Um Compilador para a Linguagem TIGER usando a Linguagem JAVA como Ferramenta de Desenvolvimento

Mariza A. S. Bigonha
Belo Horizonte, 10 de setembro de 2021.

Sumário

1 Introdução

Este documento descreve a linguagem TIGER e seu compilador. TIGER é uma pequena linguagem com funções aninhadas, registros com apontadores implícitos, arranjos, variáveis inteiras e *strings*, além de algumas construções de controle estruturadas.

Organizamos o texto em quatro partes. A primeira parte, descrita na Seção 2, apresenta a sintaxe e semântica da linguagem TIGER. A segunda parte, Seção 3, descreve suscintamente o compilador para a linguagem TIGER [2] em um ambiente JAVA. Esse compilador foi desenvolvido no Laboratório de Linguagens de Programação [9], DCC, com o objetivo de disponibilizar para os alunos de compiladores um compilador modular, completo, de tal forma que os alunos pudessem desenvolver seu próprio compilador TIGER incrementalmente, substituindo no compilador dado, respectivamente, para cada uma de suas fases: análise léxica, análise sintática, análise semântica e geração de código intermediário, o módulo do compilador dado pelo pelo módulo desenvolvido por eles. Esta forma de implementar um compilador é interessante porque o aluno além de poder usar um mesmo conjunto de testes independente da fase do compilador até o momento desenvolvida, ele sempre tem o compilador completo. No fim do projeto o aluno terá todos os seus módulos substituídos no compilador original. A terceira parte, apresentada na Seção 4, descreve como deve ser implementado o compilador para a linguagem TIGER em um ambiente JAVA. A quarta parte, Seção 5, mostra o que cada aluno deve entregar em cada uma das partes do compilador.

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 na Seção 2.4.

2.1 Declarações

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

$$\begin{aligned} \mathit{decs} &:= \mathit{dec}\ \mathit{decs} \mid \mathcal{E} \\ \mathit{dec} &:= \mathit{tydec} \mid \mathit{vardec} \mid \mathit{fundec} \end{aligned}$$

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 são definidos 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 define uma coleção de tipos que pode ser recursiva ou mutuamente recursiva. Tipos mutuamente recursivos são declarados por meio de uma seqűência 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.

Seqűenciamento: uma seqűência 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 seqűência de dígitos decimais é uma constante inteira que denota o valor inteiro correspondente.

Literal string: uma constante *string* é uma seqűência, 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 seqűência de zero ou mais expressões, separadas por ponto-e-vírgula. O resultado, se existir, da última *exp* da seqűência 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)

2.4 A Gramática 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.

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

$$
\begin{array}{rcl}
\textit{decs} & ::= & \textit{dec}\ \textit{decs} \mid \varepsilon \\
\textit{dec} & ::= & \textit{tydec} \mid \textit{vardec} \mid \textit{fundec} \\
\textit{tydec} & ::= & \textbf{type}\ \textit{id}\ \ \texttt{"="}\ \ \textit{ty} \\
\textit{ty} & ::= & \textit{id} \mid \texttt{"\{"}\ \ \textit{id}\ \ \texttt{":"}\ \ \textit{type-id}\ \textit{tyfields}_1\ \texttt{"\}"} \mid \textbf{array of}\ \textit{id} \\
\textit{tyfields} & ::= & \textit{id}\ \ \texttt{":"}\ \ \textit{type-id}\ \textit{tyfields}_1 \mid \varepsilon \\
\textit{tyfields}_1 & ::= & \texttt{","}\ \ \textit{id}\ \ \texttt{":"}\ \ \textit{type-id}\ \textit{tyfields}_1 \mid \varepsilon \\
\textit{vardec} & ::= & \textbf{var}\ \textit{id}\ \ \texttt{":="}\ \ \textit{exp} \mid \textbf{var}\ \textit{id}\ \ \texttt{":"}\ \ \textit{type-id}\ \ \texttt{":="}\ \ \textit{exp} \\
\textit{fundec} & ::= & \textbf{function}\ \textit{id}\ \ \texttt{"("}\ \ \textit{tyfields}\ \texttt{")"}\ \ \texttt{"="}\ \ \textit{exp} \\
& & \mid \textbf{function}\ \textit{id}\ \ \texttt{"("}\ \ \textit{tyfields}\ \ \texttt{")"}\ \ \texttt{":"}\ \ \textit{type-id}\ \ \texttt{"="}\ \ \textit{exp}
\end{array}
$$

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

$$
\begin{array}{rcl}
\textit{expseq} & ::= & \textit{exp}\ \textit{expseq}_1 \mid \varepsilon \\
\textit{expseq}_1 & ::= & \texttt{";"}\ \ \textit{exp}\ \textit{expseq}_1 \mid \varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
\textit{args} & ::= & \textit{exp}\ \textit{args}_1 \mid \varepsilon \\
\textit{args}_1 & ::= & \texttt{","}\ \ \textit{exp}\ \textit{args}_1 \mid \varepsilon
\end{array}
$$

$$
\begin{array}{rcl}
\textit{idexps} & ::= & \texttt{","}\ \ \textit{id}\ \ \texttt{"="}\ \ \textit{exp}\ \textit{idexps} \mid \varepsilon
\end{array}
$$

3 O Compilador para TIGER

O compilador TIGER foi implementado usando o paradigma orientado por objeto [4]. A escolha por este paradigma se explica pela necessidade de desenvolver a ferramenta o mais modular possível, o que, na orientação por objeto, vem a ser uma premissa básica. Organizamos a implementação de TIGER em seis etapas, as quais listamos a seguir:

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

O interpretador para a Máquina Virtual Java (MVJ) ainda não está disponível para uso, portanto o compilador desenvolvido vai até o código intermediário.

Para desenvolver o compilador TIGER foram utilizadas as ferramentas JLex [6] e CUP [7].

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.

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

3.1 Estruturas de Dados

3.1.1 Sintaxe Abstrata

É possível em CUP, YACC, ou em outras ferramentas de geração de compiladores juntar as fases de análise sintática e semântica. Contudo, no compilador TIGER desvinculamos os procedimentos referentes à análise sintática daqueles referentes à análise semântica de modo a propiciar o desenvolvimento modular do compilador. Geramos uma árvore de sintaxe abstrata. Esta árvore 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.
Por exemplo, para o programa TIGER: `(a := 5; a + 1)` geramos a seguinte sintaxe abstrata:

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

3.1.2 Tabela de Símbolos

A Tabela de Símbolos é uma estrutura utilizada pelo compilador para controlar as informações de tipo e escopo dos nomes utilizados em um programa. Na implementação de TIGER foram usadas tabelas hash devido à necessidade de se fornecer acesso rápido e eficiente aos símbolos. Um símbolo em TIGER pode assumir um dos tipos: `int, string, record, array, nil, void, name`.

3.1.3 Registro de Ativação

Com o objetivo de manter a construção do compilador modular, introduzimos uma camada de abstração representativa do registro de ativação, de modo a separar a semântica da linguagem fonte da representação efetiva do registro de ativação dependente da linguagem alvo.

É comum em muitas linguagens de programação ter diversas invocações de função 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. Esta estrutura de dados é conhecida como uma pilha de registros de ativação. Na prática, ela nada mais é do que um grande arranjo que pode expandir-se ou contrair-se indefinidamente, associado a um registrador especial, o *stack pointer*, que aponta para uma posição específica da pilha.

Na implementação de TIGER, a pilha inicia-se em endereços de mais alta ordem na memória, crescendo em direção a endereços de mais baixa ordem, de modo que, todas as posições além do *stack pointer* são consideradas lixo, e todas anteriores são tidas como alocadas. A Figura 1 apresenta sua estrutura.

No registro de ativação há uma área para variáveis locais; uma área para armazenar o endereço de retorno; outra para armazenar valores quando estes não puderem ser alocados em registradores; e uma área para passagem de parâmetros.

3.2 Descrição das Fases de Compilação

Nesta seção são descritas para cada uma das fases do compilador como elas foram implementadas em Java e é dessa mesma forma que vocês implementarão o compilador de vocês em Java.

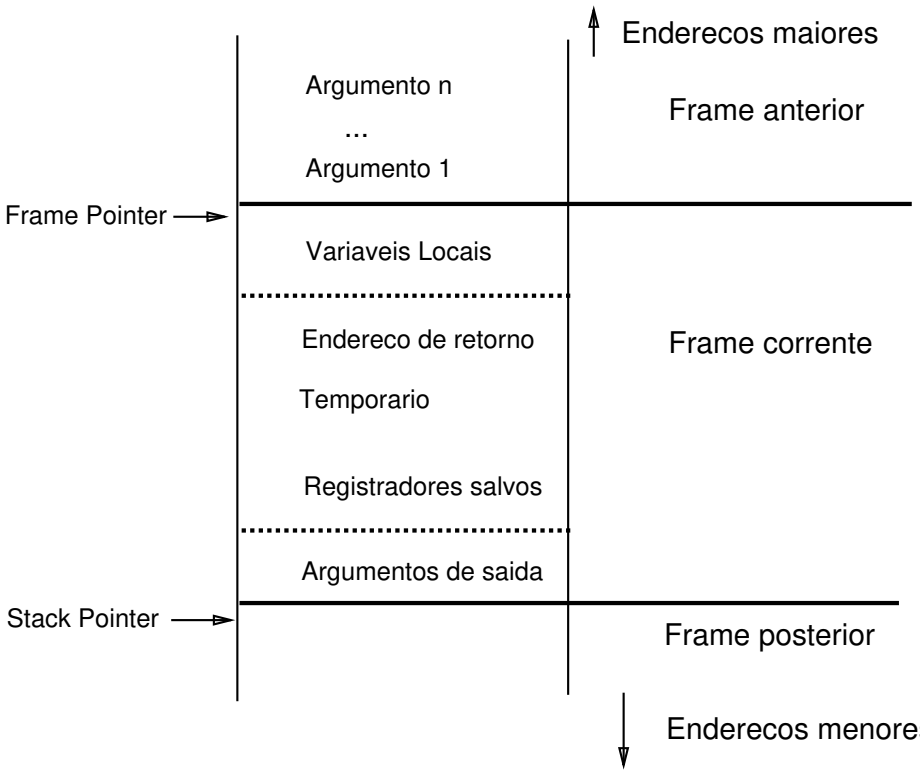

Figura 1: Estrutura do Registro de Ativação

3.2.1 Análise Léxica

Esta fase é responsável pelo agrupamento de seqűências de caracteres em *tokens*. Para implementá-la, foi utilizada a ferramenta JLex. Essa ferramenta recebe como entrada o arquivo de especificação Tiger.lex. Esse 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 em 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 a geração do analisador léxico, executa-se o comando `java JLex.Main Tiger.lex` de dentro do diretório `Parse` e obtem-se o arquivo `Tiger.lex.java`, o qual é renomeado para `Yylex.java` para manter a compatibilidade com o restante do programa.

Para gerar o arquivo Yylex.class o projetista do compilador deve executar o comando `javac Parse/Yylex.java` de dentro do diretório mais externo,

`/~mariza/Cursos/CompiladoresI/Geral/Projeto2021-2/Proj2021-2`

Por exemplo:
`/~mariza/Cursos/CompiladoresI/Geral/Projeto2021-2/Proj2021-2/javac Parse/Yylex.java`

3.2.2 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 4.2.2 é uma ferramenta para gerar analisadores sintáticos, similar à difundida Yacc [5].

CUP recebe como entrada um arquivo contendo a especificação da gramática para a qual é necessária a construção do analisador, Tiger.cup, juntamente com rotinas responsáveis pela invocação do analisador léxico. A especificação desse 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.

Em TIGER, para a geração do analisador foi utilizada 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
```

de dentro do diretório `Parse` e obteve-se os arquivos `Grm.java` e sym.java. O arquivo `CUP$Grm$actions.class` é gerado na fase se análise semântica.

A diretiva `-expect 6` é usada para ignorar os *warnings* durante a geração do analisador.

Para gerar os arquivos `Grm.class` e `sym.class` o projetista do compilador deve executar os comandos
`javac Parse/sym.java` e `javac Parse/Grm.java`
de dentro do diretório mais externo:

`/~mariza/Cursos/CompiladoresI/Geral/Projeto2021-2/Proj2021-2`

Por exemplo:
`/~mariza/Cursos/CompiladoresI/Geral/Projeto2021-2/Proj2021-2/javac Parse/Grm.java`

3.2.3 Análise Semântica e 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 subseqűente 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. O erro semântico ocorre porque 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, foi 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 do código intermediário correspondente.

As informações disponíveis para 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*.

Para gerar o arquivo`CUP$Grm$actions.class` o projetista do compilador deve executar o comando `javac Parse/Grm.java` de dentro do diretório mais externo, por exemplo:

```
/~mariza/Cursos/CompiladoresI/Geral/Projeto2021-2/Proj2021-2
```

```
/~mariza/Cursos/CompiladoresI/Geral/Projeto2021-2/Proj2021-2/ javac
Parse/Grm.java
```

durante a análise sintática, já que o arquivo `CUP$Grm$actions.class` está relacionado com as rotinas semânticas associadas às produções da gramática.

4 Metodologia para o Desenvolvimento do seu Compilador TIGER

O seu compilador para TIGER pode ser implementado na linguagem JAVA, C ou C++, de preferência em JAVA, mas antes de começar a projetá-lo, você deve se familiarizar com a linguagem TIGER. Para isso está disponível no endereço eletrônico:

```
/~mariza/Cursos/CompiladoresI/Geral/Projeto2021-2/Proj2021-2
```

os pacotes com as classes responsáveis pelas funcionalidades do compilador TIGER. A Seção 4.1 descreve cada um desses pacotes e mostra como usar o compilador.

4.1 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 esses 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 booleanos.

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

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

noindent onde:

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

Para redirecionar a saída das opções para um arquivo basta digitar a seguinte linha de comando:

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

onde:
`<nome-arquivo-saida>`: é o arquivo contendo a saída das opções requisitadas.

4.2 Ferramentas de Apoio

4.2.1 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.

4.2.2 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` [1]

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`.

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

5 Trabalho Prático: Compilador TIGER

5.1 TP1: Analisador Léxico

Cada aluno de compiladores deve implementar um analisador léxico para a linguagem TIGER. O aluno deve entregar para essa fase do compilador:

- O arquivo fonte Ylex.lex e demais arquivos se existir.
- A documentação.

A documentação deve conter:

- Nome dos componentes do grupo.
- Relato dos erros encontrados na especificação, se houver.
- Questões não solucionadas nessa etapa do trabalho.
- Todas as suposições feitas com relação as partes da especificação que não ficou clara.
- Instruções para testar o analisador léxico, por exemplo, se estiver em sua conta, como acessá-lo.
- Um resumo do seu projeto.
- Descrição do objetivo de cada classe, método em seu programa. Esta parte deve ser organizada hierarquicamente.

5.2 TP2: Analisador Sintático

Cada aluno de compiladores deve implementar neste segundo trabalho prático um analisador sintático para a linguagem TIGER de acordo com a gramática fornecida neste texto. Essa parte do compilador está coberta nas páginas 83-84 [2]. Pode ser útil ler a Seção 3.4 desse mesmo livro.

Nessa parte do trabalho o aluno deve entregar:

- O arquivo fonte Grm.cup e demais arquivos fontes necessários no seu programa.
- A documentação.

A documentação deve conter:

- Nome dos componentes do grupo.
- Relato dos erros encontrados na especificação, se houver.
- Questões não solucionadas nessa etapa do trabalho.

- Todas as suposições feitas com relação as partes da especificação que não ficou clara.
- Instruções para testar essa etapa do programa, por exemplo, se estiver em sua conta, como acessá-lo.
- Um resumo do seu projeto.
- Descrição do objetivo de cada classe, método em seu programa.
- Listagem de cada conflito *shift-reduce*, se houver, na gramática com uma explicação de como cada um deles foi resolvido e porque a solução adotada é uma solução correta.

5.3 TP3: Análise Semântica e Geração de Código Intermediário

Cada aluno de compiladores deve fazer neste terceiro trabalho prático a análise semântica e a geração do código intermediário. Essa fase cobre a construção da árvore de sintaxe abstrata, Página 106 e Capítulo 4 do livro [2], e a verificação de tipos, geração de código intermediário para a linguagem TIGER de acordo com a gramática fornecida nesse texto. Leia também os capítulos 5, 6 e 7 de [2] para auxiliá-lo.

Se eu fosse você, eu faria esse trabalho da seguinte forma:

1. Daria uma olhada em Absyn/Print.java para entender a estrutura hierárquica da árvore de sintaxe abstrata e para saber como usar a *pretty printer*.
2. Geraria a gramática para TIGER em Grm.cup.
3. Rodaria CUP com a gramática para TIGER, Grm.cup.
4. Adicionaria as rotinas semânticas para construir a árvore de sintaxe abstrata em Grm.cup.
5. Adicionaria as rotinas semânticas para fazer a verificação de tipos.

Nessa parte do trabalho o aluno deve entregar:

- Todos arquivos fontes necessários para rodar seu programa.
- A documentação.

A documentação deve conter:

- Nome dos componentes do grupo.

- Relato dos erros encontrados na especificação, se houver.
- Questões não solucionadas nessa etapa do trabalho.
- Todas as suposições feitas com relação as partes da especificação que não ficou clara.
- Instruções para testar essa etapa do programa, por exemplo, se estiver em sua conta, como acessá-lo.
- Um resumo do seu projeto.
- Descrição do objetivo de cada classe, método em seu programa.
- Uma explicação de como foi tratado os comandos `break` aninhados nos comandos *while* ou *loop*.
- Uma explicação de como foi tratado tipos recursivos e declarações de funções.
- Listagem dos fontes: Semantic.java e Env.java.

6 Conclusões

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 nós seja produzida a instrução correspondente na Máquina Virtual JAVA.

Essa parte não será cobrada no curso, mas aqueles que a fizerem receberão alguns pontos extra.

A Apêndice

A.1 Exemplos

A.1.1 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 est\'a rotulada como L0
Par\^ametro "x1" est\'a associado a "t0"
Par\^ametro "x2" est\'a associado a "t1"
Vari\'avel "resultado" est\'a associada a "t2"
An\'alise Sem\^antica: 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.1.2 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 inv\'alido na chamada da fun\c{c}\~ao "g". --
 Nome do arquivo:testcases/saida4.tig
 N\'umero da Linha: 5
 Caractere: 8
======================================================================
               Analise Semantica: ERRO!
```

A.1.3 Exemplo 3

Linha de comando:

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

```
======================================================================
1:/* Um exemplo de uso de arranjos e declara\c{c}\~ao 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 Semantica: 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://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

[9] Escalda, G. e Bigonha, M., *Um compilador para a linguagem TIGER usando a Linguagem Java como Ferramenta de Desenvolvimento*, Relatório Técnico do Laboratório de Linguagens de Programa'cão, LLP11/99.

[10] Bigonha, Mariza A. S., SIC: Sistema de Implementação de Compiladores, Tese de Mestrado, DCC/UFMG, 1984.