DotPar
Language Reference Manual
Team 3

Logan Donovan - lrd2127@columbia.edu
Justin Hines - jph2149@columbia.edu
Andrew Hitti - aah2147@columbia.edu
Nathan Hwang - nyh2105@columbia.edu
Sid Nair - ssn2114@columbia.edu

April 12, 2012

1 Introduction

DotPar is a flexible language that provides implicit nested data parallelism with powerful first-class functions while providing a familiar imperative interface. The focus is on parallel performance for arrays and loops, since these are often the source of performance bottlenecks.

Previous implementations of nested data-parallel languages have relied on the use of purely functional paradigms which can make the learning-curve steep and difficult to those who just want to pick up a new tool. DotPar provides a multi-paradigm nested data-parallel language with a friendly imperative style in addition to powerful functions. Its implicit parallelism focuses on the performance of arrays and loops, but its implementation requirements are flexible enough for other forms of parallelism as well.

2 Lexical Convention

A program consists of zero or more input statements, and one or more lines stored in files. These are translated in several phases, which are described in Section 13. The first phases do low-level lexical transformations, which reduce the program to a series of tokens.

lines:

imports_opt external_declaration
lines external_declaration

imports:

imports import_declaration
import_declaration

import_declarations will be explained in detail in Section 11.

3 Tokens

DotPar has 5 token classes: identifiers, keywords, literals, operators, and other separators. All white space, collectively including blanks, horizontal and vertical tabs, newlines, and comments as described below are ignored except as they separate certain tokens. Sometimes white space is required to separate otherwise adjacent identifiers, keywords, and literals.

3.1 Comments

The characters /* introduce a block comment, which terminates with the characters */. In addition, // adds line comments which converts everything that follows to a comment until the end of the line. Comments do not nest, and they do not occur within string or character literals.

3.2 Constants

Constants are character, number, or string literals, as well as true, false, or nil.
constant

- CHAR_LITERAL
- NUM_LITERAL
- STRING_LITERAL
- TRUE
- FALSE
- NIL

3.2.1 Character Literal

A character constant specifies a single ascii character surrounded by single quotes, such as 'a'. If one is so inclined, he or she can include such control characters as newlines by using traditional backslash delimited escape codes. For instance, a newline is '\n', and '\0' is the null character. A notable exception is the backslash itself, which is merely '\'.

A char literal has the type char and is initialized with the given ascii character. The behavior of a program that attempts to alter a char literal is undefined.

3.2.2 Number Literal

There are two ways to represent literal numbers with DotPar: the first is an integral format, without a decimal point, and the second is a floating point representation, including a decimal point.

The integral representation is merely a series of digits without spaces between them, like 31 or 42. Integral number literals are only available for base 10.

The floating point representation is a series of digits with a period embedded within or prepended to the front, and without spaces anywhere. Examples include 3.14159, or .11235, but not 125. with a trailing period.

The floating point representation is restricted to base 10, and there is no support for scientific notation.

A number literal has a type number and is initialized with the given value: if the given literal is too large, the behavior of the program will be undefined.

3.2.3 String Constant

A string literal is a sequence of ascii characters surrounded by double quotes as in ". . . ". A string has type 'array of characters' and is initialized with the given characters. One may include traditional backslash escape codes, similar to character constants explained above. The behavior of a program that attempts to alter a string literal is undefined.

3.2.4 Boolean Constant

There are only two values for a boolean constant, `true` or `false`. Their literal value is respectively represented as `true` and `false`, not decorated by any special characters.

3.2.5 Nil Constant

`nil` is a special literal that can stand in for any other type of value. `nil` evaluates to `false`. `nil` and `false` are the only values in the language that evaluate to `false`.

3.3 Identifiers

An identifier is a sequence of letters and digits, and underscores. The first character must be a letter. Identifiers are case-sensitive and may have any length.

3.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

- `import`
- `true`
- `false`
- `boolean`
- `char`
- `func`
- `number`
- `void`
- `if`
- `else`
- `elif`
- `for`

- `in`
- `return`

4 Syntax Notation

The syntax in this manual has categories written in italics and literals and characters in plain font. Optional terminal and nonterminal symbols have the subscript "opt," like below:

{expression_opt}

*An non-terminal with the name foo_opt will either go to "the empty string" or foo. Similarly, lists follow the format of imports above. To avoid repetition, rules that follow this rigid structure will not be explained in-depth. However, they will be included in the complete grammar at the end of this reference manual.

5 Meaning of Identifiers

Identifiers are simply the names given to functions and variables. A variable is used to store data and points to a specific location. The interpretation of the data is based on the variable's type. Identifiers are limited to the scope within which they are defined and are only accessible within that scope. This can be only within a specific function or an entire program.

5.1 Basic Types

DotPar contains three basic types: number, boolean and char. Variables of type number are 64-bit double precision floating point numbers. Boolean variables have only two possible values: true and false. They are internally treated as a 32-bit entities. Char variables can store any member of the set of characters. Every character has its value as equivalent to the integer code for the character.

5.2 Derived Types

These include arrays and functions which can be constructed from the basic types. Derived types include:

- *arrays* of elements of a given type

- *functions* accepting variables of certain types, and returning variables of a given type

There are infinitely many derived types. Type declaration syntax are described in more detail below.

6 Objects and lvalues

A variable is a named region of storage and an lvalue is an expression referring to that variable. For example, an lvalue expression can be an identifier with a specified type. Like in C, the term "lvalue" originates from the term left-value which indicates that it was on the left side of the assignment operator. Each operator listed below can expect lvalue operands and yields an lvalue.

7 Expressions

The precedence of expression operators is the same as the order of the major subsections outlined below in this section, with highest precedence first. In all subsections, precedence follows C-style conventions. Left- or right-associativity is specified in each subsection for the operators discussed therein. The grammar given in Section 13 incorporates the precedence and associativity of the operators.

The precedence and associativity of the operators is fully specified, but the order of evaluation of expressions is undefined, even if the subexpressions involve side effects. However, each operator combines the values it produces by its operands in a way that is compatible with the parsing of the containing expression in which it appears. The handling of overflow, divide check, and other exceptions in expression evaluation is undefined by the language spec.

7.1 Primary Expressions

Primary expressions are identifiers, constants, string literals, or expressions in parentheses.

primary_expression:
IDENTIFIER
constant
(expression)

An identifier is a primary expression. Its type is specified by its declaration, which must occur earlier in the program.

A constant is a primary expression. Its type depends on its form as discussed in section 2.2

A string literal is a primary expression. It is actually converted to an array of characters, but string literals are included as part of the grammar for programmer convenience.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

7.2 Postfix Expressions

The operators in postfix expressions group left to right.

postfix_expression:
primary expression
postfix_expression [expression]
postfix_expression (argument_expression_list_pt)

7.2.1 Array References

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. The postfix expression must evaluate to an array and the expression must evaluate to a type number.

7.2.2 Function call

A postfix expression followed by the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions, which constitute the arguments to the given function.

The term argument is used for an expression passed by a function call; the term parameter is used for an input object (or its identifier) received by a function definition, or described in function declaration. The scope is lexical.

Arguments are passed by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments.

However, array references are passed by value, so a function may modify the contents of an array passed to it.

Type agreement is strict between the parameters and arguments. Thus, DotPar is strongly and statically typed.

The order of evaluation of arguments is unspecified. In addition, the arguments and their side effects need not be fully evaluated before the function is entered if the compiler can guarantee that this does not affect the correctness of the program.

7.3 Unary Expressions

Expressions with unary operators group right-to-left.

unary_experession:
    postfix_expression
    NOT unary_expression
    SUB unary_expression

The NOT unary_expresssion refers to the ! operator. The operand of the ! operator must have a boolean type, and the result is true if the value of its operand evaluates to false, and false otherwise. The type of the result is boolean.

The SUB unary_expression refers to the - operator. The operand of the - operator must have a number type, and the result is the negation of the value of its operand. The type of the result is number.

7.4 Arithmetic Expressions

Arithmetic Expressions are grouped left to right, and include: %, /, *, + , - but this is infact is not useful, as it is parsed to preserve C style precedence. All operators return number types.

arithmetic_expression:
  unary_expression
  arithmetic_expression REM arithmetic_expression
  arithmetic_expression DIV arithmetic_expression
  arithmetic_expression MULT arithmetic_expression
  arithmetic_expression ADD arithmetic_expression
  arithmetic_expression SUB arithmetic_expression

REM refers to the % operator, the remainder which refers to the remaining value after quotient the first operand by the second. Note that the remainder of a negative value will be negative.

DIV refers to the / operator, the division which refers to quotient of the first operand and the second.

MULT refers to the * operator, the multiplication between its two operands.

ADD refers to the + operator, the addition between its two operands.

SUB refers to the binary - operator, the subtraction between its two operands.

7.5 Relational Expressions

The relational operators group left-to-right.

relational_expression:
  arithmetic_expression
  relational_expression GEQ relational_expression
  relational_expression GT relational_expression
  relational_expression LT relational_expression
  relational_expression LEQ relational_expression
  relational_expression EQ relational_expression
  relational_expression NEQ relational_expression

The GEQ refers to >= operators(greater or equal), GT to >(greater), LT to <(less than), LEQ to <= (less than or equal), EQ to == (equality), and NEQ to != (not equality). All yield false if the specified relation is false, and true if it is true. The type of the result is boolean. The usual arithmetic conversions are performed on arithmetic operands. GEQ, GT, LEQ, and LT have higher precedence than EQ and NEQ.

7.6 Conditional Expressions

The conditional operators group left-to-right and include && and ||.

conditional_expression:
    relational_expression
    conditional_expression OR conditional_expression
    conditional_expression AND conditional_expression

OR refers to the ||operator which performs a logical OR on its two operands which must be of type boolean. The result is of type boolean.

AND refers to the && operator which performs a logical AND on its two operands which must be of type boolean. The result is of type boolean.

7.7 Array Expressions

Array Expressions are conditional statements, or list_comprehnesionsand initializer_list_opt enclosed in square brackets '[' ,'['.

array_expression:
    conditional_expression
    [list_comprehension]
    [initializer_list_opt]

An initializer_list is used in the creation of array literals.

7.8 List Comprehension

*List comprehensions are used to succinctly create an array. They are equivalent in power to maps and filters, but are often a more convenient syntax to use. Although the list comprehension may be parallelized during its execution, the ordering of the resulting array is deterministic.

list_comprehension:
array_expression FOR paren_parameter_list_opt IN array_expression if_comp_opt
array_expression FOR paren_parameter_list_opt IN paren_multi_array_expression_list_opt if_comp_opt

The list comprehension syntax and behavior is very similar to Python's.

7.9 Assignment Expressions

There is one assignment operator =, and it is not used more than once per statement.

assignment_expression:
array_expression
anonymous_function_definition
postfix_expression ASSIGN array_expression
postfix_expression ASSIGN function_definition
postfix_expression ASSIGN anonymous_function_definition

The assignment operator requires an lvalue as a left operand, and the lvalue must be mutable.

The assignment operator assigns array_expression, function_definition and anonymous_function_definition to postfix_expressions.

7.10 Anonymous Function Definitions

Anonymous function definitions can be used in assignments. They are identical to regular function definitions, except they lack an identifier.

anonymous_function_definition:
FUNC: type_specifier (parameter_list_opt) compound_statement

8 External Declarations

External declarations form the basic building blocks of lines of code.

external_declaration:
function_definition
declaration

The two top-level declarations are function_definitions and declarations. Function definitions are defined as:

function_definition:
FUNC IDENTIFIER: type_specifier (parameter_list_opt) compound_statement)
compound_statement

Unlike C, functions the return type of a function is defined after the name. With potentially complicated return types, having the type come after the name makes reading the code easier. They also have the `func` keyword so that function syntax for types, definitions, and assignments.

Declarations inform the interpretation given to each identifier. Implementations decide when to reserve storage space associated with the identifier. Declarations have the form

declaration:
type_specifier declarator;
type_specifier declarator ASSIGN initializer;

where ASSIGN is =. A declaration must have one and only one declarator.

8.1 Type Specifiers

A type specifier is either a basic or derived type, examples of which are:

- number
- char[]
- func:number[](char[])

* At most one type-specifier may be given in a declaration. Type specifiers have the form:

type_specifier:
type_specifier [arithmetic_expression]
type_specifier [ ]
basic_type
VOID
func_specifier

func_specifier:
func: type_specifier (type_list)
func: type_specifier (parameter_list_opt)

basic_type:
NUMBER
CHAR
BOOLEAN

8.2 Declarators

Declarators have the syntax:

declarator:
IDENTIFIER
(declarator)

8.3 Initializers

* *initializer:*
array_expression
anonymous_function_definition

As some example types, we can see that array types include number[] and number[][]. For example array can be declared initialized as: number[] a = [1, 2];

An example function declaration is func:number[](number){ } foo. This function returns an array of numbers, and accepts a number as an argument, and is named foo. This syntax allows us to concisely declare variables as functions, which is important for a language with first-class functions.

Note that we could have given a name to the number parameter if we chose to using a parameter_list, which is a series of parameter_declarations:

parameter_declaration:
    type_specifier declarator

Although this identifier would not be used, it may be desirable to name parameter the so that the declaration is self-documenting.

9 Statements

Statements are executed sequentially and have no value. The types of statements in DotPar are listed below:

statement:
    expression_statement
    compound_statement
    selection_statement
    iteration_statement
    jump_statement

9.1 Expression Statements

Most statements are expressions of the form

expression-statement:
    expression_opt;

These statements are mostly assignments and function calls. The side effects of an expression need not be computed before the execution of the following statement if the compiler can guarantee that this does not affect correctness. This is different from other languages like C.

9.2 Compound Statements

Compound statements are multiple statements executed when one is expected. For example, the body of a function is a compound statement. In DotPar, if statements and for loops also require compound statements, even if they have single line bodies.

compound_statement:
statement_list_opt
statement_list :
statement_list_opt statement
statement_list_opt declaration
statement_list_opt function_definition

Identifiers can only be declared once within the scope of a function and cannot be the same as the identifiers passed into a function.

9.3 Selection Statements

Selection statements allow for the choice between multiple control flows.

selection_statement:
if elifs_opt else_opt
if:
IF '(' expression ')' compound_statement
else_opt:
else
else:
ELSE compound_statement
elifs_opt:
elifs
elifs:
ELIF '(' expression ')' compound_statement
elifs ELIF '(' expression ')' compound_statement

Note that `IF`, `ELSE`, and `ELIF`, are used to distinguish the terminal if, else, and elif keywords from the if, else, and elif non-terminals.

The expression within the if statement must evaluate to either true or false. When it is true the substatement is executed. With an if else statement the else substatement is executed when the expression evaluates to false. Every else is connected with the first if statement above it that is unconnected to an else.

9.4 Iteration Statements

Iteration statements specify looping.

> *iteration-statement:*
> *for (expression_opt; expression_opt; expression_opt) compound_statement*

The first expression is evaluated once and marks the start of the loop. Note that a new variable cannot be declared as part of this operation. The second expression is coerced to a boolean; if false, it terminates the loop. The third expression is evaluated after each iteration so it specifies the re-initialization for the loop.

Again, note that curly braces must be used.

9.5 Jump Statements

Jump statements, once they are reached, always transfer control regardless of any condition.

> *jump_statement:*
> *RETURN expression_opt ';'*

A function provides the value back to its caller via the return statement. It returns the value of the expression when it is evaluated. The expression once evaluated is interpreted as the type specified in the function declaration.

10 Scope

A program can be compiled from several .par files. Scoping is lexical. Thus, identifiers in DotPar have one top-level namespace in which variables and functions are defined, with a shared namespace between the two. Imports made in a file are accessible anywhere else in that file. An identifier declared outside of any function can be accessed anywhere in the program. An identifier declared in a block is available anywhere within the block, including inner functions. Note that this means that DotPar has closures. That is, if an inner function is returned from a function, it maintains access to the variables of the outer function.

11 Preprocessing

A preprocessor performs the inclusion of named files. Lines beginning with import communicate with the preprocessor. The syntax of these lines is independent of the rest of the language; they must only appear at the beginning of the file.

11.1 File Inclusion

Imports have the syntax:

> *imports:*
> *imports import_declaration*
> *import_declaration*
>
> *import_declarations:*
> *IMPORT IDENTIFIER*

A control line of the form
 import module;
means this line will be replaced by the contents of the file named filename, with extension .par. The named file is searched for in the current directory, and each file is imported only once per program. Import statements can be nested, so every file can include them.

12 Built-in Functions

The language includes several built-in functions that provide basic building blocks for more complex user-defined functions. These appear to the user as regular library functions. They can be shadowed by user functions to avoid having an unreasonable amount of reserved words. Many of these functions are self-explanatory and, as such, their formal prototypes are not given. Further explanation is present where warranted.

Finally, note that many useful functions, such as string containment, are not included. These are more appropriate as libraries than language features.

12.1 Array

```
cat(arr, other\_arr) // concatenate two arrays and return the result
```

```
each(arr, fn(element, index)) // iterate through an array
fill(dimensions, fn(index)) // fill an array using a function. dimensions
// is an array that has the size of each dimension in the output array
filter(arr, fn(element, index)) // selects certain elements from an
// array; the filter function returns a boolean
len(arr) // array length
map(arr, fn(element, index)) //runs function fn on each element of the array
reduce(sum, nums, 0) // reduce operation. The last argument is the
// initial value.
zip(arr, other\_arr) // combine two arrays into a nested array
```

12.2 String

```
len(str) // string length
```

12.3 Math

Note that all trigonometric functions operate with radians.

```
acos(n)
asin(n)
atan(n)
cos(n)
exp(num, exponent)
ln(n)
log(n, base)
sin(n)
sqrt(n)
tan(n)
ceil(n)
floor(n)
trunc(n)
round(n)
rand(n)
```

12.4 I\O

```
print(s)
println(s)
printerr(s)
read()
```

```
readln()
```

13 Future Goals

We have an ambitious set of possible additions to the language. We do not expect to complete all of them, but we do hope to add at least a few of these features. Also, note that some of these goals are mutually exclusive. For instance, if Java interoperability is implemented, adding some of the library functionality mentioned below will not be necessary. Note that these additions are in no particular order.

1. Add more assignment operators: *=, \=, +=, -=, %=
2. Do basic type inference
3. Add new control flow tools: foreach loops, while, break, and continue
4. Add new container objects, such as a struct or dictionary
5. Introduce shorthand syntax for operations like array concatenation (++)
6. Range selection for arrays and strings, e.g. arr[i:j] or arr[i:j:stride]
7. Java interoperability (this will be hard given static analysis for parallelization)
8. Namespacing
9. Immutable values
10. Library-level functionality
    - String manipulation: trim, contains, split
    - Regexes
    - Arrays: populate (like fill, but for existing arrays), max, min, push, pop
    - Basic data structures
    - Time
    - Char to int function
11. Mutating versions of some functions, such as map! and zip!

12. Function keywords and/or optional params

13. Implement more parallelized aspects in the language

14 Grammar

Note that for the grammar we specify the following precedence following YACC conventions:
\%left OR
\%left AND

\%left EQ NEQ
\%left GT GEQ LT LEQ

\%left ADD SUB
\%left MULT DIV REM

\%right UMINUS

These are the rules for the grammar:

lines	::=	imports_opt external_declaration
lines	::=	lines external_declaration
imports_opt	::=	imports
imports_opt	::=	
imports	::=	imports import_declaration
imports	::=	import_declaration
import_declaration	::=	IMPORT IDENTIFIER ";"
constant	::=	CHAR_LITERAL
constant	::=	NUM_LITERAL
constant	::=	STRING_LITERAL
constant	::=	TRUE
constant	::=	FALSE
constant	::=	NIL
argument_expression_list_opt	::=	argument_expression_list
argument_expression_list_opt	::=	
argument_expression_list	::=	assignment_expression
argument_expression_list	::=	argument_expression_list "," assignment_expression
postfix_expression	::=	primary_expression

postfix_expression	::=	postfix_expression "[" expression "]"
postfix_expression	::=	postfix_expression "(" argument_expression_list_opt ")"
unary_expression	::=	postfix_expression
unary_expression	::=	NOT unary_expression
unary_expression	::=	SUB unary_expression
arithmetic_expression	::=	unary_expression
arithmetic_expression	::=	arithmetic_expression REM arithmetic_expression
arithmetic_expression	::=	arithmetic_expression DIV arithmetic_expression
arithmetic_expression	::=	arithmetic_expression MULT arithmetic_expression
arithmetic_expression	::=	arithmetic_expression ADD arithmetic_expression
arithmetic_expression	::=	arithmetic_expression SUB arithmetic_expression
relational_expression	::=	arithmetic_expression
relational_expression	::=	relational_expression GEQ relational_expression
relational_expression	::=	relational_expression GT relational_expression
relational_expression	::=	relational_expression LT relational_expression
relational_expression	::=	relational_expression LEQ relational_expression
relational_expression	::=	relational_expression EQ relational_expression
relational_expression	::=	relational_expression NEQ relational_expression
conditional_expression	::=	relational_expression
conditional_expression	::=	conditional_expression OR conditional_expression
conditional_expression	::=	conditional_expression AND conditional_expression
opt_paren_multi_array_expression_list	::=	"(" multi_array_expression_list ")"

opt_paren_multi_array_expression_list	::=	multi_array_expression_list
multi_array_expression_list	::=	array_expression "," array_expression
multi_array_expression_list	::=	array_expression "," array_expression "," array_expression_list
array_expression_list	::=	array_expression
array_expression_list	::=	array_expression_list "," array_expression
array_expression	::=	conditional_expression
array_expression	::=	"[" list_comprehension "]"
array_expression	::=	"[" initializer_list_opt "]"
if_comp_opt	::=	if_comp
if_comp_opt	::=	
if_comp	::=	IF expression
list_comprehension	::=	array_expression FOR paren_parameter_list_opt IN array_expression if_comp_opt
list_comprehension	::=	array_expression FOR paren_parameter_list_opt IN opt_paren_multi_array_expression_list if_comp_opt
assignment_expression	::=	array_expression
assignment_expression	::=	anonymous_function_definition
assignment_expression	::=	postfix_expression ASSIGN array_expression
assignment_expression	::=	postfix_expression ASSIGN function_definition
assignment_expression	::=	postfix_expression ASSIGN anonymous_function_definition
expression	::=	assignment_expression
primary_expression	::=	IDENTIFIER
primary_expression	::=	constant
primary_expression	::=	"(" expression ")"
type_specifier	::=	type_specifier "[" arithmetic_expression "]"
type_specifier	::=	type_specifier "[" "]"
type_specifier	::=	basic_type

type_specifier	::=	VOID
type_specifier	::=	func_specifier
func_specifier	::=	FUNC ":" type_specifier "(" type_list ")"
func_specifier	::=	FUNC ":" type_specifier "(" parameter_list_opt ")"
basic_type	::=	NUMBER
basic_type	::=	CHAR
basic_type	::=	BOOLEAN
declaration	::=	type_specifier declarator ";"
declaration	::=	type_specifier declarator ASSIGN initializer ";"
declarator	::=	IDENTIFIER
declarator	::=	"(" declarator ")"
type_list	::=	type_specifier
type_list	::=	type_list "," type_specifier
parameter_list_opt	::=	parameter_list
parameter_list_opt	::=	
paren_parameter_list_opt	::=	"(" parameter_list ")"
paren_parameter_list_opt	::=	parameter_list
parameter_list	::=	parameter_declaration
parameter_list	::=	parameter_list "," parameter_declaration
parameter_declaration	::=	type_specifier declarator
initializer	::=	array_expression
initializer	::=	anonymous_function_definition
initializer_list_opt	::=	initializer_list
initializer_list_opt	::=	
initializer_list	::=	initializer
initializer_list	::=	initializer_list "," initializer
expression_statement	::=	expression_opt ";"
expression_opt	::=	expression
expression_opt	::=	
compound_statement	::=	"{" statement_list_opt "}"
statement_list_opt	::=	statement_list
statement_list_opt	::=	
statement_list	::=	statement_list_opt statement
statement_list	::=	statement_list_opt declaration

statement_list	::=	statement_list_opt function_definition
selection_statement	::=	if elifs_opt else_opt
if	::=	IF "(" expression ")" compound_statement
else_opt	::=	else
else_opt	::=	
else	::=	ELSE compound_statement
elifs_opt	::=	elifs
elifs_opt	::=	
elifs	::=	ELIF "(" expression ")" compound_statement
elifs	::=	elifs ELIF "(" expression ")" compound_statement
iteration_statement	::=	FOR "(" expression_opt ";" expression_opt ";" expression_opt ")" compound_statement
jump_statement	::=	RETURN expression_opt ";"
statement	::=	expression_statement
statement	::=	compound_statement
statement	::=	selection_statement
statement	::=	iteration_statement
statement	::=	jump_statement
anonymous_function_definition	::=	FUNC ":" type_specifier "(" parameter_list_opt ")" compound_statement
function_definition	::=	FUNC IDENTIFIER ":" type_specifier "(" parameter_list_opt ")" compound_statement
external_declaration	::=	function_definition
external_declaration	::=	declaration