Programming Languages 3

Fun Specification

1 Overview

Fun is a simple imperative language. Its purpose is to illustrate some aspects of programming language concepts and implementation.

Here is an example of a Fun program:

```
bool verbose = true

func int fac (int n):  # returns n!
    int f = 1
    while n > 1:
        f = f*n
        n = n-1 .
    return f
.

proc main ():
    int num = read()
    while not (num == 0):
        if verbose: write(num) .
        write(fac(num))
        num = read() .
.
```

This program declares a global variable `verbose`, a function `fac`, and a procedure `main`. The function `fac` has a formal parameter, and ends by returning a result. A procedure may also have a formal parameter (although procedure `main` does not), but does not return a result. Both functions and procedures may declare local variables.

2 Programs

Syntax

prog = *var-decl* * *proc-decl* + *eof*

Scope and type rules

Variables declared at the program level are global in scope. All procedures and functions are global in scope.

No variable may be accessed before it is declared. Likewise, no procedure or function may be called before it is declared; however, a procedure or function may call itself.

The program must include a procedure `main` with no parameter.

Semantics

The program is run by first elaborating its global variable declarations (if any) and then calling the procedure `main`.

3 Declarations

Syntax

proc-decl	=	‘`proc`’ *ident* ‘`(`’ *formal* ‘`)`’ ‘`:`’ *var-decl* * *seq-com* ‘`.`’	– procedure declaration
	|	‘`func`’ *type ident* ‘`(`’ *formal* ‘`)`’ ‘`:`’ *var-decl* * *seq-com* ‘`return`’ *expr* ‘`.`’	– function declaration
formal	=	(*type ident*)?	– formal parameter
var-decl	=	*type ident* ‘`=`’ *expr*	– variable declaration
type	=	‘`bool`’	
	|	‘`int`’	

Scope and type rules

Variables declared inside a procedure or function are local in scope. Formal parameters are treated as local variables.

Every variable has a declared type, either `bool` or `int`; the expression in the variable declaration must have the same type.

Likewise, every formal parameter has a declared type, either `bool` or `int`.

A procedure has type $T \to$ `void` (if it has a formal parameter of type T) or `void` $\to$ `void` (if it has no formal parameter).

A function with result type T' has type $T \to T'$ (if it has a formal parameter of type T) or `void` $\to T'$ (if it has no formal parameter). The expression following ‘`return`’ must have type T'.

Semantics

A variable declaration is elaborated by first evaluating its expression to the value *v*, then creating a variable initialised to *v*, then binding the identifier to the variable.

A procedure or function declaration is elaborated by binding the identifier to the procedure or function.

4 Commands

Syntax

com	=	*ident* ‘`=`’ *expr*	– assignment command
	|	*ident* ‘`(`’ *actual* ‘`)`’	– procedure call
	|	‘`if`’ *expr* ‘`:`’ *seq-com* (‘`.`’ | ‘`else`’ ‘`:`’ *seq-com* ‘`.`’)	– if-command
	|	‘`while`’ *expr* ‘`:`’ *seq-com* ‘`.`’	– while-command
seq-com	=	*com* *	– sequential command

Scope and type rules

In an assignment command, the identifier must be bound to a variable, and the expression must have the same type as that variable.

In an if-command, the expression must be of type `bool`.

In a while-command, the expression must be of type `bool`.

In a procedure call, the identifier must be bound to a procedure. If the procedure has type $T \rightarrow$ `void`, the procedure call must have an actual parameter of type *T*. If the procedure has type `void` → `void`, the procedure call must have no actual parameter.

Semantics

An assignment command is executed by first evaluating its expression to the value *v*, then storing *v* in the variable.

An if-command with no '`else:`' is executed by first evaluating its expression to the `bool` value *b*, and then either (a) executing the command after '`:`' if *b* is true, or (b) doing nothing if *b* is false. An if-command with '`else:`' is executed by first evaluating its expression to the `bool` value *b*, and then either (a) executing the command after '`:`' if *b* is true, or (b) executing the command after '`else:`' if *b* is false.

A while-command is executed by first evaluating its expression to the `bool` value *b*, and then either (a) exiting if *b* is false, or (b) executing the command after '`:`' and then repeating the whole while-command if *b* is true.

A procedure call without an actual parameter is executed by first elaborating the procedure's local variable declarations (if any), then executing the procedure's sequential command, then destroying any local variables. A procedure call with an actual parameter is executed by first evaluating its actual parameter to the value *v*, then creating a local variable (the formal parameter) initialised to *v*, then elaborating the procedure's local variable declarations (if any), then executing the procedure's sequential command, then destroying the formal parameter and any other local variables.

A sequential command is executed by executing its constituent commands in strict order.

5 Expressions

Syntax

expr	=	*sec-expr* (('<' | '>' | '==') *sec-expr*)?	– binary operator application
sec-expr	=	*prim-expr* (('+' | '-' | '*' | '/') *prim-expr*)*	– binary operator application
prim-expr	=	'`false`'	
	|	'`true`'	
	|	*num*	– numeral
	|	*ident*	– variable
	|	*ident* '`(`' *actual* '`)`'	– function call
	|	'`not`' *prim-expr*	– unary operator application
	|	'`(`' *expr* '`)`'	– parenthesized expression
actual	=	*expr*?	– actual parameter

Scope and type rules

A unary operator application has one sub-expression whose type must be consistent with the type of the operator. The type of the unary operator application is determined by the type of the operator. The unary operator '**`not`**' has type `bool` → `bool`.

A binary operator application has two sub-expressions whose types must be consistent with the type of the operator. The type of the binary operator application is determined by the type of the operator.

The binary operators ‘+’, ‘-’, ‘*’, and ‘/’ have type (`int` × `int`) → `int`. The binary operators ‘==’, ‘<’, and ‘>’ have type (`int` × `int`) → `bool`.

In a function call, the identifier must be bound to a function. If the function has type $T \rightarrow T'$, the function call must have an actual parameter of type T. If the function has type `void` → T', the function call must have no actual parameter. In either case the type of the function call is T'.

Semantics

A unary operator application is evaluated by first evaluating its sub-expression to the value v, then applying the unary operator to v.

A binary operator application is evaluated by first evaluating its two sub-expressions to the values v_1 and v_2, then applying the binary operator to v_1 and v_2.

A function call without an actual parameter is evaluated by first elaborating the function’s local variable declarations (if any), then executing the function’s sequential command, then evaluating the ‘`return`’ expression to the value v', then destroying any local variables. A function call with an actual parameter is evaluated by first evaluating its actual parameter to the value v, then creating a local variable (formal parameter) initialised to v, then elaborating the function’s local variable declarations (if any), then executing the function’s sequential command, then evaluating the ‘`return`’ expression to the value v', then destroying the formal parameter and any other local variables. In either case, the value of the function call is v'.

6 Lexicon

Syntax

num	=	*digit* +	– numeral
ident	=	*letter* (*letter* | *digit*) *	– identifier
space	=	(‘ ’ | ‘\t’) +	– white space
eol	=	‘\r’ ? ‘\n’	– end-of-line
comment	=	‘#’ *comment-char* * ‘\r’ ? ‘\n’	
comment-char	=	…	– any character other – than ‘\r’ or ‘\n’
digit	=	‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’	
letter	=	‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’ | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’	

Spaces, ends-of-lines, and comments do not influence the program’s phrase structure. They are there for human readers only.

7 Predefined

Fun has two predefined procedures and functions:

```
func int read ():      # inputs and returns an integer
    …
proc write (int n):    # outputs the integer n
    …
```