**COP3402 PROJECT Rascal Compiler Spring 2011**

**Due: Tuesday, April 26 at 11:59PM (no lateness allowed)**

The major project for this course is the creation of a language translator for a variant of Pascal that we shall call Rascal. The variant is defined, to a large extent, by the grammar that is displayed later in this document.

**THE SOURCE LANGUAGE:**

Before discussing the intermediate code, I need to describe some features of Rascal you will be translating. Some of this is evident from the grammar. Some is not.

1. The program statement for this language does not have a parameter list. We will not in fact support any input/output statements.
2. The notion of units is essential to the data objects of this language. All data is of base type integer, but different objects can have different units. E.g., an object may have "feet" or "hours" as its units. Some objects have no units. These are members of the super class "NUMBER". Units can have aliases. Thus, we can alias "foot" and "feet" in order to talk about 1 foot and 2 feet. We will limit distinct units to 16 different kinds. The units of an object can be accessed using the DOT operator.   
     
   Thus, if we declared DISTANCE by

unit MILE alias (MILES), FOOT alias (FEET);

var DISTANCE : MILES;

DISTANCE.unit returns "MILE",

We can recast a term by preceding it with units, enclosed in {}. So, for example, we can change miles to feet by the following:

unit MILE alias (MILES), FOOT alias (FEET);

const ft\_per\_mile = 5280 ;

var DISTANCE : MILES; D2 : FEET; X: NUMBER;

…

DISTANCE := 14; (\* NUMBER is automatically cast \*)

D2 := {feet} DISTANCE \* ft\_per\_mile;

X := D2\*5; (\* NUMBER is cast when combined with other units and forces a cast on assign \*)

1. Constants and variables have associated units, and objects with differing units may never be added, subtracted, multiplied or divided except that a unitless object (NUMBER) may be combined with anything, the resulting units being unchanged. Comparisons (tests in if and while) must be between expressions of the same units, except again numbers are automatically cast to match their mates.  
     
   We also allow variables to be simple lists (vectors).

ARRAY [ lo .. hi ] OF units;

The values of lo and hi are limited to unit-less integers. Of course lo<=hi.

The base unit NUMBER is a reserved word and represents a type that combines with all other objects. NUMBERS aliases NUMBER, and this reserved word is like an identifier, in that it is case insensitive. In fact, everything in Rascal is case insensitive, including keywords. Moreover, identifiers and units are only distinguishable based on the first up to 8 characters (underscores excluded in this count).

Any expression can be changed to unitless by casting as follows:  
  
{number} expression

1. Semantic error detection of unit mismatches requires you to compute the units for each subexpression within an expression. Errors in type mismatch must be reported. I would recommend just one such error message for an expression, but that is up to you.
2. Functions are not allowed.
3. Procedures are similar to Pascal, except that they cannot be statically nested. Thus, this version has a flat structure like found in C.
4. Arguments to procedures are simple variables. They may not include procedure names and/or constants and/or expressions with operators. Formal parameters are always by value, so results can only be returned by altering global variables.
5. Upper and lower case letters are indistinguishable in identifiers and keywords. Underscores may appear inside identifiers, but only serve to improve readability. However, they should not be embedded in keywords.

**EXPANDED OPCODES**

**Constant term**

CON op1 0 // op1 is a constant value (positive or zero)

op used for following code is offset in main memory, offset from stack mark or a –triple

Offsets for main memory should be preceded by ‘M’; stack should be preceded by ‘S’. In your recursive descent just return index to symbol table and emit should take care of rest using storetype (see symbol table description.

**Assignment operators**

:= op1 op2 // assign op2 to op1

[]= op1 op2 // assigns the value from op3 to vector component op1[op2]

\_ op3 0

**Accessor for subscripted variable**

=[] op1 op2 // evaluates to op1[op2] (used in rhs references)

**Arithmetic operations**

+ op1 op2 // operands can be to symbol table (positive)

- op1 op2 // operands can be references to other triples (negative)

\* op1 op2 // no operand can be zero

/ op1 op2

% op1 op2

++ op1 0

-- op1 0

**Unconditional jump uses first field as destination triple**

JUMP -loc 0 // jumps to triple at loc

**Conditional jumps use first field to reference triple where test value was computed**

**Second field is destination triple**

JZ -loc1 -loc2 // conditional jump on zero (=0)

JNZ -loc1 -loc2 // conditional jump on not zero (<>0)

JP -loc1 -loc2 // conditional jump on positive (>0)

JNP -loc1 -loc2 // conditional jump on not positive (<=0)

JM -loc1 -loc2 // conditional jump on minus (<0)

JNM -loc1 -loc2 // conditional jump on not minus (>=0)

**Calls and returns (assumes a stack for procedure calls)**

ARG entry 0 // pushes value associated with argument variable on stack

CALL #args #sz // adds #sz (size of local storage) to stack top;

\_ -loc1 0 // pushes return and old mark on stack; sets stack mark

// to stack location just above first ARG; calls code at loc1

RET 0 0 // pops return address; cleans up stack to current mark and returns

// also works to exit main

**Miscellaneous**

ERROR 0 0 // error (can extend to put error codes in triple)

NOP 0 0 // no operation (used when instruction deleted

// might operands to save useful information)

**LIMITATIONS**

I have limited formal parameters to be an untyped list of variable names. That’s acceptable in that all units can be automatically recast to NUMBER. It does limit us some, but it makes your life easier in that you do not need to retain parameter units and you do not need to do semantic checks for typing mismatches on procedure calls. You, in fact, only need to know the number of arguments required and be sure there is a match; associations of names to stack offsets will still need to be done when you compile the routine’s code. Realize that procedures must use globals to return results since I removed the requirement to handle “by address” arguments.

There are no logical operators. That makes life a bit easier. In fact, all tests are of the form expression RELOP expression, where the expressions are numeric.

The only new control structure not addressed in Assignment#4 is the FOR. I got rid of CASE.

**SYMBOL TABLE STRUCTURE**

Identifier Unit StoreType Offset Size Low High

Unit is important for type checking. It is index into separate unit symbol table.

StoreType for us is main memory (global) or stack. Use code of 1 for main and 2 for stack. Note: real language also uses heap.

Offset for main memory is from some base address. We assume 1, 2, etc. Of course arrays affect this.

Offset for stack is relative to latest mark. 1 is first arg, …, k is k-th arg; k+1 is first local, etc.

Size is number of elements in area allocated for variable (0 for scalar, m for array of length m = High - Low).

High and Low are available for vectors and not used for lists.

**Questions:**

How do we differentiate constants from variables?

How do we adapt this to handle procedures?

Do we need a separate type for formal arguments or can we just show them as stack variables?

Is it better to have a separate table for units, or should we adapt the symbol table for them?

**OPTIMIZATIONS**

**Easy:**

Identities: x+0 = 0+x = x\*1 = 1\*x = x

Increment/Decrement; x+1 = 1+x = ++value(x); x-1 = --value(x); works even if x is a triple reference

Note: ++ and -- are not part of language, just of intermediate code and they do not alter any variable

Strength reduction: x\*2 = 2\*x = x+x

Constant folding: Examples: 2+7 = 9; 2 \* 7 = 14;

Useless: a := a can be replaced by NOP but be careful about references to this triple by other triples

**Harder:**

Common subexpressions: Be careful – must have no intervening changes to operands unless CON entries

Code compression (getting rid of NOPs and adjusting all references correctly)

Goto chasing: jumps to jumps (e.g., unconditional jump to conditional); jumps to returns; jumps to next triple

Other ones that you think about

**WHAT YOU MUST TURN IN**

1. A C or C++ program that includes a lexical analyzer, recursive descent parser, intermediate code generator (triples are required) and some code optimizations (at least two simple optimizations; challenging optimizations can results in extra credit or can offset other less well done parts of the assignment) for the Rascal language. This program must report syntactic and unit-based semantic errors. It must also recover gracefully from errors. As in Assign#4, code is shown incrementally and at end. Symbol table must be dumped at start of each block of code for a procedure or the main (after local variables are inserted for procedure block and at start of main block where symbol table has just globals which include all procedure prototypes).
2. A log book with at least three entries per week from now until the due date (April 26, 2011). This log book should report progress made, problems encountered and solutions found. Each entry should be dated and time stamped. Always bring your log when you come to me or Remo or Chris.
3. A summary of your project that highlights its successes and shortcomings, as you perceive them. This is just a page long.
4. A collection of two to five sample programs that highlight the error recovery, code improvement and shortcomings of your compiler.

%{

/\* Rascal lexical analyzer in FLEX \*/

%}

Letter [A-Za-z]

Digit [0-9]

ID {Letter}({Letter}|{Digit}|\_)\*

A [aA]

B [bB]

C [cC]

D [dD]

E [eE]

F [fF]

G [gG]

H [hH]

I [iI]

J [jJ]

K [kK]

L [lL]

M [mM]

N [nN]

O [oO]

P [pP]

Q [qQ]

R [rR]

S [sS]

T [tT]

U [uU]

V [vV]

W [wW]

X [xX]

Y [yY]

Z [zZ]

WHITE [ \t\n\f]

%%

"(\*" {ECHO;comment();}

{Digit}+ {ECHO;

yylval = atoi(yytext);

return(INTEGER); }

":=" {ECHO;return(ASSIGN);}

";" {ECHO;return(SEMICOLON);}

":" {ECHO;return(COLON);}

"[" {ECHO;return(LBRACKET);}

"]" {ECHO;return(RBRACKET);}

"{" {ECHO;return(LBRACE);}

"}" {ECHO;return(RBRACE);}

"(" {ECHO;return(LPAREN);}

")" {ECHO;return(RPAREN);}

".." {ECHO;return(DOTS);}

"," {ECHO;return(COMMA);}

"." {ECHO;return(PERIOD);}

"=" {ECHO;return(EQ);}

"<>" {ECHO;return(NE);}

"<" {ECHO;return(LT);}

"<=" {ECHO;return(LE);}

">" {ECHO;return(GT);}

">=" {ECHO;return(GE);}

"+" {ECHO;return(PLUS);}

"-" {ECHO;return(MINUS);}

"\*" {ECHO;return(TIMES);}

"/" {ECHO;return(DIV);}

"%" {ECHO;return(MOD);}

{C}{O}{N}{S}{T}({S})? {ECHO;return(CONST);}

{U}{N}{I}{T}({S})? {ECHO;return(UNIT);}

{A}{L}{I}{A}{S}({E}{S})? {ECHO;return(ALIAS);}

{N}{U}{M}{B}{E}{R}({S})? {ECHO;return(NUMBER);}

{V}{A}{R}({S})? {ECHO;return(VAR);}

{P}{R}{O}{G}{R}{A}{M} {ECHO;return(PROGRAM);}

{P}{R}{O}{C}{E}{D}{U}{R}{E} {ECHO;return(PROCEDURE);}

{A}{R}{R}{A}{Y} {ECHO;return(ARRAY);}

{O}{F} {ECHO;return(OF);}

{B}{E}{G}{I}{N} {ECHO;return(BEGINN);}

{E}{N}{D} {ECHO;return(END);}

{I}{F} {ECHO;return(IF);}

{T}{H}{E}{N} {ECHO;return(THEN);}

{E}{L}{S}{E} {ECHO;return(ELSE);}

{F}{O}{R} {ECHO;return(FOR);}

{D}{O} {ECHO;return(DO);}

{T}{O} {ECHO;return(TO);}

{D}{O}{W}{N}{T}{O} {ECHO;return(DOWNTO);}

{W}{H}{I}{L}{E} {ECHO;return(WHILE);}

{ID} {ECHO; return(IDENT);}

{WHITE} {ECHO;}

. {ECHO;return(BAD);}

%%

comment(){

int c;

c = input(); putchar(c);

do

{

while ((c != '\*') && (c != EOF))

{

c = input();

putchar(c);

}

c = input();

putchar(c);

} while ((c != ')') && (c!=EOF));

}

yywrap(){

return(1);

}

%{

/\* Rascal parser in Bison \*/

#include <stdio.h>

#include <ctype.h>

%}

%token CONST UNIT ALIAS NUMBER VAR

%token PROGRAM PROCEDURE ARRAY OF

%token BEGINN END IF THEN ELSE WHILE DO FOR TO DOWNTO

%token INTEGER IDENT

%token SEMICOLON COLON DOTS COMMA PERIOD

%token LBRACKET RBRACKET LPAREN RPAREN LBRACE RBRACE

%token BAD

%token ASSIGN

%nonassoc EQ NE LT LE GT GE

%left PLUS MINUS

%left TIMES DIV MOD

%%

program:

program.heading decl.part proc.part statement.part PERIOD

{printf("\n success \n");};

program.heading:

PROGRAM IDENT SEMICOLON |

error BEGINN {yyerrok; yychar = BEGINN;} |

error SEMICOLON {yyerrok; yyclearin;} ;

decl.part:

unit.decl.part constant.decl.part variable.decl.part;

unit.decl.part:

UNIT unit.decl.list SEMICOLON |

/\* empty \*/ ;

unit.decl.list:

unit.decl.list COMMA unit.def |

unit.def ;

unit.def:

IDENT aliases ;

aliases:

ALIAS LPAREN alias.list RPAREN |

/\* empty \*/ ;

alias.list:

alias.list COMMA IDENT |

IDENT ;

constant.decl.part:

CONST const.decl.list SEMICOLON |

/\* empty \*/ ;

const.decl.list:

const.decl.list SEMICOLON constant.def |

constant.def ;

constant.def:

IDENT eq constant.and.units;

eq:

EQ |

ASSIGN { /\* deserves a warning \*/ } ;

constant.and.units:

INTEGER maybe.units |

IDENT ;

maybe.units:

unit.name |

/\* empty \*/ ;

unit.name:

IDENT PERIOD UNIT |

IDENT |

NUMBER ;

variable.decl.part:

VAR variable.decl.list SEMICOLON |

/\* empty \*/ ;

variable.decl.list:

variable.decl.list SEMICOLON variable.decl |

variable.decl ;

variable.decl:

identifier.list COLON variable.unit ;

identifier.list:

identifier.list COMMA IDENT |

IDENT ;

variable.unit:

ARRAY LBRACKET constant DOTS constant RBRACKET OF unit.name |

unit.name ;

constant:

INTEGER |

IDENT ;

proc.part:

proc.decl.list |

/\* empty \*/ ;

proc.decl.list:

proc.decl.list proc.definition SEMICOLON |

proc.definition SEMICOLON ;

proc.definition:

proc.heading decl.part statement.part ;

proc.heading:

PROCEDURE IDENT formal.args SEMICOLON ;

formal.args:

LPAREN formal.list RPAREN |

/\* empty \*/ ;

formal.list:

formal.list COMMA IDENT |

IDENT ;

statement.part:

BEGINN statement.list END ;

statement.list:

statement.list SEMICOLON statement |

statement ;

statement:

simple.statement |

structured.statement |

error SEMICOLON { yyerrok; yychar = SEMICOLON; } ;

simple.statement:

variable ASSIGN expression |

procedure.call |

/\* empty \*/ ;

variable:

IDENT LBRACKET expression RBRACKET |

IDENT ;

expression:

expression PLUS expression |

expression MINUS expression |

expression TIMES expression |

expression DIV expression |

expression MOD expression |

cast variable |

variable |

cast LPAREN expression RPAREN |

LPAREN expression RPAREN |

cast INTEGER |

INTEGER ;

cast:

LBRACE unit.name RBRACE ;

procedure.call:

IDENT LPAREN call.list RPAREN |

IDENT ;

call.list:

call.list COMMA IDENT |

IDENT ;

structured.statement:

statement.part |

if.statement |

repetitive.statement ;

if.statement:

IF test THEN rest.of.if ;

rest.of.if:

statement ELSE statement |

statement ;

repetitive.statement:

WHILE test DO statement |

FOR IDENT ASSIGN for.list DO statement ;

test:

expression EQ expression |

expression NE expression |

expression LT expression |

expression LE expression |

expression GT expression |

expression GE expression ;

;

for.list:

expression TO expression |

expression DOWNTO expression ;

%%

yyerror() {

printf("\n syntax error \n");

}

#include "rascal.lex.c"

int main(int argc, char \*\*argv ) {

int result;

++argv, --argc; /\* skip over program name \*/

if ( argc > 0 ) yyin = fopen( argv[0], "r" );

else yyin = stdin;

result = yyparse();

system("pause()");

return(result);

}