COMPILADOR PARA KPASCAL

Compiladores

Pedro Miguel de Almeida Verruma
verruma@student.dei.uc.pt
Nº: 2006128853

INTRODUÇÃO

Este projecto tem como principal objectivo criar um compilador para *KPascal* que foi definido como sendo um *sub-set* da conhecida linguagem *Pascal*.

Durante o processo de desenvolvimento tomou-se contacto com diversas técnicas bem conhecidas para tornar o processo de desenvolvimento de um compilador numa tarefa mais simples e eficaz.

De um modo geral foi seguida a abordagem utilizada nas aulas, mas todo o processo será descrito em detalhes nas páginas seguintes.

DESCRIÇÃO GERAL

Tal como foi referido, o projecto segue os exemplos dados nas aulas. Assim sendo divide-se em quatro grandes áreas, cada qual com tamanho e importância diferentes:

- Análise Lexical
- Análise Sintáctica
- Análise Semântica
- Tradução de Código

Estas secções serão descritas em detalhe mais à frente. Posteriormente serão apresentados alguns exemplos de código em *KPascal* que o compilador valida e em anexo são listados os erros de semântica que o compilador detecta.

Também é descrito de um modo geral a linguagem que o compilador aceita (além de se apresentar a especificação da gramática), para que seja mais simples identificar a sintaxe de cada instrução. Para que se tenha uma ideia aqui fica uma listagem do que foi implementado na totalidade (com validação sintática e semântica completas):

- Tipos de variáveis: inteiros, reais e booleans;
- If-then-else com blocos simples e complexos;
- Writeln com possbilidade de aceitar qualquer número de parâmetros (expressões ou strings);
- While com blocos complexos;
- Funções que podem ser chamadas no corpo principal do programa bem como dentro de outras funções (recursividade);
- Expressões lógicas (<, >, <=, >=, =, <>) encadeadas (and, or) com base em expressões (+, -, *, div, mod).

ANÁLISE LEXICAL

Nesta primeira fase foi utilizada a ferramenta LEX para determinar palavras-chave, nomes de variáveis e funções (identificadores), números reais e inteiros, símbolos aritméticos e lógicos e ainda a linha em que cada instrução se encontra.

Todas estas informações são depois passadas ao YACC que prossegue com a análise sintáctica descrita na secção seguinte.

Tokens utilizados na linguagem:

- PROGRAM, VAR, FUNCTION, ASSIGNMENT, IF, THEN, ELSE, WRITELN, WHILE, DO, DOT, PBEGIN, END, COLON, SEMICOLON, COMMA
- SLASH, STAR, MINUS, PLUS, DIV, MOD
- EQUAL, GE, GT, LE, LT, NOTEQUAL, AND, OR, LPAREN, RPAREN
- DTINTEGER, DTBOOLEAN, DTCHAR, DTSTRING, DTREAL, TRUE, FALSE

Durante a análise também são detectados comentários do tipo (* … *) que indicam ao compilador que a secção entre os delimitadores deve ser ignorada. São ainda apanhadas *strings*, que são definidas como qualquer texto entre pelicas: '…' e limpas de imediato para que possam ser usadas ao longo de todo o compilador. De referir ainda que toda a análise é *case insensitive*.

ANÁLISE SINTÁCTICA

Esta fase está dividida em dois grandes passos. Inicialmente foi definida a gramática da linguagem através do YACC. Este trata a informação enviada pelo LEX e tenta encontrar expressões no código fonte que combinem com as definas pelo programador para a linguagem. A isto chama-se análise sintáctica e caso seja bem sucedida garante que o programa fonte não tem erros de sintaxe, estando então pronto para inicializar a análise semântica. Resumindo, esta análise permite verificar se o programa está bem estruturado, não garantido de modo algum que está correcto.

O segundo passo é a criação da árvore de sintaxe abstracta. Esta árvore será utilizada nos passos seguintes e irá servir de código intermédio para a geração do código final, neste caso em C reduzido. Para a criação da árvore são utilizadas estruturas em C que permitem guardar toda a informação retirada do código fonte sem qualquer ambiguidade.

Durante esta fase são geradas mensagens de erro genéricas que apenas indicam a existência de um erro de sintaxe numa dada linha.

A **especificação da gramática** e a **especificação da sintaxe abstracta** são fornecidas em anexo.

ANÁLISE SEMÂNTICA

Este passo irá analisar a árvore de código intermédio de modo a detectar possíveis falhas semânticas, por exemplo, atribuição de valores do tipo X a variáveis do tipo Y. em anexo está uma listagem dos erros que o compilador detecta durante este passo. Para efectuar as referidas funções também cria a tabela de símbolos e a tabela de funções.

O funcionamento da tabela de símbolos e funções está interligado e o *scope* das variáveis locais é conseguido do seguinte modo: todas as variáveis (globais ou locais) são guardadas na mesma tabela de símbolos, mas é-lhes anexado um valor que permite determinar a sua origem (global, função X, função Y, etc). É com base este valor que a análise semântica determina se é possível usar determinada variável dentro de uma função específica. Não há recurso a ambientes de função mas é possível recriar na totalidade quais as varáveis internas de uma função e quais as globais, isto é, o seu ambiente. Esta é uma abordagem semelhante à usada nas fichas mas difere na implementação.

Nesta fase a descrição de erros é exaustiva dando o máximo de informação possível ao programador. Além dos erros também são gerados avisos que não impedem a geração final de código mas que podem, por sua vez, gerar avisos quando o código C final é corrido. Por exemplo, transformações entre inteiros e reais que originem perca de precisão.

De um modo resumido são mostrados erros ou avisos nas seguintes situações:

- Declarar duas ou mais variáveis com o mesmo nome;
- Usar variáveis não declaradas;
- Usar variáveis fora do *scope* em que foram definidas;
- Utilizar operadores sobre variáveis com tipos incompatíveis;
- Declarar duas ou mais funções com o mesmo nome;
- Utilizar operados sobre funções e varáveis com tipos incompatíveis;
- Chamadas incorrectas às funções, tanto em tipo de argumentos como em número.

Tal como foi referido, em anexo são descritos todos os casos em que a análise semântica detecta algum erro, são também fornecidos pequenos exemplos.

TRADUÇÃO DE CÓDIGO

Nesta fase já é garantido que o código fornecido ao compilador está correcto, tanto a nível de estrutura como semântico. Sendo um programa válido é possível criar o *output* em C reduzido. Este passo trata da passagem do código intermédio (árvore de sintaxe abstracta), utilizando a tabela de símbolos e a tabela de funções para o código final.

Para a criar o *scope* das funções foram utilizadas *frames* tal como apresentadas nas fichas dadas nas aulas práticas. Estas permitem que uma função chame outra dentro do seu corpo de execução, sem que se perca contexto. Por exemplo: pontos de retorno no código são guardados, pois toda a informação é salva na pilha de *frames*, que pode crescer independente da profundidade das chamadas.

Estrutura de uma *frame*:

```
typedef struct _f1{
	struct _f1* parent; //Frame pointer
	void* locals[64];   //Espaço de endereçamento para variáveis locais
	int return_address; //Endereço do código na função chamante
} frame;
```

Não são geradas quaisquer optimizações ao código sendo traduzido de um modo directo. Em anexo segue um exemplo do código gerado de modo a ser mais simples analisar o resultado final.

CONCLUSÃO

Pode-se concluir que seguir uma metodologia já estudada para o desenvolvimento de um compilador trás claras vantagens a nível de simplificação das tarefas necessárias. Tal procedimento, e em conjunto com ferramentas já bem estabelecidas, como é o caso do LEX e o YACC, permitem que grupos de tamanho reduzido (uma ou duas pessoas) consigam criar um compilador de boa qualidade, mesmo que para um reduzido *set* da linguagem.

Ainda de referir que é de extrema importância que a analise semântica seja feita com cuidado extra pois trata-se duma secção do compilador que facilmente cresce em complexidade e onde mais tempo se gasta para que a validação de código seja perfeita.

BIBLIOGRAFIA

"Processadores de Linguagens – da concepção à implementação", Rui Gustavo Crespo, 2ª Edição, 2001, IST Press

ANEXOS

ESPECIFICAÇÃO DA GRAMÁTICA

```
program :
 program_heading SEMICOLON program_block DOT;

program_heading :
 PROGRAM var_name;

program_block :
   VAR variable_declaration_part function_list PBEGIN statement_list END
 | VAR variable_declaration_part PBEGIN statement_list END
 | function_list PBEGIN statement_list END
 | PBEGIN statement_list END;

variable_declaration_part :
   variable_declaration
 | variable_declaration_part variable_declaration;

variable_declaration :
   var_name_list COLON DTINTEGER SEMICOLON
 | var_name_list COLON DTBOOLEAN SEMICOLON
 | var_name_list COLON DTREAL SEMICOLON;

var_name_list :
   var_name_list COMMA var_name
 | var_name;

function_list:
   function_list function
 | function;

function:
 FUNCTION var_name LPAREN function_argument_list RPAREN COLON function_return SEMICOLON VAR
variable_declaration_part PBEGIN statement_list END SEMICOLON
 | FUNCTION var_name LPAREN function_argument_list RPAREN COLON function_return SEMICOLON PBEGIN
statement_list END SEMICOLON;

function_argument_list:
   function_argument_list SEMICOLON function_argument
 | function_argument
 | ;


function_argument:
 var_name_list COLON function_return;
```

```
function_return:
   DTINTEGER
 | DTREAL
 | DTBOOLEAN;

statement_list :
   statement_list statement
 | statement;

statement :
   while_statement
 | if_statement
 | assignment_statement
 | writeln_statement;

while_statement :
 WHILE logical_expression DO PBEGIN statement_list END SEMICOLON;

if_statement :
   IF logical_expression THEN statement
 | IF logical_expression THEN PBEGIN statement_list END SEMICOLON
 | IF logical_expression THEN statement ELSE statement
 | IF logical_expression THEN statement ELSE PBEGIN statement_list END SEMICOLON
 | IF logical_expression THEN PBEGIN statement_list END SEMICOLON ELSE statement
 | IF logical_expression THEN PBEGIN statement_list END SEMICOLON ELSE PBEGIN statement_list END
SEMICOLON;

assignment_statement :
 var_name ASSIGNMENT expression SEMICOLON;

function_call:
 var_name LPAREN function_call_argument_list RPAREN;

function_call_argument_list:
   function_call_argument_list COMMA expression
 | expression
 |;

writeln_statement :
 WRITELN LPAREN writeln_argument_list RPAREN SEMICOLON;

writeln_argument_list :
   writeln_argument_list COMMA writeln_argument
 | writeln_argument;

writeln_argument :
   string
 | expression;
```

```
logical_expression :
   logical_expression AND logical_expression
 | logical_expression OR logical_expression
 | LPAREN logical_expression RPAREN
 | internal_logical_expression;

internal_logical_expression :
   expression EQUAL expression
 | expression GT expression
 | expression LT expression
 | expression GE expression
 | expression LE expression
 | expression NOTEQUAL expression
 | expression;

expression :
   expression PLUS expression
 | expression MINUS expression
 | expression DIV expression
 | expression STAR expression
 | expression MOD expression
 | LPAREN expression RPAREN
 | digit_sequence
 | var_name
 | function_call
 | boolean_expression;

boolean_expression :
   TRUE
 | FALSE;

var_name :
 IDENTIFIER;

digit_sequence :
   MINUS DIGSEQ
 | DIGSEQ
 | MINUS FDIGSEQ
 | FDIGSEQ;

string :
 STRING;
```

ESPECIFICAÇÃO DA SINTAXE ABSTRACTA

- **is_program** -> (<program_heading: is_program_heading> <program_block: is_program_block>)
- **is_program_heading** -> (<var_name : is_var_name>)
- **is_program_block** -> (<variable_declaration_list : is_variable_declaration_list><function_list : is_function_list><statement_list : is_statement_list>)

- **is_variable_declaration_list**
- **is_variable_declaration** -> (<var_name_list : is_var_name_list> <data_type : data_type>)

- **is_function_list**
- **is_function** -> (<function_name : is_function_name> <function_argument_list : is_function_argument_list> <data_type : data_type> <variable_declaration_part : is_variable_declaration_part>)

- **is_function_argument_list**
- **is_function_argument** -> (<var_name_list : is_var_name_list> <data_type : data_type>)

- **is_statement_list**
- **is_statement** -> is_assignment_statement v is_writeln_statement v is_if_statement v is_while_statement

- **is_assignment_statement** -> (<var_name : is_var_name> <expression : is_expression>)
- **is_writeln_statement** -> (<writeln_argument_list : is_writeln_argument_list>)
- **is_while_statement** -> (<logical_expression : is_logical_expression> <statement_list : is_statement_list>)
- **is_if_statement** -> (<then_block : statement_list v statement> <else_block : statement_list v statement> <have_block_else : boolean>)

- **is_writeln_argument_list**
- **is_writeln_argument** -> is_expression v is_string

- **is_function_call_argument_list**
- **is_function_call** -> (<var_name : is_var_name> <function_call_argument_list : is_function_call_argument_list>)

- **is_logical_expression** -> is_internal_logical_expression v is_infix_logical_expression
- **is_internal_logical_expression** -> (<expression1 : is_expression> <expression2 : is_expression> <logical_operator : logical_operator>)
- **is_infix_logical_expression** -> (<logical_expression1 : is_logical_expression> <logical_expression2 : is_logical_expression> <logical_operator : logical_operator>)
- **is_infix_expression** -> (<expression1 : is_expression> <expression2 : is_expression> <oper : operator>)
- **is_expression** -> is_var_name v is_digit_sequence v is_string v is_infix_expression v is_function_call

- **is_var_name_list**
- **is_var_name** -> (<var_name : is_var_name>)

- **is_string** -> (<string : char*>)
- **is_digit_sequence** -> is_digit_sequence v is_float_digit_sequence
- **is_boolean** -> (<boolean : unsigned>)

ERROS E WARNINGS

Os exemplos listados exemplificam o tipo de erros que a análise semântica detecta.

Instrução Assignement

```
var i, i…   // Declarar mais que uma variável com o mesmo nome gera um erro
```

```
i := r;     // Utilizar variáveis não declaradas gera um Erro
```

```
i := 1.5;   // Guardar um número Real num Integer gera um Warning
```

```
i := r;     // Guardar um Real num Integer gera um Warning (precisão)
```

```
b := i;     // Guardar um Integer num Boolean gera um Erro
```

```
b := r;     // Guardar um Real num Boolean gera um Erro
```

```
b := 1;     // Guardar um número Inteiro num Boolean gera um Erro
```

```
b := 1.5;   // Guardar um número Real num Boolean gera um Erro
```

```
i := b;     // Guardar um Boolean num Integer gera um Erro
```

```
r := b;    // Guardar um Real num Integer gera um Erro
```

```
i := i + b;    // Os erros de tipos são detectados em qualquer posição
```

```
r := 2 + i;    // Variáveis não declaradas são detectadas em qualquer posição
```

```
i := 1 + 0.2;    // Os Warnings de precisão são detectados em qualquer posição
```

Instrução Writeln

```
writeln(b + i); // Utilizar variáveis não declaradas gera um Erro
```

```
writeln(b + i);    // As expressões internas assumem a primeira variável como sendo o tipo da expressão para gerar erros
```

```
writeln(b + 0.2);    // As expressões internas convertem números para o seu tipo: Integer ou Real para comparar com outros símbolos
```

```
writeln('s' + 1);    // Misturar strings com outros tipos gera um erro de sintaxe
```

Instruções IF e While

```
if b < i then… // Utilizar variáveis não declaradas gera um Erro
```

`if (b + 0.2) < i then…`	`// Todas as expressões são testadas contra mistura de tipos`

`if b < (i + 1) then…`	`// Um erro é gerado caso as expressões de ambos os lados da expressão lógica são seja do mesmo tipo, assume-se que o primeiro símbolo numa expressão é o tipo correcto da expressão`

Chamada de Funções

`sum(i), sum(i, n)`	`// Declarar mais que uma função com o mesmo nome gera um erro, mesmo que tenham assinaturas diferentes`
`i := sum(i1, i2);`	`// O tipo de retorno da função tem de ser igual ao da variável que o vai guardar`
`i := sum(i1, i2);`	`// O número de argumentos que a função recebe tem de estar correcto ou um erro é gerado`
`i := sum(i, b, r);`	`// Os argumentos da função têm de ter o tipo correcto ou um erro é gerado`
`b := isodd(i + 1);`	`// As expressões utilizadas como argumentos são analisadas contra variáveis não declaradas e mistura de tipos`
`sum() { …i… }`	`// As variáveis nas funções são locais e utilizar variáveis do scope global gera um erro (variável não declarada)`
`sum() { sum := …}`	`// Caso uma função não tenha a atribuição de return é gerado um erro`

EXEMPLOS

Nas próximas paginas são apresentados alguns exemplos de código válido e inválido, utlizando comentários para descrever o comportamento esperado.

Estes exemplos seguem no CD, juntamente com o código fonte do compilador.

```
(* (example.pas) Este programa demostra de um modo geral todo o set da lingaguem aceite pelo compilador. *)

program Example;

var
 i1, i2: integer;
 r1, r2: real;
 b1, b2: boolean;


function test(a, b: integer) : integer;
var c, d : integer;
begin
    c := 10;
    d := 10;
    test := a + b + c + d;
end;


function rtest(i : integer; r : real) : real;
begin
    rtest := r + i;
end;


function btest(b : boolean; j : integer; k : real) : boolean;
var
    i : integer;
    r : real;
begin

    i := test(j, j) + test(j, j);
    r := rtest(i, k);
    writeln(i, ' ', r);

    if b then
        btest := false;
    else
        btest := true;
end;

begin

    i1 := 10 + 20 div 2;
    i2 := i1 * i1;
    writeln(i1, ' ** 2 = ', i2);

    r1 := 3.1415;
    r2 := r1 * 2;
    writeln(r2, ' = ', r1 * 2.0);

    b1 := true;
    b2 := false;
    b2 := b2;
    writeln(b1, ' <> ', b2);


    if i2 > i1 then
        writeln('i2 > i1');
    else
        writeln('i2 < i1');


    if r2 > r1 then
    begin
        writeln('r2 > r1');
    end;
    else
    begin
        writeln('r2 < r1');
    end;


    i1 := 0;

    while i1 < 3 do
```

```
    begin
        writeln('i: ', i1);
        i1 := i1 + 1;
    end;


    i1 := 10;
    i2 := i1;
    i1 := test(i1, i2);
    writeln(i1);

    i1 := 10;
    r1 := 10.5;
    r1 := rtest(i1, r1);
    writeln(r1);

    b1 := true;
    b2 := btest(b1, 10, 2.0);
    writeln(b2);

end.
```

```pascal
(* (power.pas) Este programa é um exemplo de recursividade utilizada para calcular potências. *)

program Power;

var a, b, n: integer;

function power(a, b : integer) : integer;
begin
    if b = 1 then
        power := a;
    else
        power := a * power(a, b - 1);
end;

begin

    a := 3;
    b := 6;
    n := power(a, b);
    writeln(a, '^', b, ' = ', n);

end.
```

```
(* (primes.pas) Este programa calcula número primos usando para tal um algortimo que calcula raizes inteiras de um dado número. *)

program Primes;

var
    i, j, h, k, max: integer;
    isprime: boolean;


function isqrt (n : integer) : integer; (* integer square root *)
var
    k, xa, xo: integer;
    run : boolean;
begin
    xa := 0;
    xo := n;
    run := true;

    while run do
    begin
        xa := (xo + (n div xo)) div 2;

        if ((xa >= xo) and ((xa - xo) <= 1)) or ((xa < xo) and ((xo - xa) <= 1)) then
            run := false;

        xo := xa;
    end;

    isqrt := xa;
end;

begin

    i := 1;
    max := 20;

    while i <= max do begin
        isprime := true;
        j := 2;
        k := isqrt(i);

        while j <= k do
        begin
            if (i div j) * j = i then
                isprime := false;

            j := j + 1;
        end;

        if isprime then
            writeln(i, ' is prime! (1..', k, ')');

        i := i + 1;
        if i > 3 then (* jump even numbers *)
            i := i + 1;
    end;

end.
```

```
(* (fibonacci.pas) Este programa é um exemplo de recursividade utilizada para calcular a sequência de Fibonacci. *)

program Fibonacci;

var n, f, max: integer;

function fibonacci(n : integer) : integer;
begin
    if n = 0 or n = 1 then
        begin
        fibonacci := n;
        end;
    else
        begin
        fibonacci := fibonacci(n - 1);
        fibonacci := fibonacci + fibonacci(n - 2);
        end;
end;

begin
    max := 20;

    n := 1;
    while n <= max do
    begin
        f := fibonacci(n);
        writeln('fibonacci(', n,') = ', f);
        n := n + 1;
    end;

end.
```

```
(* (errors.pas) Este programa gera todos os erros e avisos detecados pelo compilador (listados nas próximas páginas). *)

program Errors;

var
 i, ii, i: integer; (* Error *)
 r, rr: real;
 b, bb: boolean;

function test(): integer; (* Error *)
var z : integer;
begin
    i := 10; (* Error *)
end;

function itest(a : integer; b : real; c : boolean): integer ;
begin
    itest := 1;
end;

function rtest(a : integer; b : real; c : boolean): real ;
begin
    rtest := 1.5;
end;

function btest(a : integer; b : real; c : boolean): boolean ;
begin
    btest := true;
end;

begin

    z := 0; (* Error *)

    i := 1;
    i := 1.5; (* Warning *)
    i := true; (* Error *)
    i := i;
    i := r; (* Warning *)
    i := b; (* Error *)
    i := z; (* Error *)

    r := 1;
    r := 1.5;
    r := true; (* Error *)
    r := i;
    r := r;
    r := b; (* Error *)

    b := 1; (* Error *)
    b := 1.5; (* Error *)
    b := true;
    b := i; (* Error *)
    b := r; (* Error *)
    b := b;

    i := 1 + 1.5; (* Warning *)
    i := 1 + 1.5 + true; (* Warning, Error *)
    i := i + r + b + z; (* Warning, Error, Error *)

    r := 1 + 1.5;
    r := 1 + 1.5 + true; (* Error *)
    r := i + r + b; (* Error *)

    b := 1 + 1.5; (* Error, Error *)
    b := 1 + 1.5 + true; (* Error, Error *)
    b := i + r + b; (* Error, Error *)

    writeln(i + 1);
    writeln(i + 1.5); (* Warning *)
    writeln(i + true); (* Error *)
    writeln(i + i);
    writeln(i + r); (* Warning *)
    writeln(i + b); (* Error *)
    writeln(i + z); (* Error *)
    writeln(z + i); (* Error *)

    writeln(r + 1);
    writeln(r + 1.5);
    writeln(r + true); (* Error *)
    writeln(r + i);
    writeln(r + r);
    writeln(r + b); (* Error *)
```

```
    writeln(r + z); (* Error *)

    writeln(b + 1); (* Error *)
    writeln(b + 1.5); (* Error *)
    writeln(b + true);
    writeln(b + i); (* Error *)
    writeln(b + r); (* Error *)
    writeln(b + b);
    writeln(b + z); (* Error *)

    if i < 1 then writeln('t');
    if i < 1.5 then writeln('t');
    if i < true then writeln('t'); (* Error *)
    if i < i then writeln('t');
    if i < r then writeln('t');
    if i < b then writeln('t'); (* Error *)
    if i < z then writeln('t'); (* Error *)

    if r < 1 then writeln('t');
    if r < 1.5 then writeln('t');
    if r < true then writeln('t'); (* Error *)
    if r < i then writeln('t');
    if r < r then writeln('t');
    if r < b then writeln('t'); (* Error *)
    if r < z then writeln('t'); (* Error *)

    if b < 1 then writeln('t'); (* Error *)
    if b < 1.5 then writeln('t'); (* Error *)
    if b < true then writeln('t');
    if b < i then writeln('t'); (* Error *)
    if b < r then writeln('t'); (* Error *)
    if b < b then writeln('t');
    if b < z then writeln('t'); (* Error *)

    i := itest(1, 1.5, true);
    i := rtest(1, 1.5, true); (* Warning, Warning *)
    i := btest(1, 1.5, true); (* Error *)

    r := itest(1, 1.5, true);
    r := rtest(1, 1.5, true);
    r := btest(1, 1.5, true); (* Error *)

    b := itest(1, 1.5, true); (* Error *)
    b := rtest(1, 1.5, true); (* Error *)
    b := btest(1, 1.5, true);

    i := itest(1.5, true); (* Error, Error *)
    r := rtest(1, true); (* Error, Error *)
    b := btest(1, 1.5); (* Error *)

end.
```

Lista de erros e avisos gerados pelo program anterior (errors.pas).

```
Syntax...: 0 errors
ERROR at line 4: symbol 'i' already defined!
ERROR at line 11: symbol 'i' not declared!
ERROR at line 12: function 'test()' doesn't have a return statement (test := <value>)!
ERROR at line 31: symbol 'z' not declared!
WARNING at line 34: symbol '1.500000' is not Integer! (Precision loss)
ERROR at line 35: symbol 'true' is not Integer!
WARNING at line 37: symbol 'r' is not Integer! (Precision loss)
ERROR at line 38: symbol 'b' is being assign to Integer but is Boolean!
ERROR at line 39: symbol 'z' not declared!
ERROR at line 43: symbol 'true' is not Real!
ERROR at line 46: symbol 'b' is being assign to Real but is Boolean!
ERROR at line 48: can't assign Integer to Boolean!
ERROR at line 49: can't assign Real to Boolean!
ERROR at line 51: symbol 'i' is being assign to Boolean but is Integer!
ERROR at line 52: symbol 'r' is being assign to Boolean but is Real!
WARNING at line 55: symbol '1.500000' is not Integer! (Precision loss)
WARNING at line 56: symbol '1.500000' is not Integer! (Precision loss)
ERROR at line 56: symbol 'true' is not Integer!
WARNING at line 57: symbol 'r' is not Integer! (Precision loss)
ERROR at line 57: symbol 'b' is being assign to Integer but is Boolean!
ERROR at line 57: symbol 'z' not declared!
ERROR at line 60: symbol 'true' is not Real!
ERROR at line 61: symbol 'b' is being assign to Real but is Boolean!
ERROR at line 63: can't assign Integer to Boolean!
ERROR at line 63: can't assign Real to Boolean!
ERROR at line 64: can't assign Integer to Boolean!
ERROR at line 64: can't assign Real to Boolean!
ERROR at line 65: symbol 'i' is being assign to Boolean but is Integer!
ERROR at line 65: symbol 'r' is being assign to Boolean but is Real!
WARNING at line 68: mixing data types, '1.500000' (Real) with Integer at expression level!
ERROR at line 69: mixing data types, Boolean with Integer at expression level!
WARNING at line 71: mixing data types, 'r' Real with Integer at expression level!
ERROR at line 72: mixing data types, 'b' Boolean with Integer at expression level!
ERROR at line 73: symbol 'z' not declared!
ERROR at line 74: symbol 'z' not declared!
ERROR at line 78: mixing data types, Boolean with Real at expression level!
ERROR at line 81: mixing data types, 'b' Boolean with Real at expression level!
ERROR at line 82: symbol 'z' not declared!
ERROR at line 84: mixing data types, Boolean with Integer at expression level!
ERROR at line 85: mixing data types, Boolean with Real at expression level!
ERROR at line 87: mixing data types, 'i' Integer with Boolean at expression level!
ERROR at line 88: mixing data types, 'r' Real with Boolean at expression level!
ERROR at line 90: symbol 'z' not declared!
ERROR at line 94: mixing data types, Integer with Boolean at logical level!
ERROR at line 97: mixing data types, Integer with Boolean at logical level!
ERROR at line 98: symbol 'z' not declared!
ERROR at line 102: mixing data types, Real with Boolean at logical level!
ERROR at line 105: mixing data types, Real with Boolean at logical level!
ERROR at line 106: symbol 'z' not declared!
ERROR at line 108: mixing data types, Boolean with Integer at logical level!
ERROR at line 109: mixing data types, Boolean with Real at logical level!
ERROR at line 111: mixing data types, Boolean with Integer at logical level!
ERROR at line 112: mixing data types, Boolean with Real at logical level!
ERROR at line 114: symbol 'z' not declared!
WARNING at line 117: symbol 'rtest' is not Integer! (Precision loss)
WARNING at line 117: mixing data types, 'rtest()' return Real but is being assign to Integer!
ERROR at line 118: function 'btest()' is being assign to Integer but is Boolean!
ERROR at line 122: function 'btest()' is being assign to Real but is Boolean!
ERROR at line 124: function 'itest()' is being assign to Boolean but is Integer!
ERROR at line 125: function 'rtest()' is being assign to Boolean but is Real!
ERROR at line 128: argument #1 of 'itest()' must be Integer but is Real!
ERROR at line 128: wrong number of arguments, 'itest()' needs 3, 2 provided!
ERROR at line 129: argument #2 of 'rtest()' must be Real but is Boolean!
ERROR at line 129: wrong number of arguments, 'rtest()' needs 3, 2 provided!
ERROR at line 130: wrong number of arguments, 'btest()' needs 3, 2 provided!
Semantic.: 56 errors
```

```c
/* Código final gerado pelo compilador para o programa prime.pas */

#include <stdlib.h>
#include <stdio.h>

typedef struct _f1 {
        struct _f1* parent;
        void* locals[64];
        void* outgoing[32];
        int return_address;
}frame;

typedef enum {
        FALSE, TRUE
} boolean;

int main() {
        frame* fp = NULL;
        frame* sp = NULL;
        int r = 0; /* registo para redirect no final das funções */
        sp = (frame*) malloc(sizeof(frame));

        /* Program name: Primes */

        /* Global Variables */

        int v0; /* i */
        int v1; /* j */
        int v2; /* h */
        int v3; /* k */
        int v4; /* max */
        boolean v5; /* isprime */

        /* Functions */

        /* isqrt() ************************************************** */
        int v7; /* n */
        goto skip_isqrt;
        start_isqrt:
        fp = sp;
        sp = (frame*) malloc(sizeof(frame));
        sp->parent = fp;
        sp->return_address = r;

        int v6; /* isqrt (return value) */
        sp->locals[6] = (int *) malloc(sizeof(int));
        sp->locals[7] = (int *) malloc(sizeof(int)); /* argument: n */
        sp->locals[7] = (int *) v7;

        sp->locals[8] = (int *) malloc(sizeof(int)); /* var: k */
        sp->locals[9] = (int *) malloc(sizeof(int)); /* var: xa */
        sp->locals[10] = (int *) malloc(sizeof(int)); /* var: xo */
        sp->locals[11] = (boolean *) malloc(sizeof(boolean)); /* var: run */

        sp->locals[9] = (int *) 0; /* Assignment */
        sp->locals[10] = (int *) ((int) sp->locals[7]); /* Assignment */
        sp->locals[11] = (boolean *) TRUE; /* Assignment */

        /* WHILE (#7) */
        goto label_while_eval_7;
        label_while_body_7:
        sp->locals[9] = (int *) ((((int) sp->locals[10])+(((int) sp->locals[7])/((int) sp->locals[10])))/2); /* Assignment */

        /* IF (#5) ------------------------------ */
        goto label_if_eval_5;
        label_if_body_5:
        sp->locals[11] = (boolean *) FALSE; /* Assignment */
        goto label_if_end_5;
        label_if_eval_5:
        if ((((((int) sp->locals[9]) >= ((int) sp->locals[10])) && ((((int) sp->locals[9])-((int) sp->locals[10])) <= 1)) || ((((int) sp
        label_if_end_5:
        sp->locals[10] = (int *) ((int) sp->locals[9]); /* Assignment */
        label_while_eval_7:
        if ((((boolean) sp->locals[11]))) goto label_while_body_7;
        label_while_end_7:
        sp->locals[6] = (int *) ((int) sp->locals[9]); /* Assignment */

        v6 = ((int) sp->locals[6]);


        r = sp->return_address;
        sp = sp->parent;
        fp = sp->parent;
        goto redirect;
        skip_isqrt:; /* NOOP is needed because a label can't point to a var dec (eg: int v1;) */

        /* Statements */

        v0 = 1; /* Assignment */
        v4 = 20; /* Assignment */

        /* WHILE (#22) */
        goto label_while_eval_22;
        label_while_body_22:
        v5 = TRUE; /* Assignment */
        v1 = 2; /* Assignment */

        /* calling 'isqrt()' to 'v6' (Call ID: 0) */
        v7 = v0;
        r = 0;
```

```
        goto start_isqrt;
        assignment_0:;
        int c0 = v6;
        v3 = c0; /* Assignment */

        /* WHILE (#17) */
        goto label_while_eval_17;
        label_while_body_17:

        /* IF (#15) ------------------------------ */
        goto label_if_eval_15;
        label_if_body_15:
        v5 = FALSE; /* Assignment */
        goto label_if_end_15;
        label_if_eval_15:
        if ((((v0/v1)*v1) == v0)) goto label_if_body_15;
        label_if_end_15:
        v1 = (v1+1); /* Assignment */
        label_while_eval_17:
        if ((v1 <= v3)) goto label_while_body_17;
        label_while_end_17:

        /* IF (#18) ------------------------------ */
        goto label_if_eval_18;
        label_if_body_18:

        printf("%d", v0);
        printf(" is prime! (1..");
        printf("%d", v3);
        printf(")");
        printf("\n");

        goto label_if_end_18;
        label_if_eval_18:
        if ((v5)) goto label_if_body_18;
        label_if_end_18:
        v0 = (v0+1); /* Assignment */

        /* IF (#21) ------------------------------ */
        goto label_if_eval_21;
        label_if_body_21:
        v0 = (v0+1); /* Assignment */
        goto label_if_end_21;
        label_if_eval_21:
        if ((v0 > 3)) goto label_if_body_21;
        label_if_end_21:
        label_while_eval_22:
        if ((v0 <= v4)) goto label_while_body_22;
        label_while_end_22:

        /* Redirect functions on return */
        goto skip_redirect;
        redirect:
        if (r == 0) goto assignment_0;
        skip_redirect:

        return 0;
}
```