CSE 302: Compiler Design — Lab 4

Procedures and Functions

Out: `2019-10-14 08:00:00 CEST`
Tests due: `2019-10-21 10:00:00 CEST`
Compilers due: `2019-11-04 10:00:00 CET`

1 INTRODUCTION

In this lab you will update compiler you built in earlier labs to the language `BX2`, which is an extension of `BX1` with support for procedures and functions.

- **Tests**: by `2019-10-21`, we would like you to submit 5 *non-trivial* test programs written in the `BX2` language. These programs will be made available to everyone during the second week.
- This lab is intended to be worked **in pairs**.

This lab will be assessed. It is worth 15% (i.e., 3 points) of your final grade. To get full credit, you must submit the final version of your compiler strictly before `2019-11-04 10:00:00 CET`.

2 THE BX2 LANGUAGE

The complete grammar of `BX2` can be found in `bx2_grammar.pdf`.

2.1 *Overview*

The `BX2` language is a strict superset of the `BX1` language that adds functions and procedures to the language. Any valid `BX1` program continues to be a valid `BX2` program.

Procedures and Functions The main new feature of `BX2` is the ability to define *procedures* and *functions*, which are collectively known as *callables*. The difference between procedures and functions is that procedures do not return a value while functions always do. All callables are implicitly mutually recursive with every other callable defined in the same `BX2` file. This means that any callable can *invoke* any other callable, including themselves and those callables defined later. The syntax of procedures and functions is covered in more detail in section 2.2.

Explicit Main Procedure A `BX2` program has an explicit procedure called `main()` that begins the execution of the program. This procedure is exactly analogous to the `main()` function found in all C-like languages. The `main()` procedure *must be present* in order for a `BX2` program to be well-formed.

Declarations Anywhere `BX2` relaxes the restriction of `BX1` that all variables must be declared up front. Instead, variable declarations and statements can now be freely intermixed. Variable declarations retain their restrictions: no variable can be declared more than once (but see *Scope* below) and their optional initializers are only allowed to refer to variables declared earlier.

Variables may also be defined outside any procedures. These variables are *global variables* and may only be initialized using constants.

Scope Because variable declarations become just another kind of statement, BX2 also generalizes the notion of blocks from BX1 to *scopes*. A variable declared in a block lasts until the end of the block, after which it may no longer be referenced. Within one block a variable may not *shadow* another variable, that is, a variable that is declared earlier in the same scope. However, a variable name becomes available again after the block in which it is declared is closed, i.e., after it goes out of scope.[1]

2.2 *Procedures and Functions*

Procedures A ⟨procedure⟩ begins with the keyword proc followed by the name of the procedure and its list of formal parameters. This is followed by a ⟨block⟩ (scope) that contains the body of the procedure.

```
⟨procedure⟩ ::= "proc" ⟨variable⟩ "(" ⟨parameter-groups⟩? ")" ⟨block⟩

⟨parameter-groups⟩ ::= ⟨parameter-group⟩ ("," ⟨parameter-groups⟩)?
⟨parameter-group⟩ ::= ⟨variable⟩ ("," ⟨variable⟩) * ":" ⟨type⟩
```

The name of the procedure must be globally unique, i.e., there must not be any other global variable, function, or procedure with that same name. The parameters of the procedure must have pairwise distinct names. The division of parameters into groups is purely syntactic sugar; there is no difference between the following two definitions.

```
proc p(x1, x2 : int64) { ... }
proc p(x1 : int64, x2 : int64) { ... }
```

The order of parameters, however, is important.

Functions A ⟨function⟩ is like a ⟨procedure⟩, except it also returns a result of a specified type.

```
⟨function⟩ ::= "fun" ⟨variable⟩ "(" ⟨parameter-groups⟩? ")" ":" ⟨type⟩ ⟨block⟩
```

Once again, the name of a function must be globally unique.

Calling Both functions and procedures may be *called* (aka *invoked*) by means of their name and a list of arguments whose types match that of the formal parameters of the callable. When calling a procedure, the call has no value and can therefore not be used as part of an expression. When calling a function, the call produces a value of the return type of the function, which can then be used as part of an expression.

Procedure calls and function calls have the same syntax. We relax the grammar slightly and allow every ⟨expr⟩ to be used as a ⟨stmt⟩. It will then be up to the type checker to guarantee that procedure calls are not being used as subexpressions of an ⟨expr⟩.

```
⟨expr⟩ ::= ··· | ⟨call⟩

⟨call⟩ ::= ⟨variable⟩ "(" (⟨expr⟩ ("," ⟨expr⟩) *) ? ")"

⟨stmt⟩ ::= ··· | ⟨expr⟩ ";"
```

[1]When a variable goes out of scope, it does not necessarily mean that the space allocated to that variable in the stack is freed or reused by other variables. That is an optional optimization that you should only attempt to implement when you've got everything else in this lab working.

Returning   The `return` keyword is used to terminate a call. If the call was to a procedure, then the argument to `return` is either absent or is a call to another procedure. For functions, the argument to `return` can be any expression of the right type, which includes calls to other functions. In either case, `return` is always a ⟨stmt⟩.

> ⟨stmt⟩ ::= · · · | ⟨return⟩
>
> ⟨return⟩ ::= "return" ⟨expr⟩? ";"

As with calls, it is the job of the type-checker to make sure that a `return` from a procedure does not have an expression argument, and that a `return` from a function has an expression argument of the right type.

2.3 *Variable Declarations and Scopes*

Unlike BX1, there is no longer a distinction between variable declarations (⟨vardecl⟩) and statements (⟨stmt⟩). Indeed, we explicitly include the former into the latter.

> ⟨stmt⟩ ::= · · · | ⟨vardecl⟩

A new variable may be declared at any point in the program, not just in the beginning. As before, a variable may be declared with an initial value, which must be an expression formed out of variables declared earlier. A variable being declared must also not already be declared in the same scope.

Shadowing   Every declared variable is required to be unique in its own scope, but an inner scope can declare the same variable (not necessarily with the same type). The inner declaration is said to *shadow* the outer declaration. Within the inner scope the variable name now refers to the inner declaration, while outside the scope the outer declaration continues to exist. Some possibilities:

```
proc f() {
  var x = 10 : int64;
  {
    print x;                    // prints "10"
    x = x + 20;
    var x = false : bool;       // shadows the outer x
    print x;                    // prints "false"
    // var x = 42 : int64;      // illegal: redeclaration of x in same scope
    {
      var x = 42 : int64;       // OK in deeper scope
    }
  }
  print x;                      // prints "30"
  {
    var x = x + 10 : int64;     // in (x + 10), x refers to outer scope
    print x;                    // prints "40" because x is now declared inside
  }
  print x;                      // prints "30"; outer x was not modified
}
```

Global Variables   Global variables are those variables that are declared outside any callable. These variables may be initialized to constants.

⟨globalvar⟩ ::= "var" ⟨globalvar-init⟩ ("," ⟨globalvar-init⟩)* ":" ⟨type⟩ ";"

⟨globalvar-init⟩ ::= ⟨variable⟩ ("=" (⟨number⟩ | ⟨bool⟩))?

BX2 Programs A BX2 program is a sequence of ⟨procedure⟩, ⟨function⟩, and ⟨globalvar⟩ declarations. The ordering of these declarations is immaterial. In order for the BX2 program to be well formed, exactly one of the declarations should be the declaration of the `main()` procedure that looks as follows:

```
proc main() {
  // ...
}
```

⟨program⟩ ::= (⟨globalvar⟩ | ⟨procedure⟩ | ⟨function⟩)*

2.4 *Type-Checking*

To make sure that a syntactically well-formed BX2 program makes sense, we will *type-check* the program. Because BX2 callables are all allowed to call each other, to type-check a BX2 program we have to proceed in two phases; in each phase we will traverse the entire program.

- Phase 1: collect all the *type signatures* for the callables
- Phase 2: type-check every expression and statement with respect to these type signatures.

Phase 1 This first phase is fairly straightforward. All it does is make sure that each callable name is unique, and then constructs a map from the name of the callable to its parameter types (and optional return type if it is a function). During this phase, the rest of the AST remains unchecked.

Phase 2 During the second phase, the types of all the callables is already assumed. With this information, the rest of the program is traversed to check the following aspects of a well-typed BX2 program:

- Every variable is declared at most once per scope and its initial value (if present) can be computed at the point of its declaration based on the variables already defined.
- In a procedure body, the occurrences of `return` have either no arguments or the argument is another procedure call.
- In a function body, every occurrence of `return` has exactly one argument which is an expression of the same type as the return type of the function.
- In a function body, every code path ends with a `return`. In particular, there must always be a `return` following a `while`.

2.5 *Week 1 Tasks*

1. Update the grammar and parser from BX1 to BX2
2. Write the type-checker for BX2 as described above
3. Construct 5 interesting test cases that successfully pass your type-checker
4. Optionally: submit some type-incorrect test cases

To help you with these tasks, we provide a reference interpreter for BX2.

```
/users/profs/info/kaustuv.chaudhuri/CSE302/BX2/interpret.exe
```

Run it as usual with a BX2 source file as its sole argument. For example:

```
$ /users/profs/info/kaustuv.chaudhuri/CSE302/BX2/interpret.exe fibs.bx
0
1
1
2
3
⋮
```

3 COMPILING BX2

To come:

1. Declaration hoisting
2. RTL with explicit frames (ERTL)
3. Global variables
4. Position-independent code
5. Extra credit: tail-call elimination for compatible stack frames