Formal Languages and Compilers - Exercises
Lecture 9
Subprograms in Crème CAraMeL

17/04/2012

Outline

Assumptions and simplifications

- Let's add
  - declarations of subprograms (procedure)
  - execution of subprograms (call and passing the parameters)
- no declarations inside the `begin...end`
- declarations are non-static: dynamic local environment (DLE)
- passing the parameters only *by value*

Assumptions and simplifications

- Let's add
  - declarations of subprograms (procedure)
  - execution of subprograms (call and passing the parameters)
- no declarations inside the `begin...end`
- declarations are non-static: dynamic local environment (DLE)
- passing the parameters only *by value*

Assumptions and simplifications

- Let's add
  - declarations of subprograms (procedure)
  - execution of subprograms (call and passing the parameters)
- no declarations inside the `begin...end`
- declarations are non-static: dynamic local environment (DLE)
- passing the parameters only *by value*

Assumptions and simplifications

- Let's add
  - declarations of subprograms (procedure)
  - execution of subprograms (call and passing the parameters)
- no declarations inside the `begin...end`
- declarations are non-static: dynamic local environment (DLE)
- passing the parameters only *by value*

Outline

Procedure: example

```
program
    var x : int

    procedure proc1(a: int)
    begin
        write(a)
    end;

    procedure proc()
        var x : int
    begin
        x := 5;
        call proc1(x);
        write(x)
    end

begin
    x := 40;
    write(x);
    call proc();
    call proc1(x)
end
```

Output

```
40
5
5
40
```

Procedure: implementation

Syntax

parser.mly: new token PROCEDURE, CALL, COMMA (“,”)

lexer.mll: strings corresponding to token

syntaxtree.ml: constructors for

- declarations
- formal parameters
- calls
- actual parameters
- other modifications

parser.mly: productions to construct new nodes

Procedure: implementation

Syntax

parser.mly: new token PROCEDURE, CALL, COMMA (“,”)

lexer.mll: strings corresponding to token

syntaxtree.ml: constructors for

- declarations
- formal parameters
- calls
- actual parameters
- other modifications

parser.mly: productions to construct new nodes

Procedure: implementation

Syntax

parser.mly: new token PROCEDURE, CALL, COMMA (",")

lexer.mll: strings corresponding to token

syntaxtree.ml: constructors for

- declarations
- formal parameters
- calls
- actual parameters
- other modifications

parser.mly: productions to construct new nodes

Procedure: implementation

Syntax

parser.mly: new token `PROCEDURE`, `CALL`, `COMMA` (“,”)

lexer.mll: strings corresponding to token

syntaxtree.ml: constructors for

- declarations
- formal parameters
- calls
- actual parameters
- other modifications

parser.mly: productions to construct new nodes

Procedure: implementation

Semantics

- new value in the environment:

  Descr_Procedure **of** param list $*$ dec list $*$ cmd
- declaration
- execution (call), which computes:
  - evaluation of actual parameters
  - type checking for the list of parameters
  - actual execution of the procedure

Procedure: implementation

Semantics

- new value in the environment:
  Descr_Procedure **of** param list $*$ dec list $*$ cmd
- declaration
- execution (call), which computes:
  - evaluation of actual parameters
  - type checking for the list of parameters
  - actual execution of the procedure

Procedure: implementation

Semantics

- new value in the environment:

  Descr_Procedure **of** param list ∗ dec list ∗ cmd
- declaration
- execution (call), which computes:
  - evaluation of actual parameters
  - type checking for the list of parameters
  - actual execution of the procedure

Procedure: implementation

Semantics

- new value in the environment:

  Descr_Procedure **of** param list $*$ dec list $*$ cmd
- declaration
- execution (call), which computes:
  - evaluation of actual parameters
  - type checking for the list of parameters
  - actual execution of the procedure

Procedure: implementation

Semantics

- new value in the environment:

  Descr_Procedure **of** param list $*$ dec list $*$ cmd

- declaration
- execution (call), which computes:
  - evaluation of actual parameters
  - type checking for the list of parameters
  - actual execution of the procedure

Outline

Function: example

```
program
  var x : int

  function fact(a: int): int
    var b : int
  begin
    if (a = 0) then
        fact := 1
    else begin
        b := call fact(a − 1);
        fact := a ∗ b
    end
  end

  begin
    x := call fact(12);
    write(x)
  end
```

Output

```
479001600
```

Function: implementation

Syntax

- Keyword: `function`
- New nodes for: declaration, execution (call), evaluation (call!)
- Adjust the syntax tree

Semantics

- Declaration (alert: a location for return value is needed!)
- Evaluation
- Execution

Function: implementation

Syntax

- Keyword: `function`
- New nodes for: declaration, execution (call), evaluation (call!)
- Adjust the syntax tree

Semantics

- Declaration (alert: a location for return value is needed!)
- Evaluation
- Execution

Function: implementation

Syntax

- Keyword: `function`
- New nodes for: declaration, execution (call), evaluation (call!)
- Adjust the syntax tree

Semantics

- Declaration (alert: a location for return value is needed!)
- Evaluation
- Execution

Function: implementation

Syntax

- Keyword: `function`
- New nodes for: declaration, execution (call), evaluation (call!)
- Adjust the syntax tree

Semantics

- Declaration (alert: a location for return value is needed!)
- Evaluation
- Execution

Function: implementation

Syntax

- Keyword: `function`
- New nodes for: declaration, execution (call), evaluation (call!)
- Adjust the syntax tree

Semantics

- Declaration (alert: a location for return value is needed!)
- Evaluation
- Execution

Projects

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Projects for 1-person groups (one OR the other)

- Pointers and dynamic memory management
- Pointers and record
- Array implemented by linked lists

Projects for 2-people groups (again, only one)

- Multidimensional matrices and slices
- Pointers and different ways of passing the parameters
- Tuple type and multiple return variables

Projects for 3-people groups (and again)

- Record, pointers, multidimensional matrices and slices
- Multiple return values and passing parameters

Pointers

Declaration

```
var p : ^int;
var q: ^^float;
```

Referencing(@) and dereferencing (^)

```
x := 1;
p := @x;
y := ^p + 4;
```

If x and y are integers, then in the end $y = 5$.

Dynamic memory

Add to the language the possibility to use pointers and allocation/deallocation of dynamic memory (heap), using one of the following approaches of memory release:

- Reference counter
- Garbage collection

A correct implementation will allow to create and use the dynamic data structures using pointers in `Crème CAraMeL`.

Vectors "by linked list"

Change the implementation of vectors in `Crème CAraMeL` in a way that the following operations are possible:

`v(i) := 5`	inserting an element (growing the length of the vector)
`v[i] := 5`	substitution of an element (the length remains the same)
`v#i`	deleting an element (the vector becomes shorter)
`v?5`	returns an integer `i` if vector contains value 5 at position `i` and an integer -1 if there is no value 5 in the vector

Record

Definition

```
type name_record = record {
    name_field_1 : type;
    ...
    name_field_n : type;
}
```

Declaration

```
var v : name_record;
```

Access

```
v.name_fieldi := expression;
a := v.name_fieldi;
```