Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Informática

Compiladores
2010/2011 – 2º Semestre

Compilador da Linguagem PJAVA
Meta Final

Realizado por:

Daniel Francisco Lopes de Carvalho
Nº2008108761
dflopes@student.dei.uc.pt

Pedro Miguel Ricardo Geadas
Nº2006131902
pmrg@student.dei.uc.pt

Índice

Descrição geral do projecto

Com a elaboração do presente relatório pretendemos explicar e apresentar de uma forma clara e objectiva, todo o trabalho realizado para o desenvolvimento do compilador PJAVA, baseado na linguagem Java. A criação do mesmo passou por várias fases, nomeadamente: análise lexical, análise de sintaxe, construção da árvore de sintaxe abstracta, análise semântica e tradução para código C final.

A nossa linguagem dispõe de cinco tipos de dados elementares (integer, float, double, boolean e String).

Relativamente às *Strings*, será possível executar a operação aritmética mais comum sobre elas, a concatenação. Para isso criámos uma função, ***System.concat(String ID,type t1, (…) type tn)***, que irá receber como parâmetros uma variável do tipo String (ID), para guardar o resultado da concatenação, e pode receber vários tipos *(type)* para concatenar, que podem ser strings ou outro tipo qualquer, na verdade. Optamos por criar esta função devido à estrutura do nosso programa, uma vez que não considerámos o facto da concatenação em C ser completamente diferente do JAVA. Assim sendo, alterar toda a parte da aritmética (que envolve neste caso o operador +), iria trazer grandes mudanças no nosso código. Deste modo, e para garantir todas as funcionalidades no nosso compilador, decidimos criar esta função.

No nosso trabalho final, conseguimos permitir toda a gramática base da linguagem original Java, da parte procedimental pedida, desde as *estruturas de controlo if, else if e else*, *estruturas de repetição for e while*, até às *expressões lógicas*, *relacionais* e *aritméticas*, passando pela *definição e chamada de funções*, com ou sem argumentos, e ainda *instruções de impressão* para ecrã.

No caso da *definição de funções*, não é permitida a sua definição dentro de outras funções, tal como em Java.

No caso das *instruções de controlo* (if, else if e else), funciona como no JAVA: se vier só uma instrução para executar, não necessita chavetas; caso existam várias, as chavetas têm de ser colocadas.

No caso das *funções de impressão* para ecrã, criámos duas funções distintas: ***System.out.print([arg_list]) e System.out.println([arg_list])***; ambas funcionam como no JAVA, podendo receber diferentes tipos de operações (uma lista de argumentos, *[arg_list]*), e procede à sua concatenação, aceitando aqui o operador + para o efeito. A diferença entre as duas, tal como no JAVA, refere-se ao caractere de mudança de linha, que na primeira função é inexistente.

Implementámos também a parte dos ***comentários*** (uni e multi-linha), ***leituras de input***, nos ciclos as ***instruções de break e continue***, e ainda a possibilidade das ***funções serem do tipo void***, caso em que não retornam nada.

No caso das *leituras de input*, resolvemos criar também duas funções distintas: ***System.in.read([VAR]) e System.in.readln([VAR]).*** Cada uma destas funções recebe como parâmetro uma variável, onde vai guardar o que foi lido, sendo distinguidas pelo facto de *a primeira ler um valor*, de um dos vários tipos excepto boolean, enquanto que *a segunda lê toda uma linha.* De modo a simplificar e a tornar a nossa linguagem mais dinâmica, consoante o tipo da variável passado às funções de leitura, será então o tipo que vai ter que ler do input (Exemplo: Passamos int VAR à função, ficando: System.in.read(VAR); → indica-nos que vai ler um inteiro, e guarda-o em VAR).

Na *chamada de funções*, podemos também passar como argumentos outras funções,

desde que essas retornem o tipo esperado como parâmetro pelas primeiras.

A parte do *return* está também ela de acordo com o Java, podendo as funções retornar qualquer um dos tipos existentes, bem como retornar o valor de retorno da execução de outra função (Exemplo: return soma(a,b); → retorna o valor retornado pela função soma()). Esta funcionalidade também está implementada para todas as *estruturas de repetição* (if, for, while), em que é possível efectuar determinada acção com base no valor de retorno de uma função. Ainda acerca da *estrutura for*, é possível inicializar uma ou várias variáveis no mesmo, ou utilizar variáveis previamente inicializadas no código PJAVA.

Na análise semântica, é então detectada a existência ou não de erros no código, types mismatch, etc.., que citaremos mais adiante.

Durante toda a elaboração do projecto, fomos utilizando sempre as estruturas aconselhadas, presentes ao longo das fichas semanais, com algumas modificações necessárias, as quais iremos explicar melhor também mais à frente.

Gramática da linguagem PJAVA e sintaxe abstracta

Para melhor compreensão e análise do compilador realizado, deixamos no final do presente relatório dois anexos contendo toda a gramática do compilador assim como a especificação da sintaxe abstracta, anexos A e B, respectivamente.

Passemos, no entanto, a explicar os blocos mais importantes da gramática, muito resumidamente:

Um programa é composto por uma classe *(<class>)*. Dentro dessa classe, podemos encontrar várias declarações *(<declaration>)*, neste caso correspondentes às variáveis globais, e uma lista de definições de funções *(<func_list>)*, que no mínimo terá de conter uma função *main*, a função principal *(<func_def>)*.

Cada *<declaration>*, tem uma lista de variáveis a ser declarada *(<dec_list>)*, que consiste numa atribuição/declaração, ou em várias *(<atr>)*. Cada <atr> é então composto por um *<ID>*, caso em que só foi declarada, ou por uma atribuição propriamente dita *(<atribution>)*, caso em que é declarada e inicializada. Em cada <atribution>, encontramos uma lista *(<atribution_list>)* de atribuições, ou apenas uma atribuição no caso de ser só uma, e uma expressão a atribuir *(<expression>)*. A nossa <expression> contém então as expressões aritméticas *(<aritmetic_exp>)* , lógicas *(<logic_exp_not, logic_exp_or_and>)*, relacionais *(<relac_exp>)*, dando prioridade a expressões dentro de parênteses, ou pode ser um tipo *(<type>)*. Cada <type> pode então ser um *Integer, Float, Double, String, Boolean, um ID* de uma variável, ou ainda a *chamada de uma função (<func_call>)*. A uma <func_call> corresponde portanto um ID (nome da função), e uma lista de argumentos, que pode ser vazia *(<call_list>)*. Cada elemento da <call_list> corresponde a um <type>, e contempla todos os tipos, incluindo chamadas a funções desde que estas retornem o tipo esperado, obviamente.

Cada <func_def> é composta por um ID (nome da função), seguido de uma lista de argumentos *(<arg_list>)* e um bloco respeitante ao código da função *(<cont_func>)*. Em <cont_func> podemos encontrar uma lista de statements *(<statement_list>)*, que pode também ser vazia. Um <statement_list> contém vários statements *(<statement>)*. Cada bloco <statement> poderá ser uma instrução *(<instruction>)* , um bloco composto *(<compound_stmt>)*, uma operação de print *(<println_statement>)*, uma declaração *(<declaration>)*, uma chamada de função *(<func_call>)*, uma instrução de retorno *(<return_statement>)*, uma instrução de leitura *(<read_statement>)*, uma operação de concatenação *(<concat>),* ou ainda uma instrução de break ou continue (<break_cont>).

Cada <compound_statement> será uma estrutura de controlo if *(<if_statement>)*, for *(<for_statement>)* ou while *(<while_statement>)*. Uma instrução *(<instruction>)* consiste numa instrução de linha única que poderá ser uma das seguintes: expressão *(<expression>)*, atribuição *(<atribution>)*, operação do tipo ,por exemplo, a++ ou a--; *(<plusminus>)*. O <println_statement> tem uma <instruction>, para imprimir, e o <return_statement> por sua vez também, mas ao invés de imprimir, vai ser o resultado de retorno de uma função. Finalmente, o nosso <read_statement> irá receber um ID, onde irá guardar aquilo que for lido do teclado.

Análise Semântica

Nesta secção serão apresentadas as soluções adoptadas e os tipos de erros detectados, bem como as structs que armazenam determinada informação/símbolos.
Sendo assim, na análise semântica, tratamos da detecção de erros de semântica que será efectuada através de uma passagem.

Tabela de símbolos e ambientes

Para esta fase do compilador, surge a necessidade de termos uma estrutura de dados que
suporte o armazenamento de cada símbolo utilizado no código. Para tal criamos a seguinte
estrutura:

```
typedef struct _table_element{
        char name[32];              // nome
        disc_type type;             // tipo
        int offset;                 // offset respectivo
        struct _table_element *next; // ponteiro para o next element
} table_element;
```

Com a estrutura acima podemos então guardar o nome do símbolo (name), o tipo do símbolo
em causa (type), o offset que o símbolo terá na frame e um ponteiro para o próximo elemento da tabela.

Cada programa na linguagem PJAVA é representado por um ambiente global único que
contem todos os restantes ambientes dentro dele, respeitante a funções, e a lista de variáveis globais.

```
typedef struct _prog_env{
        table_element* global; // variáveis globais
        environment_list* procs; // funções
}prog_env;
```

Para a representação de cada ambiente criamos a seguinte estrutura de dados:

```
typedef struct _env_list{
        char* name;         //nome
        int offset;         //offset
        int passagem;       // flag que diz o número da passagem, 1 ou 2
        int total_args;     // total d argumentos da função
        disc_type type;     // tipo de retorno
        struct _env_list *next;  //ponteiro para a próxima função
        table_element *locals; //variáveis locais à função
        table_element *arguments; // argumentos
}environment_list;
```

Com esta estrutura podemos saber o nome da função a que diz respeito o

ambiente (name), saber o offset respectivo, o número de argumentos da função (total_args) e o tipo de retorno da funçao (type). Temos também aqui um ponteiro para as variáveis locais à função (locals) bem como dos seus argumentos (arguments).

A análise semântica é efectuada através de uma passagem pela AST. O facto de efectuarmos
apenas uma passagem não permite efectuar a definição de uma função no fim do código e chamar essa função no início. Apenas nos apercebemos deste facto já no final o que não nos deu tempo de alterar no nosso compilador até à data limite de entrega, já que as alterações em todo o código seriam profundas, pois seriam necessárias duas passagens pela árvore nesta parte da análise semântica, para detectar as funções existentes.

Caso seja detectado algum erro, o programa pára e não será gerado qualquer tipo de código C.

Tipos de erros detectados

A detecção de erros efectuada na análise semântica é bastante completa. As regras obedecem quase todas à linguagem de base, o JAVA, no entanto, tivemos de impor uma ou duas restrições também devido a situações com que nos deparámos já numa fase avançada do projecto, como já foi de resto referido acima.

Para simplificar a explicação dos erros detectados, fica de seguida uma lista de tópicos indicando os tipos de erros detectados:

• Duplicação de símbolos/variáveis em ambientes comuns.
• Não definição de uma símbolo/variável que se tente usar.
• Utilização de identificadores inexistentes.

• Operações com tipos incompatíveis.
• Declaração de tipos de dados inválidos.
• Erros na leitura de dados do input. (Não é possível ler do tipo Boolean)

• Não existência da função main, ou existência de erros na mesma.
• Chamadas de funções inexistentes.
• Quantidade de parâmetros de uma função (na chamada).
• Incompatibilidades nos tipos de argumentos passados a uma função.

• Operações lógicas inválidas (&&, || e !).
• Operações aritméticas inválidas (+, -, *, / e %). (Não é possível fazer uma operação aritmética sobre Strings, para o fazer deve ser usada a função **System.concat(String destino, qualquer tipo, qualquer tipo, …….);,** como já foi especificado na Secção 1 (Descrição Geral) deste relatório).

• Retorno de tipos de dados diferentes numa chamada de função.
• Falta de retorno numa função não void.
• Retorno de uma função void.

• Uso de break/continue fora de um ciclo.

Na apresentação de erros é indicado o número total e a respectiva linha do código PJAVA onde os mesmos se encontram. Para manter esta funcionalidade, no caso dos comentários multi-linha tivemos de criar um estado, <COMENTARIOS>, de modo a incrementar correctamente o número da linha em que nos encontramos.

Tradução para código final

Nesta secção serão apresentadas as estruturas de dados criadas para a tradução de código final e as soluções adoptadas.

É nesta última etapa que passamos o código original (PJAVA) para o nosso código final em C restringido. Para tal, o nosso compilador irá criar um ficheiro de nome result.c, que à posterior poderá ser compilado e executado.

Nesta etapa foi criada uma estrutura de dados adicional que apresentamos em baixo.

```
typedef struct _frame{
        struct _frame* parent;      //frame pointer - ambiente da função chamante
        void* locals[64];           //espaço de endereçamento para variáveis locais
        void* outgoing[32];         //espaço de endereçamento para argumentos de funções chamadas
        char* frame_buffer;         //serve para guardar conteudos do concat e println
        int return_address;         //endereço do código na função chamante
}frame;
```

Esta struct servirá para guardar as variáveis em memória necessárias à execução do nosso programa, sendo que estará ligada ao código final do programa compilado e não ao nosso compilador.

Foi necessária a criação de um buffer por frame, para ir guardando os argumentos dos prints, concat e reads.

Durante esta fase de tradução do código PJAVA para código C, fomos obrigados a fazer duas passagens em certas partes da AST, como por exemplo nas operações aritméticas, de forma a resolver as chamadas de funções antes de atribuições, somas, operações de retorno, entre outras. No fundo o que acontece no código C final é que a função é chamada, resolvida, é guardado o valor de retorno no array return_value e por fim usa esse return_value no lugar da chamada de função.

Estrutura do programa final

O nosso programa final será constituído apenas pela função main, sendo que assim para a
simulação das chamadas das diferentes funções serão usadas as funcionalidades do C, label e
goto. No inicio do programa é necessário importar algumas livrarias do C, assim como a nossa frame:

```
#include "frame.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Buffer

No caso da manipulação de Strings no nosso programa, bem como para ler do input, encontrámos algumas dificuldades. As mesmas foram superadas pelo recurso a buffers auxiliares, para onde copiamos a informação que vai sendo lida e fazemos a devida concatenação, se for caso disso. De modo a que o programa funcione correctamente, procedemos à alocação do espaço necessário tanto dos buffers, como da variável que vai receber a informação, dinamicamente. Tivemos de recorrer às funções de manipulação de Strings do C, strcpy(), strcat() e strlen() para efectuar a concatenação propriamente dita, e ao scanf() e fgets() para o caso das leituras do teclado.

Variáveis Temporárias

No nosso ficheiro final temos também algumas variáveis temporárias definidas. Para efeitos de tratamento do return, temos um array com os diferentes valores dos returns guardados, uma variável _ra, que nos indica qual o endereço de retorno da função actual e um _last_ra que guarda o endereço de retorno da última função, ou ainda a flag_else, que nos indica, no caso de uma instrução que contenha else, se já entrou em algum if/else if previamente, caso em que já não entrará na parte else.

ANEXO A – GRAMÁTICA

```
program:            CLASSSYMB ID '{' class '}'
;

class:              declaration ';' class
|                   func_list
;

func_list:          func_list func_def
|                   func_def
;

statement_list:     statement_list      statement
|                   statement
;

statement:          instruction ';'
|                   compound_statement
|                   println_statement ';'
|                   declaration ';'
|                   func_call ';'
|                   return_statement ';'
|                   read_statement ';'
|                   concat ';'
|                   break_cont ';'
;

break_cont:         BREAK
|                   CONTINUE
;

concat:             CONCATSYMB '(' concat_args ')'
;

concat_args:        type ',' concat_args
|                   type ',' type
;

read_statement:     READ '(' ID ')'
|                   READLINE '(' ID ')'
;

instruction:        expression
|                   atribution
|                   plusminus
;

Plusminus:          ID PLUSPLUS
|                   ID MINUSMINUS
;

compound_statement: if_statement
|                   while_statement
|                   for_statement
;

println_statement:  PRINTLN '(' println_args ')'
|                   PRINT '(' println_args ')'
;

println_args:       type PLUS println_args
|                   type
;

return_statement:   RETURNSYMB
|                   RETURNSYMB instruction
;
```

```
for_statement:          FOR '(' for_args ';' relac_exp ';' for_args ')' cont_statement
;

for_args:               dec_list
|                       declaration
|                       plusminus
|                       NULL
;

if_statement:           IF '(' instruction ')' cont_statement elseif_statement_list else_statement
;

while_statement:        WHILE '(' instruction ')' cont_statement
;

else_statement:         ELSE cont_statement
|                       NULL
;

elseif_statement_list:  elseif_statement_list elseif_statement
|                       NULL
;

elseif_statement:       ELSEIF '(' instruction ')' cont_statement
;
cont_statement:         statement
|                       '{' statement_list '}
|                       '{' '}'
|                       ';'
;

func_def:               FUNCSYMB symbols ID '(' arg_list ')' cont_func
;
func_call:              ID '(' call_list ')'
;

call_list:              call_list ',' type
|                       type
|                       NULL
;

cont_func:              '{' statement_list '}'
|                       '{' '}'
;

arg_list:               arg_list ',' symbols ID
|                       symbols ID
|                       NULL
;


symbols:                INTSYMB
|                       DOUBLESYMB
|                       FLOATSYMB
|                       STRINGSYMB
|                       VOIDSYMB
|                       BOOLSYMB
;


declaration:            INTSYMB dec_list
|                       DOUBLESYMB dec_list
|                       FLOATSYMB dec_list
|                       STRINGSYMB dec_list
|                       BOOLSYMB dec_list
;

dec_list:               atr
|                       dec_list ','  atr
;

atr:                    ID
|                       atribution
;

atribution:             atribution_list expression
;
```

```
atribution_list:        atribution_list ID EQUAL
|                       atribution_list ID PLUSEQUAL
|                       atribution_list ID MINUSEQUAL
|                       ID EQUAL
|                       ID MINUSEQUAL
|                       ID PLUSEQUAL
;

expression:             exp_minus_plus
|                       '(' expression ')'
|                       logic_exp_or_and
|                       relac_exp
|                       aritmetic_exp
|                       logic_exp_not
|                       type
;

exp_minus_plus:         MINUS expression
|                       PLUS expression
;

logic_exp_not:          NOT expression
;

logic_exp_or_and:       expression AND expression
|                       expression OR expression
;

relac_exp:              expression EQUALEQUAL expression
|                       expression NOTEQUAL expression
|                       expression GREATEREQUAL expression
|                       expression LESSEQUAL expression
|                       expression LESS expression
|                       expression GREATER expression
;

aritmetic_exp:          expression PLUS expression
|                       expression MINUS expression
|                       expression MULT expression
|                       expression DIV expression
|                       expression MOD expression
|                       expression POT expression
;

type:                   INT
|                       FLOAT
|                       STRING
|                       ID
|                       func_call
|                       BOOL
;
```

ANEXO B – Sintaxe Abstrata

is_program --> (<statments: is_class>)

is_class --> (<class: is_class><declaration:is_declaration><func_list: is_func_list>)

is_func_list --> (<func_list: is_func_list><func_def: is_func_def>)

is_statement_list --> (<statment_list: is_statement_list><statement: is_statement>)

is_statement --> is_instruction V is_compound_statement V is_println_statement V is_declaration V is_func_call V return_statement V read_statement V is_concat V is_break_cont

is_break_cont --> is_BREAK V is_CONTINUE

is_concat: (<argumentos: is_concat_args>)

is_concat_args: (<concat_args: is_concat_args><type: is_type>)

is_read_statement --> is_READ V is_READLINE

is_instruction --> is_atribution V is_expression V is_plusminus

is_plusminus --> is_PLUSPLUS V is_MINUSMINUS

is_compound_statement --> is_if_statement V is_while_statement V is_for_statement

is_println_statement --> is_PRINTLN V is_PRINT

is_return_statement --> is_instruction V NULL

is_func_def --> (<return_type: is_symbols><nome_func: is_ID><parametros: is_arg_list><cont_func: is_cont_func>)

is_symbols --> is_INTSYMB V is_DOUBLESYMB V is_FLOATSYMB V is_STRINGSYMB V is_VOIDSYMB V is_BOOLSYMB

is_if_statement --> (<instr:is_instruction><iconts:is_cont_statement><ieisl:is_elseif_statement_lis><ielses:is_else_statement>)

is_while_statement --> (<instr:is_instruction><iconts:is_cont_statement>)

is_else_statement --> is_cont_statement V NULL

is_elseif_statement --> (<instr: is_instruction><iconts:is_cont_statement>)

is_elseif_statement_list --> (<instr: is_statement_list>)

is_cont_statement --> is_statement V is_statement_list

is_declaration --> is_dec_list

is_dec_list --> (<dec_list: is_dec_list><atr: is_atr>

is_atr --> is_atribution V is_id

is_atribution --> (<atrib_list:is_atribution_list><operacao:is_expression>)

is_atribution_list --> (<atribution_list: is_atribution_list><nome_atr: is_ID>

is_expression --> is_exp_minus_plus V is_expression V is_logic_exp_or_and V is_logic_exp_not V is_aritmetic_exp V is_relac_exp V is_type

is_exp_minus_plus --> (<exp:is_expression><oper: is_MINUS V is_PLUS>)

is_logic_exp_not --> (<exp:is_expression><oper:is_oper>)

is_logic_exp_or_and --> (<exp1:is_expression><oper:is_oper><exp2:is_expression>)

is_relac_exp --> (<exp1: is_expression><oper:is_oper_r><exp2:is_expression>)//is_IGUAL V is_MENOR V is_MENOR_IGUAL V is_MAIOR_IGUAL V is_MAIOR V is_DIFERENTE

is_aritmetic_exp --> (<exp1: is_expression><oper:is_oper_a><exp2:is_expression>)

is_type --> is_INTEGER V is_FLOAT V is_DOUBLE V is_STRING V is_func_call V is_BOOL

is_func_call --> (<nome: is_ID><argumentos: is_call_list>)

is_oper_a --> is_PLUS V is_MINUS v is_MULT v is_DIV v is_MOD v is_POT

is_oper_r --> is_GREATER v is_GREATEREQUAL v is_LESS v is_LESSEQUAL v is_EQUAL v is_NOTEQUAL

is_oper_l --> is_AND v is_OR v is_NOT

is_id --> (<id:char>)

is_id_list

is_println_args

is_call_list