HIGH POINT UNIVERSITY

Department of Computer Science

CSC4510 Programming Language Design and Translation

The Syntax of *lille*

<prog>	::=	**program** <ident> **is** <block> **;**
<block>	::=	{ <declaration> }* **begin** <statement_list> **end** [<ident>]
<declaration>	::=	<ident_list> : [**constant**] <type>
		[**:=**<number> | **:=**<string> | **:=**<bool>] ;
	|	**procedure** <ident> [**(** <param_list> **)**] **is** <block> **;**
	|	**function** <ident> [**(**<param_list>**)**] **return** <type> **is** <block> **;**
<type>	::=	**integer** | **real** | **string** | **boolean**
<param_list>	::=	<param> { **;** <param> }*
<param>	::=	<ident_list> **:** <param_kind> <type>
<ident_list>	::=	<ident> { **,** <ident> }*
<param_kind>	::=	**value** | **ref**
<statement_list>	::=	<statement> **;** { <statement> **;** }*
<statement>	::=	<simple_statement>
	|	<compound_statement>
<simple_statement>	::=	<ident> [**(** <expr> { **,** <expr> }* **)**]
	|	<ident> **:=** <expr>
	|	**exit** [**when** <expr>]
	|	**return** [<expr>]
	|	**read** [**(**] <ident> { **,** <ident> }* [**)**]
	|	**write** [**(**] <expr> { **,** <expr> }* [**)**]
	|	**writeln** [**(**] [<expr> { **,** <expr> }*] [**)**]
	|	**null**
<compound_statement>	::=	<if_statement>
	|	<loop_statement>
	|	<for_statement>
	|	<while_statement>

<if_statement> ::= **if** <expr> **then** <statement_list>
{ **elsif** <expr> **then** <statement_list> }*
[**else** <statement_list>]
end if

<while_statement> ::= **while** <expr> <loop_statement>

<for_statement> ::= **for** <ident> **in** [**reverse**] <range> <loop_statement>

<loop_statement> ::= **loop** <statement_list> **end loop**

<range> ::= <simple_expr> **..** <simple_expr>

<expr> ::= <simple_expr> [<relop> <simple_expr>]
| <simple_expr> **in** <range>

<bool> ::= **true**
| **false**

<relop> ::= **>** | **<** | = | **<>** | **<=** | **>=**

<simple_expr> ::= <expr2> { <stringop> <expr2> }*

<stringop> ::= **&**

<expr2> ::= <term> { { <addop> | **or** } <term> }*

<addop> ::= + | –

<term> ::= <factor> { { <multop> | **and** } <factor> }*

<multop> ::= * | /

<factor> ::= <primary> [** <primary>]
| [<addop>] <primary>

<primary> ::= **not** <expr>
| **odd** <expr>
| **(** <simple_expr> **)**
| <ident> [**(** <expr>{ , <expr> }* **)**]
| <number>
| <string>
| <bool>

<string> ::= **"** { <*character*> }+ **"**

<ident> ::= letter { [_] <ident2> }*

<ident2> ::= letter | digit

<number> ::= <digit_seq> [**.** <digit_seq>] [<exp> <addop> <digit_seq>]

<digit_seq> ::= digit { digit }*

<exp> ::= E | e

<pragma> ::= **pragma** <pragma_name> [**(** <digit_seq> | <ident> **)**] ;

<pragma_name> ::= error_limit | trace | untrace | debug

The reserved words of the language are:

And	Begin	Boolean	Constant	Else	Elsif	End	Eof	Exit
False	For	Function	If	In	Integer	Is	Loop	Not
Null	Odd	Or	Pragma	Procedure	Program	Read	Real	Ref
Return	Reverse	String	Then	True	Value	When	Write	Writeln
While								

Special symbols in the language are:

:=	;	:	,	(	)	..	<	>	=
<>	<=	>=	&	+	–	*	/	**	"

The Lexical Structure of *lille*

(a) There are three "symbol-separators" in lille: a blank (space), a tab and the end of a line (perhaps with a preceding comment, see (g) below). Zero or more symbol-separators may occur between any two consecutive symbols, before the first symbol of a program or after the last symbol of a program. There must be at least one symbol-separator between any pair of numbers or any pair of identifiers (reserved words are a special form of identifier).

(b) The syntax of the language *lille* is specified above; in these syntax rules,

- "character" is any of the printable ASCII characters (including blank),
- "letter" is any of the letters of the English alphabet, upper or lower case, and
- "digit" is any of the digit characters "0", "1", ..., "9".

(c) Numbers are in decimal notation. The "E" in the syntax of a number may be in either upper or lower case. Note that

- a number is real if it contains a period ("."), otherwise it is integer,
- an integer may only have a non-negative exponent after "E", if an exponent is specified,

(d) Any character as defined in (b) above may occur in a string. No case conversion is to be performed on strings. A string must be contained completely on one line of the source program (not divided between lines) and may be of any length (although the operating system and its utilities may impose some limit on the length of a line). If it is desired to have a '"' character in a string, it is simply entered twice. For example,

"I said ""Help"""

is the string ‘I said "Help"’. If no closing quote is found before the end of the line on which the string started, an error will be reported by the lexical analyser.

(e) Unprintable characters should be skipped by the character handler; that is, they should not be listed and should not be passed to the scanner.

(f) All characters in an identifier (or a keyword representation) are significant, and there is no limit on the length of an identifier.

(g) Comments in *lille* begin with the sequence “--” and extend to the end of the line containing that sequence. A comment may occur between any two symbols; for example,

```
program p -- A program that does something
is --- another comment
begin a -- an (undeclared) variable
:= 3 end;
-- a final comment
```

is a syntactically (but not semantically) correct *lille* program. (Note also that the final comment should be listed by the compiler.)

(h) Pragmas can occur anywhere and should be dealt with by the lexical analyzer. They are not to be dealt with by the parser. The pragma names are *not* reserved words. The pragmas have the following meanings:

error_limit(*n*)
- Sets the maximum number of errors reported to *n*. If more than n errors are detected the compiler aborts compilation but still produces a complete listing file (i.e., no further errors are detected and flagged).

trace(*v*)
- Causes the compiler to generate code which will print out the name and value of the variable *v* every time it is modified. *v* must have already been defined and in scope.

untrace(*v*)
- Stops generation of the tracing code. *v* must already have been defined and in scope.

debug(*on*)
- Turns tracing on for all variables.

debug(*off*)
- Turns tracing off for all variables. Variables for which tracing was turned on with the pragma *trace* will continue to be traced.

The Type Rules of *lille*

type *ARITH* **is** *INT* | *REAL*
type *ORDERED* **is** *ARITH* | *STRING* | *BOOLEAN*

<ident_list> : **constant** <type> *: INT* **:=** <number> *: INT*;
<ident_list> : **constant** <type> *: REAL* **:=** <number> *: ARITH* ;
<ident_list> : **constant** <type> *: STRING* **:=** <string> *: STRING* ;
<ident_list> : **constant** <type> *: BOOLEAN* **:=** <bool> *: BOOLEAN* ;
<ident_list> : <type> *: ORDERED* ;
procedure <ident> [(<param_list>)] **is** <block> ;
function <ident> [(<param_list>)] **return** <type> *: ORDERED* **is** <block> ;

integer *: INT* → *INT*
real *: REAL* → *REAL*
string *: STRING* → *STRING*
boolean *: BOOLEAN* → *BOOLEAN*

<ident_list> : <param_kind> <type> *: ORDERED*

<ident> *: LIST[ORDERED]* [(<expr> *ORDERED* { , <expr> *ORDERED* }*)]
<ident> *: INT* := <expr> *: INT*
<ident> *: REAL* := <expr> *: ARITH*
<ident> *: STRING* := <expr> *: STRING*
exit when <expr> *: BOOLEAN*
return <expr> *: ORDERED*

read <ident> *: ARITH*
write <expr> *: ARITH*
write <expr> *: STRING*
writeln <expr> *: ARITH*
writeln <expr> : *STRING*
eof → *BOOLEAN*
not <expr> *: BOOLEAN* → *BOOLEAN*
<factor> *: BOOLEAN* **and** <factor> *: BOOLEAN* → *BOOLEAN*
odd <expr> *: INTEGER* → *BOOLEAN*
real2int (<expr> *: ARITH*) → *INTEGER*

int2real (<expr> *: ARITH*) → *REAL*

int2string (<expr> *: ARITH*) → *STRING*

real2string (<expr> *: ARITH*) → *STRING*

if <expr> *: BOOLEAN* **then** <statement_list>

{ **elsif** <expr> *: BOOLEAN* **then** <statement_list> }*

[**else** <statement_list>]

end if

while <expr> *: BOOLEAN* <loop_statement>

for <ident>*: INT* **in** [**reverse**] <simple_expr>*: INT* .. <simple_expr>*: INT* <loop_statement>

<simple_expr> *: ARITH* <relop> <simple_expr> *: ARITH* → *BOOLEAN*

<simple_expr> *: BOOLEAN* <relop> <simple_expr> *: BOOLEAN* → *BOOLEAN*

<simple_expr> : *INT* **in** <simple_expr> *: INT* .. <simple_expr> *: INT* → *BOOLEAN*

<expr2> : *STRING* { <stringop> <expr2> **:** *STRING* }+ → *STRING*

<expr2> : *STRING* <stringop><expr2> *: STRING* → *STRING*

<expr2> : *ORDERED* → *ORDERED*

<term> *: INT* { <addop> <term> *: INT* }+ → *INT*

<term> *: INT* { <addop> <term> *: REAL* }+ → *REAL*

<term> *: REAL* { <addop> <term> *: INT* }+ → *REAL*

<term> *: REAL* { <addop> <term> *: REAL* }+ → *REAL*

<term> *: BOOLEAN* { **or** <term> *: BOOLEAN* }+ → *BOOLEAN*

<term> *: ARITH* → *ARITH*

<term> *: ORDERED* → *ORDERED*

<factor> *: INT* { <multop> <factor> *: INT* }+ → *INT*

<factor> *: INT* { <multop> <factor> *: REAL*}+ → *REAL*

<factor> *: REAL* { <multop> <factor> *: INT* }+ → *REAL*

<factor> *: REAL* { <multop> <factor> *: REAL* }+ → *REAL*

<factor> *: STRING* { <multop> **and** *: STRING* }+ → *STRING*

<factor> *: ORDERED* → *ORDERED*

<primary> *: REAL* ** <primary> *: INT* → *REAL*

<primary> *: INT* ** <primary> *: INT* → *INT*

<primary> *: ORDERED* → *ORDERED*

<ident> *: ORDERED* → *ORDERED*

<number> *: ARITH* → *ARITH*

<string> *: STRING* → *STRING*

<ident> *: LIST[ORDERED]* [(<expr>*: ORDERED* { , <expr>*: ORDERED* }*)] → *ORDERED*

(<simple_expr> *: ORDERED*) → *ORDERED*

true → *BOOLEAN*

false → *BOOLEAN*

These are the type rules for the language *lille*. The building up of parameter lists for procedure declarations is a part of the environment handling (symbol table) section of your compiler and hence has not been covered here.

Note that *int2real, real2int, int2string,and real2string* are not reserved words of the language. They are all predefined functions of the language and as such may be redefined by the user.

The Semantics

The Static Semantics of *lille*

Scope Rules

(a) The scope rules used in *lille* are identical to those used in programming languages such as Ada, Modula, and Pascal. Identifiers are only visible within the block (including enclosed blocks) they are declared, and only after the point where they are defined. Identifiers can only be used after they are declared.

(b) Names must have a unique meaning at a particular block level of a *lille* program. (That is, names must be unambiguous). *lille* does not permit function and procedure overloading, except in the case of the language defined procedures **read**, **write** and **writeln**, and the predefined functions *int2real*, *real2int*, *int2string*, and *real2string*.

(c) A name must be declared before it is used. Note that it will not be possible to write *lille* programs involving certain kinds of indirect recursion. For example, the following is illegal in *lille*:

```
procedure B is begin ... A ... end B; -- A not yet declared
procedure A is begin ... B ... end A;
```

although the following is legal:

```
procedure A is
 procedure B is begin ... A ... end B; -- A has been declared
 begin ... B ... end A;
```

Declarations

(a) If a declaration includes the keyword "**constant**", this is the declaration of a named constant; such a declaration must also include "**:=**" followed by a number, string or a boolean.

(b) If "**constant**" is omitted from a declaration, this is the declaration of a variable and the declaration must not contain "**:=**" followed by a number, string or a boolean. (There is no variable initialisation in *lille*.)

(c) If an identifier is given after the keyword "**end**" in a block, it must be the same as the name of the program, function or procedure of which this block is the body.

(d) Functions may only have value parameters.

Statements

(a) The identifier on the left-hand side of an assignment statement must be a variable or parameter name that is both accessible and modifiable.

(b) The identifier in a procedure or function call must be the name of a procedure or function definition respectively.

(c) The identifier in a "**read**" statement must be the name of a variable or parameter of appropriate mode visible to the block containing the "**read**" statement. Only integer and real values may be read in

from the keyboard. The identifiers following the keyword "**read**" may, or may not, be enclosed in parenthesis.

(d) The expression in a "**write**" or "**writeln**" statement must be a string, integer or real. The expressions following the keyword "**write**" or "**writeln**" may, or may not, be enclosed in parenthesis.

(e) A function body must contain a return statement that returns a value of the appropriate type. A "**return**" statement with no trailing expression is permitted in the main program and in a procedure. Each time a "**return**" statement is encountered the trailing expression, if any, is evaluated and the block terminated.

(f) The for loop control variable is implicitly declared by the for statement, and its scope is only the statement sequence contained by the for loop. The for loop control variable behaves as though it is a constant of type integer. The for loop control variable may not be modified through an assignment. The for loop control variable may not be referenced by the expressions which denote the lower and upper bound of the for loop control variable's value. If a variable of the same name as the for loop control variable already exists, then the variable is masked (i.e., made unavailable) in the statement sequence within the for loop. This is the same semantics as in the programming language Ada.

Expressions

(a) If a primary consists of an identifier, it must be the name of an accessible constant or variable, function, or parameter.

The Dynamic Semantics of the Language *lille*

(a) The dynamic semantics of assignment, function call and procedure call statements in lille are the same as the corresponding statements in Ada.

(b) A conditional "**exit**" statement jumps to the point after the appropriate "**end loop**" only if the condition after the keyword "**when**" evaluates to true. An "**exit**" statement with no "**when**" clause is equivalent to "exit when true". An "**exit**" statement may not occur outside of a loop.

(c) The "**read**" statement causes values to be input from the input file and to be assigned to the specified variables.

(d) The "**write**" statement causes the given value of type string, integer or real to be written onto the output file. The value specified in a single "**write**" statement is written onto a single line of the output file. Note that, in the case of a string parameter, only those characters present in the string should be printed; in particular, no extra leading or trailing blank characters should be printed. The "write" procedure takes a single parameter only.

(e) The "**writeln**" statement works in the same way as the "**write**" statement except it also terminates output to the current line and any additional output will appear on a new line. If no argument is provided to the "**writeln**" statement, output is terminated to the current line and the next output will appear on a new line.

(f) When executing an "**if**" statement, the conditions in the "**if**" and "**elsif**" parts are evaluated sequentially from the first to the last; if one is found that is "true", the corresponding list of statements is executed and control then passes to the point after the "**end if**". If none of the conditions is true and the "if" statement contains an "else" part, the list of statements in that part is executed.

(g) In executing a "**while**" statement, the condition is evaluated; if it is "**true**", the loop is executed and the condition evaluated again. If the condition is "**false**", control passes to the point after the corresponding "**end loop**".

(h) A simple "**loop**" statement, (a "**loop**" statement without the iterative schemes "**while**" and "**for**" preceding it) is an "infinite" loop, in the sense that if control reaches the bottom of the loop, it then passes unconditionally to the top of the loop. Clearly the loop can terminate, using an "exit" statement.

(i) The condition "**odd**(i)" evaluates to "**true**" if the integer "i" is odd, and to "**false**" otherwise. "**Odd**" takes a single argument of type integer.

(j) The condition "p and q evaluates to "**true**" if the boolean expressions p and q are both "true", "**false**" otherwise.. "**And**" requires two boolean expesssions.

(k) The condition "p or q" evaluates to "**true**" if either of the boolean expressions p or q are "**true**", "**false**" if neither p nor q are "true". "**Or**" requires two boolean expressions.

(l) The range test "i **in** lb .. ub" is equivalent to: " $lb \leq i \leq ub$".

(m) The relational operators "**=**", "**<>**" (not equal), ..., "**>=**" have the same meanings as the corresponding ones in C++. (As usual, "**false**" < "**true**")

(n) The function "**not**" causes the expression to evaluate to the logical complement of the value of the operand. That is, "not true" yields "**false**" and "**not false**" yields "**true**. "**Not**" is applicable to a boolean expression.

(o) The conditions "**true**" and "**false**" have their obvious meanings.

(p) The function "**eof**" evaluates to "**true**" if no further values remain on the input file, and to "**false**" otherwise.

(q) The arithmetic operators "**+**", "**–**", "*****", and "/" have the same meanings as in C++; the first two of these are unary as well as binary operators, and the last two are binary only. The operator "******" represents exponentiation and raises the first operand to the power given by the second operand (which must be an integer).

(r) A function must return a value and all parameters to the function must be value mode only (i.e., read-only). Your compiler is free to delay detection of the fact that a function did not execute a "return" statement until run-time.

(s) The predefined function "real2int" takes a real expression as an argument and returns the corresponding real number.

(t) The predefined function "int2real" takes an integer expression as an argument and returns the corresponding integer number (i.e., the real number truncated toward zero).

(u) The predefined function "int2string" takes an integer argument and returns the corresponding string value.

(v) The predefined function "real2string" takes a real argument and returns the corresponding string value.

(w) The string operator "**&**" performs string concatenation and produces a new string.

(x) In executing a for loop, the range <lb> **..** <ub> must be such that <lb> and <ub> are integer expressions. The expressions are to be evaluated once only before the loop is executed. If the value of <lb> is greater than <ub> then the loop has no effect and the contained statement sequence is not executed. The for loop control variable starts with the value of <lb> and the loop is executed until the value of the for loop control variable exceeds <ub>. The for loop control variable is incremented by one at the end of each iteration of the loop.

If the keyword "**reverse**" is present, then the for loop control variable starts with the value of <ub> and the loop is executed until the value of the for loop control variable is less than <lb>. The for loop control variable is decremented by one at the end of each loop iteration.

Integer to real conversion happens automatically if necessary (e.g., assignment of an integer value to a real variable). Real to integer conversion must be explicit via the predefined function "int2real".

(y) If a parameter is of mode "**ref**", then any assignment to the formal parameter is immediately reflected by a change in value to the corresponding actual parameter.

(z) If a parameter is of mode "**value**" then any assignment to the formal parameter does not affect the value of the corresponding actual parameter. Assignment is permitted to a "**value**" mode parameter and its value is updated, but the updated value is not copied back into the actual parameter on successful completion of the procedure or function call.

(aa) The dynamic semantics are specified more fully by examining the PAL code generation sequences for *lille* programs. These sequences specify the operational semantics of *lille* programs.

Michael Oudshoorn
August 13, 2023