

A Seção 6 apresenta a interface do sistema e descreve como usá-lo.

A Seção 7 apresenta as conclusões.

O Apêndice A.1 apresenta a sintaxe de TIGER.

O Apêndice A.2 apresenta alguns exemplos compilados em TIGER.

2 A Linguagem TIGER

A linguagem Tiger é uma pequena linguagem com funções aninhadas, registros com apontadores implícitos, arranjos, variáveis inteiras e *strings*, além de algumas construções de controle estruturadas. TIGER é composta de duas seções, uma seção de declarações e uma seção de variáveis e expressões. Ela possui também uma biblioteca de funções. A gramática completa de TIGER é apresentada no Apêndice A.1.

2.1 Declarações

Uma seqüência de declarações é uma seqüência de declarações de tipos, variáveis ou funções. Nenhuma pontuação separa ou termina declarações individuais.

decs := *dec decs* | $\mathcal{E}$
dec := *tydec* | *vardec* | *fundec*

2.1.1 Tipos

Existem dois tipos pré-declarados: *int* e *string*. Tipos adicionais devem ser definidos ou redefinidos via declarações de tipos.

Registros: são definidos por meio da listagem de seus campos enclausurados entre chaves, com cada campo descrito por *Nome-Campo : type-id*, sendo *type-id* um identificador definido por uma declaração de tipos.

Arranjos: é definida como **array of** *type-id*. O tamanho do arranjo não é especificado como parte do tipo, devendo esta tarefa ser realizada em tempo de execução.

Tipos mutuamente recursivos: uma coleção de tipos pode ser recursiva ou mutuamente recursiva. Tipos mutuamente recursivos são declarados por meio de uma seqüência consecutiva de declarações de tipos, sem intervir declarações de valor ou de funções.

Reuso de nomes de campo: tipos de registro diferentes podem usar o mesmo nome para seus campos.

2.1.2 Variáveis

Uma declaração de variável é dada por:

var *id* := *exp*
var *id* : *type-id* := *exp*

No primeiro caso, o tipo da variável é determinado via o tipo da expressão. No segundo, o tipo, além de ser dado, deve ser semelhante ao tipo da expressão.

2.1.3 Funções

As funções são declaradas como:

function *id(tyfields)* = *exp*
function *id(tyfields)* : *type-id* = *exp*

O primeiro caso representa uma declaração de procedimento, sendo que procedimentos não retornam valores. O segundo caso representa uma declaração de função, sendo o tipo retornado especificado pelo não terminal *type-id*. O não-terminal *exp* representa o corpo

do procedimento ou função, e *tyfields* especifica os nomes e tipos dos parâmetros. Todos parâmetros são passados por valor.

2.2 Variáveis e Expressões

2.2.1 Variáveis

L-Value: um l-value é um local cujo valor pode ser lido ou atribuído, podendo ser representado por variáveis locais, parâmetros de procedimentos, campos de registros ou elementos de arranjos.

2.2.2 Expressões

As expressões em TIGER podem ser:

L-Value: quando usado como expressão, o l-value é avaliado para o conteúdo do local correspondente.

Seqüenciamento: uma seqüência de duas ou mais expressões, envoltas por parênteses e separadas por ponto-e-vírgula, avaliam todas as expressões em ordem. O resultado, se existir, é o resultado da última expressão.

Nenhum valor: um abre-parênteses, seguido por um fecha-parênteses, bem como a expressão *let* com nada entre *in* e *end*, são exemplos de expressões que não produzem nenhum valor.

Literal inteiro: uma seqüência de dígitos decimais é uma constante inteira que denota o valor inteiro correspondente.

Literal string: uma constante *string* é uma seqüência, entre aspas("), de zero ou mais caracteres.

Chamada de função: definida como a aplicação da função *id()* ou *id(exp{, exp})*. Se *id* representar um procedimento, uma função que não retorna resultado, então o corpo de função não deve produzir nenhum valor, bem como a aplicação da função.

Aritmética: expressões da forma: *exp op exp*, nas quais *op* representa: +, −, *, / requerem argumentos inteiros e produzem resultados inteiros.

Comparação: expressões da forma: *exp op exp*, nas quais *op* representa: =, <>, >, <, >=, <=. Estes operadores testam pela igualdade ou não de seus operandos, produzindo o valor 1 para verdadeiro e 0 para falso. Todos estes operadores podem ser aplicados a operandos inteiros. Além disso, os operadores =, <> também podem ser aplicados a registros ou arranjos do mesmo tipo.

Comparação de strings: os operadores de comparação também se aplicam a strings, testando se o conteúdo de dois deles é ou não igual.

Operadores booleanos: são as expressões da forma: *exp op exp* com *op* sendo & ou |. Qualquer valor inteiro diferente de zero é considerado verdadeiro, sendo o valor inteiro zero considerado falso.

Precedência de operadores: O menos unário (negação) possui a maior precedência. Em seguida vêm os operadores *, /, seguidos por +, −, depois por =, <>, >, <, >=, <=, então por & e finalmente |.

Associatividade dos operadores: os operadores *, /, +, − são todos associativos à esquerda, enquanto os operadores de comparação não se associam.

Criação de registros: definido como *type-id id = exp, id = exp*. Cria uma nova instância de registro do tipo *type-id*.

Criação de arranjos: definido como *type-id[exp1] of exp2*.

Atribuição: definido como *lvalue := exp*. Avalia-se primeiramente o *lvalue*, depois *exp* e então atribui o resultado da expressão ao conteúdo de *lvalue*.

if-then-else: a expressão *if exp1 then exp2 else exp3* avalia a expressão inteira *exp1*. Se o resultado for diferente de zero, *exp2* é avaliado, e caso contrário, avalia-se *exp3*. As expressões *exp2* e *exp3* devem ser do mesmo tipo, que também é o tipo de toda a expressão-if.

If-then: a expressão *if exp1 then exp2* inicialmente avalia a expressão inteira *exp1*. Se o resultado for diferente de zero, então *exp2* é avaliada.

While: a expressão *while exp1 do exp2* avalia a expressão inteira *exp1*. Se o resultado for diferente de zero, então *exp2* é executada, e toda a expressão *while* é reavaliada.

For: a expressão *for id := exp1 to exp2 do exp3* itera *exp3* para cada valor inteiro de *id* entre *exp1* e *exp2*. A variável *id* é uma nova variável implícitamente declarada pelo *for*, cujo escopo cobre somente *exp3*. Se o limite superior for menor que o inferior, então o corpo não é executado.

Break: a expressão *break* termina a avaliação da expressão *while* ou *for* mais próximas. Uma expressão *break* que não esteja dentro de um *while* ou um *for* é considerada ilegal.

Let: a expressão *let decs in expseq end* avalia as declarações *decs*, tipos associados, variáveis e procedimentos cujo escopo se estende sobre *expseq*. O não terminal *expseq* representa uma seqüência de zero ou mais expressões, separadas por ponto-e-vírgula. O resultado, se existir, da última *exp* da seqüência será então o resultado de toda a expressão *let*.

2.3 Biblioteca de Funções

TIGER possui algumas funções em sua biblioteca; como os nomes das mesmas são autoexplicativos, vamos apenas listá-las.

- function print(s : string)
- function flush()
- function getchar() : string
- function ord(s : string) : int
- function chr(i : int) : string
- function size(s : string) : int
- function substring(s : string, first : int, n : int) : string
- function concat(s1 : string, s2 : string) : string
- function exit(i : int)

3 Linguagem JAVA

Por questão de completeza, e considerando que JAVA será usada como ambiente de programação, esta seção expõe as principais características dessa linguagem.

3.1 Histórico

A linguagem JAVA [3] foi criada como parte de um grande projeto da Sun Microsystems, Inc., cuja missão era desenvolver aplicativos complexos e avançados para aplicação em pequenos dispositivos eletrônicos. Esses dispositivos são sistemas portáteis, distribuídos, confiáveis e incorporados em tempo real, dirigidos aos consumidores em geral. As primeiras tentativas de criação de tais dispositivos receberam nomes como: Digital Assitence e Portable Data Assistentes(PDAs).

A primeira tentativa da Sun com esses aplicativos deveria ter sido implementada em C++. Mas em conseqüência de compilações pobres e uma lista cada vez maior de problemas com o C++, principalmente vazamentos de memória e vários problemas de herança, a Sun congelou o C++ e implementou uma nova linguagem, a JAVA. Desde o seu início, em 1992, surgiram muitas outras aplicações para JAVA. Com ela é possível criar programas auxiliares, que podem ser usados para montar documentos HTML particularmente interessantes.

3.2 O que é JAVA?

JAVA é ao mesmo tempo um ambiente e uma linguagem de programação desenvolvida pela Sun Microsystems, Inc. Trata-se de mais um representante da nova geração de linguagens orientadas por objetos e foi projetada para resolver os problemas da área de programação cliente/servidor.

Sendo uma linguagem orientada por objetos, todos os elementos dos programas são tratados como objetos. Os objetos podem ser variáveis, sub-rotinas ou a própria aplicação. A idéia que existe por trás das linguagens orientadas por objetos é que um objeto pode incluir tanto dados quanto código: um objeto "número" , por exemplo, pode incluir o valor do número e o código necessário para representá-lo.

Os aplicativos em JAVA são compilados em um código de *bytes* independente de arquitetura. Esse código de *bytes* pode então ser executado em qualquer plataforma que suporte um interpretador JAVA. JAVA requer somente um fonte e um binário e, mesmo assim, é capaz de executar em diversas plataformas, o que faz dele um sonho de todos os que realizam manutenção em programas.

As primeiras notícias que se ouviu de JAVA era que ela possibilitava colocar animações em uma página Web. JAVA fazia isso por meio de um descritor HTML especial (`<applet>`), aceito pelo browser HotJava, que permitia incluir um programa em uma página, da mesma maneira em que se insere uma imagem. O programa era transferido e executava na máquina do usuário. A sua aplicação não se limitava a animações. O que se podia fazer com JAVA dependia apenas da imaginação do programador. Esses programas, que executavam dentro de páginas Web, foram chamados de Applets.

Muitos ainda achavam que JAVA era só mais um brinquedo para tornar a Web mais atraente. O interesse cresceu quando as pessoas começaram a descobrir que JAVA era mais que isso. Era uma poderosa linguagem que podia ser usada para desenvolver aplicações, como as que se faz hoje usando C++. E mais: JAVA é muito mais simples, segura e robusta e possui uma qualidade que a diferencia de todas as outras: a independência de plataforma. Com JAVA, é possível desenvolver um único programa e executá-lo em qualquer lugar.

As principais características de JAVA são:

JAVA é orientada por objetos:

A linguagem JAVA é um membro do paradigma orientado por objetos (OO) das linguagens de programação. As linguagens que aceitam este paradigma, como JAVA e C++, seguem a mesma filosofia básica, mas diferem em sintaxe e estilo. As linguagens orientadas por objetos oferecem muitas vantagens sobre as linguagens procedurais tradicionais. Como os objetos encapsulam dados e funções relacionados em unidades coesas, é fácil localizar dependências de dados, isolar efeitos de alterações e realizar outras atividades de manutenção, talvez o mais importante, as linguagens OO facilitam a reutilização. Não existem variáveis globais ou funções independentes. Toda variável ou método pertence a uma classe ou objeto e só pode ser invocada através dessa classe ou objeto.

JAVA é distribuída:

Distribuição de informações para compartilhamento e trabalho conjunto, com a distribuição de carga de trabalho do processamento, é uma característica essencial dos aplicativos cliente/servidor. Felizmente para os programadores JAVA, há uma biblioteca de procedimentos TCP/IP incluída nos códigos-fonte e de distribuição binária do JAVA. Isso facilita aos programadores o acesso remoto às informações, usando protocolos como HTTP e FTP.

JAVA é compilável e independente de plataforma:

Um programa escrito em JAVA precisa ser compilado antes de ser executado. O compilador traduz o código-fonte e gera arquivos objeto chamados arquivos de classe. Cada programa JAVA consiste da implementação de uma única classe que depois de compilado, pode ser executado em qualquer plataforma onde exista um sistema de tempo de execução JAVA (*runtime*).A compilação de um programa em C++ realiza a tradução do código-fonte da linguagem em instruções que são interpretadas pelo microprocessador da máquina onde foi compilado. O programa então só executa em outra máquina que tenha o mesmo tipo de processador. Já um programa em JAVA, quando compilado, gera instruções, chamadas de *bytecodes*, para um microprocessador virtual. Existem implementações desse microprocessador virtual para várias plataformas e o programa então executará em qualquer uma delas. Por ter suas instruções interpretadas por um software, processador virtual, os programas em JAVA são mais lentos que os escritos em C ou C++. A igualdade é atingida usando microprocessadores virtuais com compiladores Just-In-Time (JIT), que já são bastante comuns. Esse tipo de sistema converte as instruções em *bytecodes* para instruções do microprocessador no momento da execução, fazendo com que programas escritos em JAVA não tenham grandes perdas de desempenho em relação a programas escritos em C ou C++.

JAVA é multitarefa:

Os objetos binários de códigos de bytes do JAVA são formados por seqüências de execuções múltiplas e simultâneas. Essas seqüências são conhecidas como contextos de execução ou processos leves. As linguagens C e C++ são membros de um paradigma de execução em seqüência única, por não oferecerem suporte a seqüências no nível de linguagem. JAVA, no entanto, oferece suporte no nível de linguagem para multitarefa, resultando em uma abordagem de programação mais poderosa e de múltiplas facetas.

JAVA é dinâmica:

A linguagem JAVA foi projetada para se adaptar a um ambiente dinâmico, em constante evolução. Possui uma representação de tempo de execução que permite que o programa saiba a classe a que pertence um objeto, na hora em que o recebe, durante a execução. Isso permite a inclusão dinâmica de classes que podem estar localizadas em qualquer endereço da Internet. JAVA também suporta a integração com métodos nativos de outras linguagens. Desta forma, podem surgir em breve

aplicativos híbridos em JAVA e C++, aproveitando o grande volume de código existente hoje em C++.

JAVA é robusta:
Quanto mais robusto um aplicativo, mais confiável ele será. Isso é desejável tanto para os desenvolvedores de software quanto aos consumidores. A maioria das linguagens OO, como o C++ e JAVA, possuem tipos fortes. Isso significa que a maior parte da verificação de tipos de dados é realizada em tempo de compilação, e não em tempo de execução. Isso evita muitos erros e condições aleatórias nos aplicativos. JAVA, ao contrário do C++, exige declarações explícitas de métodos, o que aumenta a confiabilidade dos aplicativos.

JAVA é segura:
Como JAVA foi criada para ambientes de rede, os recursos de segurança receberam muita atenção. Por exemplo, para executar um binário transferido pela rede, é possível que o mesmo esteja infectado por vírus. Os aplicativos JAVA apresentam garantia de resistência contra vírus e de que não são infectados por vírus, pois não são capazes de acessar *heaps*, *stacks* ou memória do sistema. Em JAVA, a autenticação do usuário é implementada com um método de chave pública de criptografia. Isso impede de maneira eficaz que *hackers* e *crackers* examinem informações protegidas como nomes e senhas de contas.

JAVA é simples:
Um dos principais objetivos do projeto de JAVA foi criar uma linguagem o mais próxima possível do C++, para garantir sua rápida aceitação no mundo do desenvolvimento OO. Outro objetivo do seu projeto foi eliminar os recursos obscuros e danosos do C++, que fugiam à compreensão e aumentavam a confusão que poderia ocorrer durante as fases de desenvolvimento, implementação e manutenção do software. JAVA é simples porque é pequena. O interpretador básico de JAVA ocupa aproximadamente 40kb de RAM, excluindo-se o suporte a multitarefa e as bibliotecas padrão, que ocupam outros 175kb. Mesmo a memória combinada de todos esses elementos é insignificante, se comparada a outras linguagens e ambientes de programação.

JAVA oferece alto desempenho:
Há muitas situações em que a interpretação de objetos de códigos de *bytes* proporciona desempenho aceitável. Mas outras circunstâncias exigem desempenhos mais elevados. JAVA concilia tudo isso oferecendo a tradução dos códigos de *bytes* para o código de máquina nativo em tempo de execução. O alto desempenho permite a implementação de seus aplicativos *web* em JAVA, na forma de programas pequenos e velozes, que podem ampliar significativamente os recursos tanto do cliente quanto do servidor.

3.3 Como é Formado um Programa JAVA

A maioria dos programas em JAVA contém um ou mais objetos ou unidades de compilação. Cada unidade contém declarações de pacote e importação, além de declarações de classes e interface. Embora se possa ter várias classes e interfaces por unidade, cada unidade pode ter no máximo uma interface e uma classe pública, porque é dessa maneira que a unidade será acessada por outros objetos do sistema. O restante deve ser privado, o que pode ser declarado explicitamente como privado na declaração de classe, ou aceito como *default*.

O compilador JAVA, `javac`, compila código-fonte JAVA em código de *bytes*. Esses códigos de *bytes* contêm instruções independentes de máquina. Em seguida, o interpretador de JAVA, também chamado de `java`, e parte do ambiente de execução de JAVA,

interpreta esses códigos de *bytes*. Ao criar uma unidade de compilação de JAVA, cada uma deverá ter um sufixo .java.

Exemplo de um programa JAVA pode ser visto a seguir:

```
//comentario de uma linha
public class ExemploJava
{
      public static void main(String args[])
      {
            System.out.println("Exemplo Java");
      }
}
```

onde,

//comentario de uma linha : comentários em JAVA seguem a mesma sintaxe de C++. "//" inicia uma linha de comentário, sendo todo o restante da linha ignorado. Existe também um outro tipo de comentário formado por /*Insira o comentário aqui*/, podendo ser intercalado com linhas de código. Comentários são tratados como espaços em branco.

public class ExemploJava { : class é a palavra reservada que marca o início de uma classe; *public* é um especificador.

ExemploJava : representa o nome dado a esta classe.

public static void main(String args[]) {

System.out.println("Exemplo Java"); } : *public* é um qualificador do método que indica que o mesmo é acessível externamente a esta classe (para outras classes que eventualmente sejam criadas). `Static` é um outro qualificador ou "specifier" , que indica que o método deve ser compartilhado por todos os objetos que são criados a partir dessa classe. Os métodos `static` podem ser invocados, mesmo quando não foi criado nenhum objeto para a classe, com base na seguinte sintaxe:

`<NomeClasse>.<NomemetodoStatic>(argumentos);`.

void : semelhante ao C++ ou C, é o valor de retorno da função, quando a função não retorna nenhum valor é retornado `void`, uma espécie de valor vazio que deve ser especificado.

main : este é um nome particular de método que indica para o compilador o início do programa. É dentro deste método e via iterações entre os atributos, variáveis e argumentos nele visíveis que o programa se desenvolve.

(String args[]) : representa o argumento de `main` e por conseqüência de todo o programa. É um vetor de *strings* que é formado quando são passados ou não argumentos da invocação do nome do programa na linha de comando do sistema operacional.

System.out.println("Exemplo Java"); : chamada do método `println` para o atributo `out` da classe ou objeto `system`, o argumento é uma constante do tipo String.println, a qual imprime a string e posiciona o cursor uma linha abaixo.

4 Máquina Virtual JAVA

4.1 Descrição

A Máquina Virtual JAVA(MVJ) [9] é uma máquina abstrata definida por uma especificação formal. Sendo assim, ela pode ser implementada de diversas maneiras, tal como um interpretador, um compilador ou mesmo em *hardware*, conferindo grande flexibilidade à arquitetura na qual ela será aplicada.

A relação existente entre a MVJ e a linguagem JAVA se resume a um formato particular de arquivo denominado arquivo *class*. Este arquivo contém as instruções da MVJ, ou *bytecodes*, e uma tabela de símbolos, além de informações auxiliares. Para fins de segurança, seu formato é bastante rígido e restrições estruturais são impostas a seu código. Apesar disso, qualquer linguagem com funcionalidade que possa ser expressa em termos de um arquivo *class* válido pode ser hospedada pela MVJ. Este arquivo possui todas as definições necessárias para a execução de código na máquina virtual, e nele encontramos uma lista ordenada, estruturada, que define variáveis, constantes, objetos com seus atributos e métodos representados na forma de tabelas e seqüência de *opcodes* e operandos.

O ciclo de vida de uma classe pode ser dividido em duas partes: uma responsável pelo carregamento das classes e outra pela execução.

Na primeira parte, composta por três fases, é efetuado o carregamento e a preparação para execução:

Fase 1: Carregamento
: Carrega o fluxo binário contido no arquivo *class* para a estrutura interna de dados.

Fase 2: Ligamento [1]
: Esta fase subdivide-se em três partes:

  Verificação: assegura que o arquivo obedeça à semântica da linguagem JAVA e não viole a integridade da máquina virtual.

  Preparação: aloca memória para as variáveis de classe, inicializando-as com valores `default`.

  Resolução: efetua trocas de referência, mudando-as de simbólica para direta. É opcional, já que pode ser executada no momento de referência ao símbolo.

Fase 3: Inicialização
: Liga as variáveis de classe previamente alocadas com os valores determinados pelo programador.

Na segunda parte, responsável pela execução, é feita a interpretação e execução dos *bytecodes*. Para armazenar todos os dados da execução de uma aplicação é definida uma área de execução de dados da seguinte forma:

- área de métodos
- heap
- pilhas JAVA
- registradores PC
- pilhas de métodos nativos

A área de métodos e o *heap* são compartilhados por todos os *threads*, sendo que na área de métodos são armazenadas as informações sobre as classes do programa e no *heap* cada objeto instanciado.

[1] do inglês Linkage

Cada *thread* possui um conjunto de registradores PC, *stack pointer*, etc e uma pilha. Esta pilha armazena *frames*, os quais são compostos por variáveis locais, parâmetros de chamada, valores de retorno e resultados intermediários.

A MVJ não possui registradores de uso geral, sendo a pilha a responsável por armazenar valores intermediários. Esta técnica foi adotada em virtude da proposta de independência de plataforma, de modo que quanto menos registradores fossem exigidos, maior seria o número de arquiteturas capazes de permitir a execução da máquina virtual.

Para realizar a execução, um conjunto de instruções é tomado como base, sendo que a maioria das instruções envolve operação de pilha, uma vez que a MVJ é baseada em pilha.

4.2 Tipos de Dados

Os tipos de dados sobre os quais a MVJ opera são os seguintes:

- Primitive Types
  - Numeric Types
    - Floating-Point Types
      - float
      - double
    - Integral Types
      - byte
      - short
      - int
      - long
      - char
  - returnAddress
- Reference Types
  - reference
    - class types
    - interface types
    - array types

5 Metodologia de Desenvolvimento do Compilador para TIGER

5.1 Ferramentas de Apoio

Para desenvolver o compilador, foram utilizadas as ferramentas JLex [6] e CUP [7]. A primeira foi usada no projeto da análise léxica, e a segunda foi utilizada durante a análise sintática.

5.1.1 JLex

A ferramenta JLex é um gerador de analisador léxico para JAVA que recebe como entrada um arquivo com a especificação léxica da linguagem na forma de expressões regulares e gera o código fonte JAVA correspondente ao analisador léxico.

Esta ferramenta encontra-se disponível no endereço [6] em formato de código fonte JAVA e, portanto, deve ser compilada antes de sua execução. Deve-se, ainda, ajustar-se o CLASSPATH para que a mesma possa ser referenciada a partir de qualquer caminho do ambiente em questão. A seguir apresentamos como isto pode ser feito em ambiente Unix com bash `cshell`. Supondo que os arquivos correspondentes ao JLex estejam no caminho `/java/JLex`, inclua no arquivo `.cshrc` a diretiva:

```
setenv CLASSPATH ./:/java
```

Para executar o JLex entre com a linha de comando:

```
java JLex.Main <NOME DO ARQUIVO>
```

onde `<NOME DO ARQUIVO>` é o arquivo de especificação léxica para a linguagem em questão. Como resultado da execução desta linha de comando, será gerado o arquivo `<NOME DO ARQUIVO>.java` se não houver erros no arquivo de entrada.

5.1.2 CUP

A ferramenta CUP é um gerador de analisador sintático LALR para JAVA. Como resultado, ela gera o código fonte JAVA correspondente ao analisador sintático.

Esta ferramenta pode ser obtida no endereço [7] e deve ser compilada pelo interpretador JAVA antes de ser executada. Tal como na ferramenta JLex, ela deve ser incluída no CLASSPATH para ser referenciada a partir de qualquer caminho do ambiente. Supondo-se que a ferramenta foi instalada no caminho `/java/java_cup`, podemos referenciar a variável CLASSPATH da seguinte forma:

```
setenv CLASSPATH ./:/java
```
[2]

Para executar o CUP entre com a linha de comando:

```
java java_cup.Main < <NOME DO ARQUIVO>
```

onde `<NOME DO ARQUIVO>` é o arquivo contendo a especificação da gramática para a linguagem em questão. Não havendo erros durante a execução do CUP, são gerados dois arquivos fonte em JAVA, `parser.java` e `sym.java`, além do arquivo binário `CUP$Grm$actions.class`.

[2] o caminho `/java/java_cup` descrito é aquele que contém o arquivo `Main.class`.

5.2 Estruturas de Dados

5.2.1 Sintaxe Abstrata

É possível em CUP, YACC, ou em outras ferramentas de geração de compiladores juntar as fases de análise sintática e semântica. Mas, tendo em vista que um dos requisitos básicos propostos para este projeto é o desenvolvimento modular do mesmo, torna-se necessário desvincular os procedimentos referentes à análise sintática daqueles referentes à análise semântica. Para tanto, construímos uma árvore de sintaxe abstrata, a qual torna mais fácil a leitura e manutenção do compilador. A linguagem na qual os programas são escritos é conhecida como sintaxe concreta. Árvores de sintaxe concreta são úteis no entendimento da gramática, porém contêm muita informação desnecessária ao compilador. O uso direto de tais árvores é incoveniente, uma vez que muitos dos *tokens*, como os de pontuação, são redundantes e não transmitem qualquer informação relevante.

Sintaxe abstrata, por outro lado, é mais simples, uma vez que omite os detalhes irrelevantes da sintaxe concreta, tornando-se conveniente no uso por um compilador. A árvore de sintaxe abstrata exprime a estrutura das expressões do programa fonte com todos os procedimentos sintáticos resolvidos mas sem qualquer interpretação semântica, provendo uma interface limpa entre o analisador sintático e as demais etapas do compilador. O uso de sintaxe abstrata na contrução de compiladores é um fato recente. Tal técnica não era comumente utilizada até há poucos anos atrás devido à quantidade insuficiente de memória requerida para representar uma árvore sintática completa. Computadores modernos, entretanto, não mais apresentam esse problema. Por exemplo, para o programa TIGER a seguir:

```
(a := 5; a + 1)
```

seria gerada a seguinte sintaxe abstrata:

```
SeqExp(ExpList(AssignExp(SimpleVar(a), IntExp(5)),
ExpList(OpExp(PLUS, varExp(SimpleVar(a)), IntExp(1)))))
```

5.2.2 Tabela de Símbolos

A Tabela de Simbolos é uma estrutura utilizada pelo compilador para controlar as informações de tipo e escopo dos nomes utilizados em um programa. Na implementação aqui proposta, tabelas hash são utilizadas devido à necessidade de se fornecer acesso rápido e eficiente aos símbolos. Além disso, informações de escopo são utilizadas para informar o escopo de determinado símbolo em um instante específico de execução do programa. O uso de símbolos no lugar de *strings* justifica-se na eliminação de comparações desnecessárias, uma vez que todas as diferentes ocorrências de determinado *string* convertem para o mesmo símbolo. Além disso, operações de manipulação de símbolos tais como comparações e extração de chaves *hash* são extremamente rápidas.

Um símbolo pode assumir um dos tipos listados abaixo:

- int
- string
- record
- array
- nil
- void
- name

5.2.3 Registro de Ativação

É comum em muitas linguagens que diversas invocações de uma função sejam realizadas ao mesmo tempo. Para dar suporte a esses diversos ambientes de execução é necessária uma estrutura de dados capaz de armazenar informações de variáveis locais, parâmetros, endereço de retorno, além de eventuais temporários requisitados.

A estrutura de dados que utilizaremos é conhecida como uma pilha de registros de ativação, que nada mais é que um grande arranjo que pode expandir-se ou contrair-se indefinidamente, associado a um registrador especial, o *stack pointer*. Este registrador aponta para uma posição específica da pilha, de modo que todas as posições além dele são consideradas lixo, e todas anteriores são tidas como alocadas.

Uma pilha inicia-se em endereços altos de memória, crescendo em direção a endereços menores. Sua estrutura é apresentada na Figura 1.

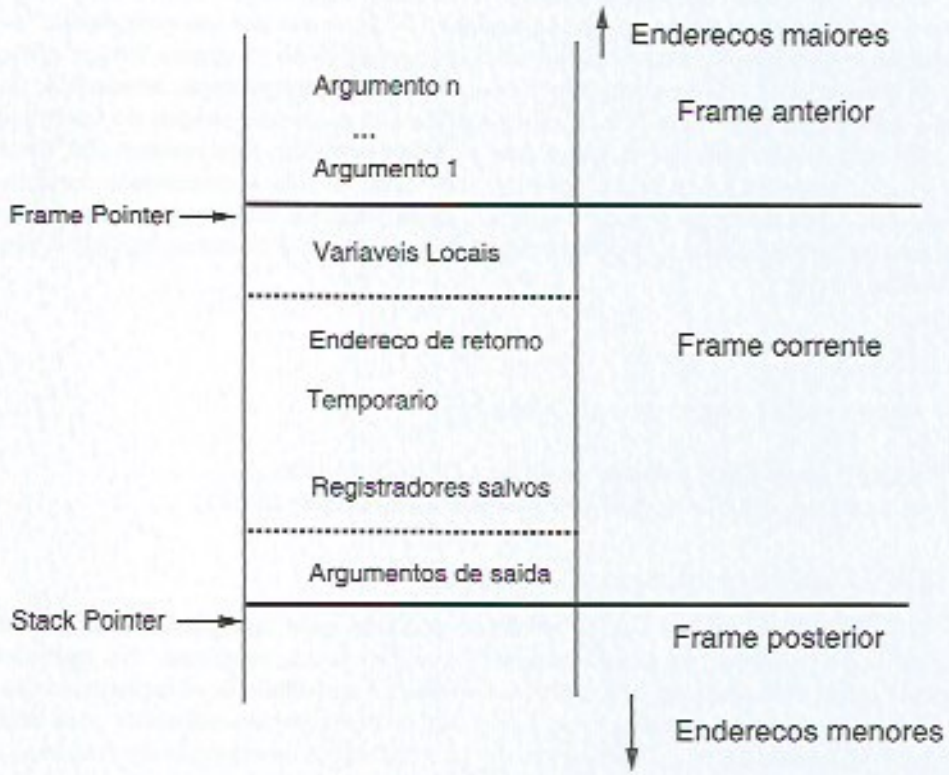

Figura 1: Estrutura do Registro de Ativação

Analisando a figura, cada registro de ativação possui áreas reservadas para itens específicos. Há uma área para variáveis locais, que são alocadas ou neste espaço de endereçamento ou em registradores temporários; uma área para armazenar o endereço de retorno para onde o fluxo de controle deve retornar quando da conclusão da rotina que foi chamada; outra para armazenar valores quando estes não puderem ser alocados em registradores, ou quando registradores em uso forem requisitados, devendo seus conteúdos serem guardados em memória; finalmente, há a área para passagem de argumentos, necessária quando a rotina correspondente ao registro de ativação corrente invocar uma função e necessitar passar parâmetros a ela.

Variáveis em TIGER que são acessadas a partir de um procedimento aninhado devem ser armazenadas no registro de ativação, de modo a ser possível localizá-las independen-

temente de onde estão sendo acessadas. Para isso, utilizamos o conceito de *link estático*, de modo a permitir saber com que entidade cada registro de ativação se comunica. Um *link estático* nada mais é que uma referência ao procedimento que invocou o procedimento corrente. Com isso, para encontrar a área de memória referente a determinada variável acessada em um nível diferente do de sua declaração, basta percorrer-se o *link estático* de cada registro de ativação até que a ocorrência desejada seja encontrada.

Sempre objetivando a construção modularizada do compilador, introduzimos uma nova camada de abstração representativa do registro de ativação, de modo a separar a semântica da linguagem de origem da representação efetiva do registro de ativação dependente da linguagem alvo.

5.3 Descrição das Fases de Compilação

Nesta seção são descritas para cada uma das fases do compilador como elas foram implementadas.

5.3.1 Análise Léxica

Esta fase é responsável pelo agrupamento de seqüências de caracteres em tokens. Para implementá-la, foi utilizada a ferramenta JLex. Esta ferramenta recebe como entrada o arquivo de especificação Tiger.lex. Este arquivo contém a especificação do analisador léxico a ser construído e é organizado em três seções, separadas pela diretiva %%, como mostrado a seguir:

Código do usuário
%%
Diretivas JLex
%%
Expressões regulares

onde:

Código do usuário: este código é copiado no topo do arquivo fonte do analisador léxico a ser gerado, sendo útil, por exemplo, para declaração de pacote ou importação de classes externas.

Diretivas JLex: esta seção engloba diversas diretivas para definição de macros, declaração de estados, além de customizações do analisador.

Expressões regulares: consiste de uma série de regras responsáveis pela quebra da entrada em tokens.

Exemplo de um arquivo de especificação JLex:

```
package Parse;
%%
%type java_cup.runtime.Symbol
%{
 private java_cup.runtime.Symbol tok(int kind, Object value) {
    return new java_cup.runtime.Symbol(kind, yychar, yychar+yylength(),
value);
 }
%}
 %eofval{
        {
         return tok(sym.EOF, null);
        }
 %eofval}
%%
" "     {}
\n      {newline();}
","     {return tok(sym.VIRG, null);}
```

Para geração do analisador léxico, executa-se o comando `java JLex.Main Tiger.lex` e obtém-se o arquivo `Tiger.lex.java`, o qual deve então ser renomeado para `Ylex.java` para manter a compatibilidade com o restante do programa.

5.3.2 Análise Sintática

Esta fase é responsável pelo agrupamento dos tokens produzidos pela análise léxica em frases gramaticais. A tarefa de construir tabelas sintáticas LR(1) [1] ou LALR(1) [1] é uma tarefa muito simples de ser automatizada, de modo que ela raramente é implementada de outra forma que não por meio do uso de ferramentas geradoras de analisadores sintáticos. CUP 5.1.2 é uma tal ferramenta, similar à difundida Yacc [5].

Para geração do analisador utilizamos a ferramenta CUP em conjunto com o arquivo Tiger.cup, invocando o comando

```
java java_cup.Main -parser Grm -expect 6 < Tiger.cup > Grm.out 2> Grm.err
```

e obtendo os arquivos `Grm.java`, sym.java e `CUP$Grm$actions.class`.
CUP recebe como entrada um arquivo contendo a especificação da gramática para a qual é necessária a construção do analisador, juntamente com rotinas responsáveis pela invocação do analisador léxico. A especificação deste arquivo possui um preâmbulo, seguido pelas regras gramaticais. No preâmbulo são declarados os terminais e não-terminais da gramática, além de ser especificado de que modo o analisador léxico se comunica com o sintático. As regras gramaticais são produções da forma

exp ::= *exp PLUS exp*, onde *exp* é um não-terminal e *PLUS* é um terminal.

Exemplo de um arquivo de especificação CUP:

```
package Parse;
scan with {: return lexer.nextToken(); :};
terminal String ID, STRING;
terminal Integer INT;
non terminal program;
start with program;
program ::= ID
```

Esta fase do compilador produz como saída um conjunto de arquivos contendo o código referente ao analisador sintático, os quais, após compilação e execução, informam se a linguagem de entrada para o compilador está ou não com a sintaxe correta.

5.3.3 Análise Semântica e Tradução para Código Intermediário

Esta fase é responsável pela verificação de erros semânticos no programa fonte, além de realizar a captura de informações de tipo para a etapa subseqüente de geração de código.

A verificação de tipos consiste em verificar se todo termo possui o tipo para ele esperado. Por exemplo, o programa

```
let
  type tipoarranjo = array of int
  var Arranjo:tipoarranjo := tipoarranjo [10] of 0
in Arranjo[2] = "nome"
end
```

é aceito pelo analisador sintático apesar de estar semanticamente incorreto, uma vez que um *string* está sendo armazenado em uma posição do arranjo preparada para receber somente valores inteiros.

Uma representação intermediária de código é uma forma de se escrever código de baixo nível não atrelado a nenhuma linguagem ou máquina específica, garantindo a portabilidade do compilador.

Para a implementação do código intermediário, é criada uma árvore, a qual delineia a estrutura hierárquica natural de um programa fonte. Especificamente, para cada regra de produção existente, cria-se um nó correspondente na árvore, o qual contém as informações de código intermediário correspondentes.

As informações possíveis de serem utilizadas na geração de código intermediário são as seguintes:

CONST(*i*): representa a constante inteira *i*.

NAME(*n*): é a constante simbólica *n*, correspondente a um *label* de uma linguagem *assembly*.

TEMP(*t*): é o temporário *t*. Similar a um registrador de uma máquina real.

BINOP(*o*, *e1*, *e2*): é a aplicação do operador binário *o* aos operandos *e1* e *e2*, sendo *e1* avaliado antes de *e2*. O operador *o* pode ser qualquer operador aritmético ou lógico.

MEM(*e*): representa o conteúdo de *n bytes* de memória a partir do endereço *e*, onde *n* é o tamanho de uma palavra na máquina para a qual o código será gerado.

CALL(*f*, *l*): representa a chamada de um procedimento *f* com sua lista de argumentos em l.

ESEQ(*s*, *e*): representa um *statement* seguido por uma expressão.

MOVE(*destino*, *origem*): utilizado para implementar atribuições e inicializações, podendo *destino* representar um TEMP ou um MEM.

EXP(*e*): avalia *e* e descarta o resultado.

JUMP(*e*, *labels*): transfere o controle incondicionalmente para o endereço calculado pela expressão *e*. A lista de *labels* indica todas as possíveis posições para as quais a expressão e pode avaliar.

CJUMP(*o*, *e1*, *e2*, *t*, *f*): representa um desvio condicional, no qual as expressões *e1* e *e2* são avaliadas e comparadas em seguida via operador *o*, desviando para *t* ou *f* dependendo do resultado da comparação.

SEQ(*s1*, *s2*): similar a ESEQ, mas com ambos os componentes *s1* e *s2* statements.

LABEL(*n*): representa um *label* de linguagem *assembly*.

6 Interface com o usuário

A execução do compilador pode ser feita a partir de qualquer plataforma que contenha um interpretador *java* instalado.

O compilador está disponibilizado em diversos pacotes, cada um contendo classes relacionadas e responsáveis por determinada funcionalidade do compilador. Listamos em seguida estes pacotes, descrevendo a funcionalidade principal de cada um:

/Absyn: camada de abstração que permite separar as ações sintáticas das ações semânticas.

/Assem: provê uma estrutura de tipos de dados para instruções de linguagem *assembly* sem designação de registradores.

/ErrorMsg: responsável pela produção de mensagens de erro.

/Frame: provê a estrutura de dados referente ao registro de ativação, sem ater-se a detalhes de implementação.

/JavaVM: implementa detalhes referentes à Máquina Virtual JAVA.

/Main: responsável pelo direcionamento do fluxo de dados no compilador.

/Parse: contém os analisadores léxico e sintático.

/Semant: contém o analisador semântico.

/Symbol: provê a tabela de símbolos em conjunto com suas operações de acesso.

/Temp: provê os temporários e os *labels*.

/Translate: provê a tradução para código intermediário.

/Tree: provê a linguagem da árvore de representação intermediária.

/Types: descreve os tipos de dados da linguagem TIGER.

/Util: provê a classe de lista de boolcanos.

Para executar o compilador deve-se digitar a seguinte linha de comando:

```
java Main.Main <NOME DO ARQUIVO> [-opções]
```

onde:

`<NOME DO ARQUIVO>`: é o arquivo contendo o programa TIGER a ser compilador.

e opções pode ser:

absyn: imprime na saída padrão a árvore de sintaxe abstrata.

listinput: imprime na saída padrão a listagem do arquivo de entrada.

listparse: imprime na saída padrão os estados percorridos pelo analisador sintático.

intermcode: imprime na saída padrão o código intermediário gerado.

7 Conclusões

Durante o desenvolvimento desse projeto foram construídas as principais etapas de um compilador para a linguagem TIGER, sendo elas a análise léxica responsável pelo agrupamento de seqüências de caracteres em tokens, a análise sintática responsável pelo agrupamento dos tokens em frases gramaticais, a análise semântica responsável pela verificação de erros semânticos e captura de informações de tipo, e a geração de código intermediário responsável pela produção de código de baixo nível não atrelado a nenhuma linguagem ou máquina específica. Para a geração de código para a Máquina Virtual JAVA é sugerido que a árvore de representação intermediária seja percorrida a partir de sua raiz, de modo que para cada conjunto de nodos seja produzida a instrução correspondente na Máquina Virtual JAVA.

O objetivo proposto neste projeto foi cumprido, uma vez que o mesmo poderá ser usado como ferramenta no ensino do curso de compiladores, de forma que cada etapa listada acima possa ser substituída pelos módulos implementados pelo aluno.

A Apêndices

A.1 Sintaxe de Tiger

Antes de apresentarmos a BNF da gramática, ressaltamos alguns pontos-chave para a compreensão da mesma:

- Os não terminais são *exp, decs, dec, tydec, ty, tyfields, tyfields1, vardec, fundec, l-value, type-id, expseq, expseq1, args, args1, idexps.*
- Símbolos terminais aparecem entre aspas.
- Palavras-chave são reservadas e aparecem em negrito.

Gramática Referente à Linguagem Tiger:

$$
\begin{array}{rcl}
exp & ::= & l\text{-}value \mid \textbf{nil} \mid \text{"("}\ expseq\ \text{")"} \mid num \mid string \\
 & & \mid \text{-}\ exp \\
 & & \mid id\ \text{"("}\ args\ \text{")"} \\
 & & \mid exp\ \text{"+"}\ exp \mid exp\ \text{"-"}\ exp \mid exp\ \text{"*"}\ exp \mid exp\ \text{"/"}\ exp \\
 & & \mid exp\ \text{"="}\ exp \mid exp\ \text{"<>"}\ exp \\
 & & \mid exp\ \text{"<"}\ exp \mid exp\ \text{">"}\ exp \mid exp\ \text{"<="}\ exp \mid exp\ \text{">="}\ exp \\
 & & \mid exp\ \text{"\&"}\ exp \mid exp\ \text{"|"}\ exp \\
 & & \mid type\text{-}id\ \text{"\{"}\ id\ \text{"="}\ exp\ idexps\ \text{"\}"} \\
 & & \mid type\text{-}id\ \text{"["}\ exp\ \text{"]"}\ \textbf{of}\ exp \\
 & & \mid l\text{-}value\ \text{":="}\ exp \\
 & & \mid \textbf{if}\ exp\ \textbf{then}\ exp\ \textbf{else}\ exp \\
 & & \mid \textbf{if}\ exp\ \textbf{then}\ exp \\
 & & \mid \textbf{while}\ exp\ \textbf{do}\ exp \\
 & & \mid \textbf{for}\ id\ \text{":="}\ exp\ \textbf{to}\ exp\ \textbf{do}\ exp \\
 & & \mid \textbf{break} \\
 & & \mid \textbf{let}\ decs\ \textbf{in}\ expseq\ \textbf{end}
\end{array}
$$

$$
\begin{array}{rcl}
decs & ::= & dec\ decs \mid \varepsilon \\
dec & ::= & tydec \mid vardec \mid fundec \\
tydec & ::= & \textbf{type}\ id\ \text{"="}\ ty \\
ty & ::= & id \mid \text{"\{"}\ id\ \text{":"}\ type\text{-}id\ tyfields_1\ \text{"\}"} \mid \textbf{array of}\ id \\
tyfields & ::= & id\ \text{":"}\ type\text{-}id\ tyfields_1 \mid \varepsilon \\
tyfields_1 & ::= & \text{","}\ id\ \text{":"}\ type\text{-}id\ tyfields_1 \mid \varepsilon \\
vardec & ::= & \textbf{var}\ id\ \text{":="}\ exp \mid \textbf{var}\ id\ \text{":"}\ type\text{-}id\ \text{":="}\ exp \\
fundec & ::= & \textbf{function}\ id\ \text{"("}\ tyfields\ \text{")"}\ \text{"="}\ exp \\
 & & \mid \textbf{function}\ id\ \text{"("}\ tyfields\ \text{")"}\ \text{":"}\ type\text{-}id\ \text{"="}\ exp
\end{array}
$$

$$
\begin{array}{rcl}
l\text{-}value & ::= & id \mid l\text{-}value\ \text{"."}\ id \mid l\text{-}value\ \text{"["}\ exp\ \text{"]"} \\
type\text{-}id & ::= & id
\end{array}
$$

$$
\begin{array}{rcl}
expseq & ::= & exp\ expseq_1 \mid \varepsilon \\
expseq_1 & ::= & \text{";"}\ exp\ expseq_1 \mid \varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
args & ::= & exp\ args_1 \mid \varepsilon \\
args_1 & ::= & \text{","}\ exp\ args_1 \mid \varepsilon
\end{array}
$$

$$
idexps ::= \text{","}\ id\ \text{"="}\ exp\ idexps \mid \varepsilon
$$

A.2 Exemplos

A.2.1 Exemplo 1

Linha de comando:

```
java Main.Main Testes/teste1.tig -listinput -absyn -intermcode
```

```
Linha de comando:

  java Main.Main Testes/teste1.tig -listinput -absyn -intermcode

================================================================================
1:let
2:      function soma(x1: int, x2: int) : int = x1 + x2
3:      var resultado : int := 0
4:in
5:      resultado := soma(2, 5)
6:end
================================================================================

Listagem da Sintaxe Abstrata:

  LetExp(
   DecList(
    FunctionDec(soma
     Fieldlist(
      x1
      int,
      Fieldlist(
       x2
       int,
       Fieldlist())),
     int
     OpExp(
      PLUS,
      varExp(
       SimpleVar(x1)),
      varExp(
       SimpleVar(x2))),
     FunctionDec()),
    DecList(
     VarDec(resultado,
      int,
      IntExp(0)),
     DecList())),
   SeqExp(
    ExpList(
     AssignExp(
      SimpleVar(resultado),
      CallExp(soma,
       ExpList(
        IntExp(2),
        ExpList(
```

```
        IntExp(5))))))))
Analise Sintatica: OK!

Listagem do Codigo Intermediario:

soma esta rotulada como L0
Parametro "x1" esta associado a "t0"
Parametro "x2" esta associado a "t1"
Variavel "resultado" esta associada a "t2"
Analise Sembntica: OK!

null:
SEQ(
 SEQ(
  EXP(
   CONST 0),
  MOVE(
   TEMP t2,
   CONST 0)),
 MOVE(
  TEMP t2,
  CALL(
   NAME L0,
    TEMP fp,
    CONST 2,
    CONST 5)))

L0:
MOVE(
 TEMP rv,
 BINOP(PLUS,
  TEMP t0,
  TEMP t1))
```

A.2.2 Exemplo 2

Linha de comando:

```
java Main.Main Testes/teste2.tig -listinput -absyn
```

```
================================================================================
1:/* Erro: chamada do procedimento difere dos parametros formais */
2:let
3:      function g (a:int , b:string):int = a
4:in
5:      g("one", "two")
6:end
================================================================================

Listagem da Sintaxe Abstrata:

  LetExp(
   DecList(
    FunctionDec(g
     Fieldlist(
      a
      int,
      Fieldlist(
       b
       string,
       Fieldlist())),
     int
     varExp(
      SimpleVar(a)),
     FunctionDec()),
    DecList()),
   SeqExp(
    ExpList(
     CallExp(g,
      ExpList(
       StringExp(one),
       ExpList(
        StringExp(two)))))))
Analise Sintatica: OK!
================================================================================
 -- Tipo de argumento invalido na chamada da funcao "g". --
 Nome do arquivo:testcases/saida4.tig
 Numero da Linha: 5
 Caractere: 8
================================================================================
Analise Sembntica: ERRO!
```

A.2.3 Exemplo 3

Linha de comando:

```
java Main.Main Testes/teste3.tig -listinput -absyn -intermcode
```

```
1:/* Um exemplo de uso de arranjos e declaracao de novos tipos */
2:
3:let
4:              type a = array of int
5:              type b = a
6:
7:              var arr1:a := b [10] of 0
8:in
9:              arr1[2]
10:end
```

```
Listagem da Sintaxe Abstrata:

  LetExp(
   DecList(
    TypeDec(a,
     ArrayTy(int),
      TypeDec(b,
      NameTy(a))),
    DecList(
     VarDec(arr1,
      a,
      ArrayExp(b,
       IntExp(10),
       IntExp(0))),
     DecList())),
   SeqExp(
    ExpList(
     varExp(
      SubscriptVar(
       SimpleVar(arr1),
       IntExp(2))))))
Analise Sintatica: OK!

Listagem do Codigo Intermediario:

Variavel "arr1" esta associada a "t0"
Analise Sembntica: OK!

null:
EXP(
 ESEQ(
  SEQ(
   EXP(
    CONST 0),
   MOVE(
    TEMP t0,
```

```
  CALL(
   NAME initarray,
    CONST 0,
    CONST 10,
    CONST 0))),
MEM(
 BINOP(PLUS,
  TEMP t0,
  BINOP(MUL,
   CONST 2,
   CONST 4)))))
```

Referências

[1] Aho, A. V.; Sethi, R.; Ullman, J.D., *Compilers Principles, Techniques, and Tools*, Addison Wesley, 1986.

[2] Appel, Andrew W., *Modern Compiler Implementation in JAVA*, Cambridge University Press, 1997.

[3] Entsminger, Gary, *The Way of JAVA*, Prentice Hall, Inc, 1997.

[4] Meyer, Bertrand, *Object-oriented Software Construction*, Prentice Hall, C.A.R. Hoare, 1998.

[5] Johnson, S.C., *Yacc-yet another compiler compiler*, Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[6] Berk, Elliot, *JLex: A Lexical Analyser Generator for JAVA*$^{(TM)}$, http://www.cs.princeton.edu/~appel/modern/java/JLex, 1997.

[7] Hudson, Scott, *LALR Parser Generator for JAVA*^{TM}, http://www.cs.princeton.edu/~appel/modern/java/CUP, 1998.

[8] http://www.cs.princeton.edu/~appel/modern

[9] http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html