02257 Applied functional programming

Michael R. Hansen
DTU COMPUTE
January 8, 2020

Project 2:
A compiler for an imperative programming language

In this project you shall make a compiler for an imperative programming language. To solve this exercise you should extend a "minimal" solution that is available on Inside. The different parts of the project will be introduced in lectures covering:

- Scanning and Parsing
- Type check
- Symbolic code and compilation: declarations, expressions and statements
- Compilation: arrays
- Compilation: functions and procedures

The project can be solved in groups having two, three or four persons. Single-person groups must be approved by me.

To submit the solution to this project, you must:

1. give a demo on Monday, January 20, from 13:00-16:00, and
2. submit your project on Inside no later than 17:00 on Monday, January 20.

Note that: **All group members must be available at the demo**.

1 Background

You shall develop a compiler for an imperative programming language in the functional programming language F#. The methods, tools and techniques for doing so are described in:

PLC: *Programming Language Concepts*, Peter Sestoft, Springer 2012.

This book has a supporting website `https://www.itu.dk/people/sestoft/plc/` containing the programs from the book and slides from a course using this book. In particular, the Chapters 3, 7, 8 and 12 are relevant for this project.

The language for which you should develop a compiler has many similarities to micro-C and ideas and even code fragment from the micro-C compiler (see Chapter 8 of PLC) can be used in your project. It is ok to reuse code and ideas from the micro-C compiler; but when doing so, you must state this explicitly in comments in your code.

The language for which you should develop a compiler is a variant of micro-C that is based on a version of *guarded commands*, introduced by E.W. Dijkstra:

EWD472: *Guarded commands, non-determinancy and formal derivation of programs*, Edskar W. Dijkstra, Comm. ACM 18 (1975), 8:453–457. This article can be extracted from `https://en.wikipedia.org/wiki/Guarded_Command_Language`.

Guarded commands as used in this project are introduced in the Section 3.6.

2 The handout

Further information on the imperative programming language is given in lectures and the minimal solution that is available from Inside. This solution contains the following files:

- `AST.fs`: Defines types for the abstract syntax.
- `Parser.fsy`: Input file to `fsyacc`.
- `Lexer.fsl`: Input file to `fslex`.
- A number of files generated by the `fslex` and `fsyacc` tools.
- `TypeCheck.fs`: Defines type-checking functions for expressions, statements, etc.
- `CodeGen.fs`: Defines compilation functions for expressions, statements, etc.
- `CodeGenOpt.fs`: Defines optimized compilation functions for expressions, statements, etc.
- `Util.fs`: Contains functions to parse, type check, compile and execute programs.
- `Script.fsx`: Defines a number of test cases.
- A number of files with sample programs. These files has extension `.gc`.
- `README.txt`.

3 The programming language: the basic version

The ingredients of the basic version are: basic types and constants, variables, declarations, expressions, statements and programs:

3.1 Programs

The basic structure of a program is:

```
begin
   $dec_1$, ..., $dec_m$ ;
   $s_1$; ...; $s_n$
end
```

that is, a list of m declarations separated by comma ',' followed by a list of n statements separated by semicolon ';', where $m \geq 0$ and $n \geq 0$. Semicolon is also used to separate the declaration and statement lists. In the following sections we shall described declarations and statements in details.

3.2 Types and constants

There are two *basic types*: `int` and `bool` for the types of integers and Booleans, respectively. Examples of *constants* of type `int` are `-12033`, `0` and `12344565`. The constants `true` and `false` are the only constants of type `bool`.

3.3 Identifiers, keywords, variables and declarations

An *identifier* is a string starting with letter that may be followed by zero or more letters and digits. Examples of identifiers are `x`, `X`, `xyz`, `x1234`, `do` and `begin`.

The *keywords* or *reserved words* of the language (including later extension) are:

```
abort begin bool do end false fi function if int od procedure return skip true
```

Identifiers that are not keywords can be used as *variables* (and *function and procedure names* in later extensions). Examples of *simple declarations* are `x:int` and `flag : bool`.

A *declaration list* is a non-empty sequence of simple declarations separated by comma '`,`'. For example: `x : int , y: int, flag :bool`. If there are multiple declarations of a variable, say x, then the last declaration for x will override previous ones.

3.4 Expressions

There are two kinds of expressions in the language: *arithmetical expressions* and *Boolean expressions.*

Arithmetical expressions are formed from integer values and variables of type `int` by use of arithmetical operations like `+`, `-` and `*`.

Expressions obtained by comparing two arithmetical expressions, like $e_1 \texttt{=} e_2$, $e_1 \texttt{<=} e_2$, $e_1 \texttt{<>} e_2$ (inequality) and $e_1 \texttt{<} e_2$, are Boolean expression. Boolean expressions are formed from comparisons of arithmetical expressions, Boolean constants and variables of type `bool` by use logical operations, like negation `!`, conjunction `&&` and disjunction `||`. Furthermore, Boolean expressions can be compared using equality $b_1 \texttt{=} b_2$ and inequality $b_1 \texttt{<>} b_2$ thereby forming new Boolean expressions.

Conjunction has a *short circuit semantics*, that is, an expression $b_1 \texttt{\&\&} b_2$ is false if the value of b_1 is false (and in this case b_2 is not evaluated at all). If b_1 is true then the value of the conjunction is the truth value of b_2. Disjunction has a short-circuit semantics as well.

3.5 Statements

Statements are constructed from *primitive statements* by use of sequential composition and Dijskstra's *guarded commands.*

3.5.1 The primitive statements

Assignment: $x{:=}e$, where x is a variable and e is an expression. The value of the expression e is computed and assigned to the variable x.

The print statement: `print` e, which prints the value of the expression e on the console. If e is a Boolean expression, the `true` is printed as 1 and and `false` as 0.

A variable x is used in two contexts. One is in an expression, like $x + 2$, where it denotes contents of the location for x in the storage. This is also called the *rvalue* (right-hand-side value) of x. Another context is in the left-hand side of an assignment like $x{:=}33$. This occurrence of x denotes the storage location, also called the *lvalue* (left-hand-side value) of x.

Remark: The original assignment statements for the guarded command language by E.W. Dijkstra were simultaneous assignments of the form $x_1, x_2, \ldots, x_n$:=$e_1, e_2, \ldots, e_n$, where the expressions $e_1, e_2, \ldots, e_n$ are evaluated in the initial state and their values are simultaneous assigned to the variables $x_1, x_2, \ldots, x_n$.

3.5.2 Sequential composition of statements

Statements can be composed in a list as follows: s_1;s_2;...;s_n, $n \geq 0$. Note that semicolon ';' is only placed between statements. A sequential composition statement is executed starting with s_1. If s_1 terminates, then s_2 is executed, and so on. If $n = 0$ the construct terminates without causing any effect. This (lack of) effect is also obtained by the statement `skip`.

3.5.3 Blocks

Parentheses are used to group statements: $\{sl\}$, where $sl = s_1$;s_2;...;s_n. This statement is called a *block*. Blocks are also used when introducing local declarations in Section 4.2.

3.6 Guarded commands

There are two kinds of guarded commands: *alternative* and *repetition.*

An alternative command has the form:

```
if b₁ -> sl₁
 |  ...
 | bₙ -> slₙ
fi
```

comprising n guarded statements of the form b_i->sl_i, where b_i is a Boolean expression also called a *guard* and sl_i is a statement list. The guarded statements are separated by a bar '|'.

An alternative command is executed as follows: The guards are evaluated starting with b_1, then with b_2 and so on. As soon as a true guard b_i is found, the corresponding statement sl_i is chosen for execution after which the alternative command terminates. If all guards evaluates to false, the alternative command terminates abnormally (by terminating the entire program).

A alternative command having no guarded alternatives, i.e. $n = 0$, terminates the entire program abnormally. This command is also called '`abort`'.

An example program with an alternative command is `Ex3.gc` from the handout:

```
begin
  x : int;
  x := 4;
  if  x=3 -> print 3
   |  x=4 -> print 4
   |  x=5 -> x:=5
  fi
end
```

This program will print 4 and then terminate.

A repetition command has the form:

```
do $b_1$ -> $sl_1$
 |  ...
 | $b_n$ -> $sl_n$
od
```

A repetition command is executed as follows: The guards are evaluated starting with b_1, then with b_2 and so on. If all guards are false, then the repetition command terminates. Otherwise, as soon as a true guard b_i is found, the corresponding statement list sl_i is chosen for execution after which the repetition command is executed again.

A repetition command having no guarded alternatives, i.e. $n = 0$, terminates immediately without having any effect. This command is also called '`skip`'.

An example program with an repetition command is `Ex4.gc` from the handout:

```
begin
  x : int;
  x := 1;
  do  x=1 -> print x; x:= x+1
   |  x=2 -> print x; x:= x+1
   |  x=3 -> print x; x:= x+1
  od;
  print x
end
```

This program will print the four numbers 1, 2, 3 and 4 on separate lines and then terminate.

Remark: The original semantics for guarded commands as given by E.W. Dijkstra was non-deterministic. For example, for an alternative command, a statement list sl_i having a true guard b_i is chosen non-deterministically.

3.7 Tasks

As a first step you should extend the minimal project handed out to obtain a compiler for the basic language described above. This involves, amongst other things:

- Extension of the type checker. This involves checking that variables are declared before they are used, typing of Boolean and arithmetic expressions and checking that statements are well-typed.

- Extension of the compiler. The major task here is to generate symbolic code for guarded commands.

- Test that the compiler works correctly on the programs `Ex1.gc` - `Ex6.gc` and `Skip.gc`. These programs can be parsed using the minimal solution.

- Test the compiler on a suitable set of example programs, including programs that should be rejected because they are ill-typed.

4 Adding functions and local declarations

In this first extension you should add functions and local declarations. These extensions can be developed and tested separately.

4.1 Adding functions

To capture functions in the language: you should add *functions declarations*, *function application* and the `return` statement.

The `return` statement

A return statement has the form:

`return` e

where e is an expression. Execution of this statement in a functions's body will terminate a function application with the value of the expression e. A return statement is well-typed if e is a well-typed expression with type τ and the return statement occurs in the body of a function declaration (see below), where the return type of the function is τ.

Function declarations

A function declaration has the form:

`function` f`(`$x_1 : \tau_1$`, ..., `$x_n : \tau_n$ `) :` τ `=` s

where f is the name of the function being declared, $x_1 : \tau_1, \ldots, x_n : \tau_n$ is a specification of the formal parameters, that is, x_i is the i'th formal parameter with the type τ_i, τ is the return type of the function, and s is the statement constituting the body of the function. The parameter passing mechanism is call-by-value.

A function declaration is well-typed, if

- the formal parameters $x_1, \ldots, x_n$ are all different,
- for every return statement `return` e occurring in s, expression e has type τ, and
- the statement s is well-typed.

The type environment used in the type check of the body of f is obtained by extending current type environment with types for formal parameters and for f (to allow recursive declarations).

Function application

A function application is an expression with the form:

f`(`e_1`, ..., `e_n`)`

where f is a function name and $e_1, \ldots, e_n$ is the list of actual parameters.

A function application is well-typed having type τ, if the types of the actual parameters matches the types of the formal parameters for f and τ is the return type for f.

An example program with a function declaration, function application and a return statement is `Ex7.gc` from the handout:

```
begin
  x: int,
  function f(x: int): int = { print x; return x+1 };
  x:= f(2);
  print x
end
```

This program will print the two numbers 2 and 3 on separate lines and then terminate.

An example of a declaration of a recursive function is `factRec.gc` from the handout:

```
begin
  res : int,
  function fact(n: int): int = if n=0  -> return 1
                                | true -> return n * fact(n-1)
                               fi;
  res := fact(4);
  print res
end
```

This program will print the number 24 and then terminate.

4.2 Local declarations

A local-declaration statement has the form:

$$\begin{array}{l}\{\ x_1 : \tau_1,\ \ \ldots,\ \ x_m : \tau_m\ \ ; \\ \quad s_1;\ \ \ldots\ \ ;\ \ s_n\ \ \}\end{array}$$

where $x_1 : \tau_1, \ldots, x_m : \tau_m$ is a list of declarations separated by comma ',' and $s_1; \ldots ; s_n$ is the list of statements separated by semicolon ';'. Semicolon is also used to separate the declaration and statement lists.

Note that local function declarations are not possible. The same restriction is imposed in the micro-C language. Another restriction in the micro-C language is that local declarations are only allowed in bodies of function declarations.

An example program with declarations is `factCBV.gc` from the handout:

```
begin
  res : int, x: int,
  function fact(n: int): int =
  {  y:int ;
     y := 1 ;
     do ! n = 0  ->  y := n*y; n:=n-1 od ;
     return y } ;
  x := 4;
  res := fact(x);
  print x;
  print res
end
```

This program will print the two numbers 4 and 24 on separate lines and then terminate.

4.3 The tasks

Extend your compiler so that it can compile and type check the features described above. You may consider an extension allowing local declarations also at the top-most level. Test that the compiler works correctly on the programs `Ex7.gc`, `fact.gc`, `factRec.gc` and `factCBV.gc`. Include also suitable tests of you own.

5 Adding arrays

You shall now add arrays to the language in a fashion similar to that of micro-C.

5.1 Array declarations

A declaration of an array a containing i elements of type τ has the form:

$a : \tau[i]$

where $i \geq 0$ and τ is either `int` or `bool`. That is, arrays of arrays of ... are not allowed. (The same restriction is imposed in micro-C.)

5.2 Array indexing

An array-index expression has the form: $a[e]$, where a is an array and e is an arithmetical expression.

If the value of e is i, then the expression denotes the i'th element of the array, that is, the rvalue of $a[e]$. If $a[e]$ occurs in an assignment such as $a[e]{:=}e_1$, then $a[e]$ denotes the location of the i'th element of a, that is, the lvalue of $a[e]$.

5.3 Specification of formal array parameters

A specification of a formal array parameter a to a function has the form:

$a : \tau[]$

where τ is either `int` or `bool`. Note that the length of the array is not specified.

Parameter passing of arrays is as for micro-C, that is, it is based on call by value, where the value of a is a reference to the first element (with index 0) of the array.

An example program with arrays is `A0.gc` from the handout:

```
begin
  a : int[3],
  function f(): int = { b: int[2]; b[1] := 7; return b[1] },
  function g(c: int[]): int = { c[0] := 25; print c[0]; return -1 };
  print 3;
  a[0] := 2;
  print a[0];
  print f();
  print g(a);
  print a[0]
end
```

This program prints the numbers 3, 2, 7 , 25 , -1 and 25 on separate lines and then terminate.

5.4 The tasks

Extend your compiler for the basic language so that it can compile and type check the features described above. Test that the compiler works correctly on the programs `A0.gc`, `A1.gc`, `A2.gc` and `A3.gc`. Include also suitable tests of you own.

6 Adding multiple assignments

The guarded commands language introduced by Dijkstra includes multiple assignments of the form:

$$v_1, v_2, \ldots, v_n := e_1, e_2, \ldots, v_n$$

where e_i is and expression and v_i is a variable of the form x_i or the form $a_i[e'_i]$, allowing assignments to both simple variables and array locations. To be well-typed e_i and v_i must have the same types.

The semantics is as follows:

1. The expressions e_i are evaluated from left to right producing n values $val_1, \ldots, val_n$.
2. The left-hand side variables v_i are evaluated from left to right producing n locations $loc_1, \ldots, loc_n$.
3. val_i is assigned to loc_i, for $1 \leq i \leq n$.

Note that the evaluation order is important due to side-effects.

6.1 The tasks

Extend your compiler so that it can compile and type check multiple assignments. Develop tests showing that the compiler works correctly.

7 Adding procedures

You shall now add procedures to the language. To capture procedures you should add *procedure declarations* and a statement for *procedure calls*.

Procedure declarations

A procedure declaration has the form:

`procedure` p`(`$x_1 : \tau_1$`, ..., `$x_n : \tau_n$` ) = `s

where p is the name of the procedure being declared, $x_1 : \tau_1, \ldots, x_n : \tau_n$ is a specification of the formal parameters, that is, x_i is the i'th formal parameter with the type τ_i, and s is the statement constituting the body of the procedure. The parameter passing mechanism is call-by-value.

A procedure declaration is well-typed, if

- the formal parameters $x_1, \ldots, x_n$ are all different, and

- the statement s is well-typed.

The type environment used in the type check of body is obtained by extending the current type environment with types for formal parameters and for p (to allow recursive declarations).

Procedure call

A procedure call is an statement with the form:

$p(e_1,\ \dots,\ e_n)$

where p is a procedure name and $e_1, \dots, e_n$ is the list of actual parameters. A procedure call is well-typed, if the types of the actual parameters matches the types of the formal parameters.

An example program with procedures is `Swap.gc` from the handout:

```
begin
   procedure swap(a: int[], i: int, j: int) =
   { tmp: int; tmp := a[i]; a[i] := a[j]; a[j] := tmp},
   procedure printA(a: int[], len:int) =
   { i: int;  i:= 0; do ! i=len -> print a[i]; i:= i+1 od},
   a: int[3];

   a[0]:= 0; a[1]:=1; a[2]:=2;
   printA(a,3);
   swap(a,0,1);
   printA(a,3)
end
```

The procedure `swap` is used to exchange two elements of an array and `printA` to print the elements of an array. A recursive procedure declaration is found in `QuickSortV1.gc`.

7.1 The tasks

Extend your compiler so that it can compile and type check the features described above. Test that the compiler works correctly on the programs `A4.gc`, `Swap.gc` and `QuickSortV1.gc`. Include also suitable tests of you own.

8 Adding pointers

You shall now add pointer types to the language, together with two operators: the *address-of* operator and an operator for *dereferencing a pointer*. This shall be added in a fashion similar to that for micro-C, but we shall used a different notation, inspired by Modula 2.

8.1 Pointer types and declarations

If τ is a type, then $\hat{}\tau$ is a pointer type. A variable x, declared to have a pointer type as follows $x : \hat{}\tau$, must refer to a location containing a value of type τ.

8.2 The address-of operator &

The pointer that refers to x is obtained using the address-of operator & as follows &x.

8.3 The dereference operator ^

If variable x has a pointer type, then the contents of the location pointed to by x (that is, its rvalue) is obtained using the dereference operator ^ as follows x^.

Used at the left-hand side of an assignment, such as x^:=e, then x^ denotes the location pointed to by x (that is, the lvalue of x^).

An example illustrating these features is `Par1.gc` from the handout:

```
begin
  x : int,
  procedure p(y: ^int) = { print y^; y^ := 1 };
  x:= 0;
  p(&x);
  print x
end
```

This program prints 0 and 1 on separate lines and terminates. The formal parameter `y: ^int` of procedure `p` is a *result parameter*, that is, a pointer to where the result should be put.

8.4 The tasks

Extend your compiler so that it can compile and type check the features described above. Test that the compiler works correctly on the programs `Par1.gc`, `QuickSortV2.gc` and `factImpPTyp.gc`. Include also suitable tests of you own. Try also to test using `Par2.gc`.

9 Suggestions for further extensions/considerations

There are many ways to test, extend and improve the language/compiler developed as described above. This section is intended as an inspiration for further extensions.

9.1 Assessment of the generated code

Make an assessment of the code generated by your compiler. For example, will you compiler frequently generate code having jump to jumps, addition of zero, etc.?

You may, for example, have a look at the analyses of code for the unoptimized programs in Section 12.3.4 of PLC. Reformulate these programs in your language and analyse the code generated by your compiler.

9.2 Develop an optimizing compiler

In Chapter 12 of PLC there is description of an optimizing compiler for micro-C based on backwards generation of the code. The corresponding compiler for micro-C is found in the file `Contcomp.fs`. The same idea can be used for your guarded-command-based language.

The file `CodeGenOpt.fs` contains the basic infrastructure from `Contcomp.fs` and exploits this infrastructure on the minimal language. Extend `CodeGenOpt.fs` to cover your language and assess the generated code.

9.3 More interesting examples

You can try to to develop more programs to test your compiler. For example, `ex11.c` from PLC is a micro-C program for finding all solutions to the n-queens problem. You may try to reformulate it in your language and test it.

9.4 More language features

In exercises in Chapter 8 of PLC there are several suggestions for language extensions. You may try to incorporate some of them.

10 The assignment

You should hand in two files on Inside as a group:

- A zip-archive containing your solution.
- A pdf-file containing at most 3 pages including the front page:
  1. A front page with names and study numbers of the persons in the group. Handing in as a group, the members account for (1) the extension from the minimal solution is made by the group members only, (2) no part of their solution is distributed to other groups, and (3) the group members have contributed equally to the solution. If the group members did not contribute equally to the solution, then that must be stated on the front page.
  2. The following pages should describe status and highlights of the solution. It must be stated which programs from the handout the solution can handle. It should also describe further extensions. It must be possible to verify the status by executing the `Script.fsx` file. There should be a brief reflections over the project. If you have spotted natural generalizations, changes or extensions of the project, then describe/discuss them in the reflection section.