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 the 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, but is *not* backwards-compatible. That is, most `BX1` programs will not syntactically be valid `BX2` programs.

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. Global variables do not have a fixed ordering, and functions and procedures are allowed to refer to any global variable, even those variables that are declared after the callable.

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⟩.

[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.

⟨expr⟩ ::= ··· | ⟨call⟩

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

⟨stmt⟩ ::= ··· | ⟨expr⟩ ";"

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 are shown in figure 1.

Global Variables Global variables are those variables that are declared outside any callable. These variables must be initialized to constants. The name of a global variable must also be unique.

⟨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() {
  // ...
}
```

```
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
}
```

Figure 1: Example of scopes and shadowing

⟨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 and to refer to any global variable, 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 and global variables.
- 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 and global variable name is unique, and then constructs a map from the name of the variable or 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 global variables and 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 (WEEK 2)

3.1 *Simplification*

Before starting to work on the backend, write a *simplification* pass that removes some features from the typed AST of BX2:

- Move all variable declarations to the top of the function or procedure by following the unique numbering and hoisting mechanism that you saw in class
- Remove nested scopes wherever you can. The scoping braces `{ }` should now only be found at the outermost level of a callable body, and in the then and else blocks of an `if` or the body of a `while`.

3.2 *Global Variables and PC-Relative Addressing*

Global variables need to be placed in the `.data` section of the assembly file. They continue to be globally visible names, so they must be maked `.globl`. The variable name itself will be a label in the assembly file, which like all labels will be a pointer to a part of memory. Following the label, you must declare the bytes that constitute the initial value of the variable. For now we will use 64-bit slots for both `int64` and `bool` variables, whose bytes are declared with the `.quad` declaration (standing for "quad word", where a "word" is two bytes). Figure 2 shows the BX2 code for two global variable declarations and the correspnding assembly file.

To read and write a global variable, we will use *PC-relative addressing* (also known as *RIP-relative addressing*), which is the default for AMD64. In PC-relative addressing, the memory location of global

```
var x = 42 : int64;
var b = false : bool;
```

```
    .globl x
    .section .data
    .align 8
x:
    .quad 42

    .globl b
    .section .data
    .align 8
b:
    .quad 0
```

Figure 2: Compiling global variables

variables is specified in terms of *offsets* from `%rip` instead of as absolute (64-bit) addresses in memory. This works when an executable is no more than $2^{32} = 4\text{GiB}$ in size, which is generally a reasonable assumption. To use PC-relative addressing to address `x`, say, one writes `x(%rip)` which the assembler interprets as a PC-relative address of `x` and replaces `x` by the actual offset of `x` from `%rip`. To illustrate, here is a `main` function that returns with exit code equal to the value of the global variable `x`.

```
    .globl main
    .section .text
main:
    movq x(%rip), %rax
    ret
```

3.3 *RTL with Explicit Frames (ERTL)*

To incorporate the C (SysV) calling convention, you will write a new intermediate phase of the compiler which makes a version of the RTL language but with support for explicit frames, called ERTL. The main difference between RTL and ERTL is that in ERTL the registers of the `AMD64` architecture are allowed as operands to certain instructions. These are called *machine registers* to distinguish them from the pseudo-registers of RTL.

ERTL has nearly the same set of instructions RTL. The new instructions are `newframe` and `delframe` that allocate and dispose a stack frame respectively. The `newframe` instruction must be one of the first instructions in the body of a callable, and `delframe` must be called before every `return` instruction. Before a `newframe` and after a `delframe` it is illegal to use any machine registers. When a `newframe` is subsequently compiled into assembly, your compiler needs to translate it into code that saves the current `%rsp` in `%rbp` and then subtracts the space for all the used pseudo-registers.[2] Likewise, `delframe` will restore the old values of `%rsp` and `%rbp` before exiting the function.

ERTL also modifies some of the instructions of RTL. Specifically the `call` and `return` instructions are changed to no longer take pseudo-registers as arguments. Instead, the arguments to a call must be placed in the argument registers or pushed onto the stack according to the calling convention, while `return` expects the return value to be in the `%rax` machine register.

The full list of ERTL instructions are shown in figure 3.

[2]Since we are continuing to spill all pseudo-registers onto the stack.

ERTL instruction	Description
`Li: move n, r1` $\longrightarrow$ `Lo`	move imm., `r1 = n`
`Li: copy r1, r2` $\longrightarrow$ `Lo`	copy regs, `r2 = r1`
`Li: unop op, r1` $\longrightarrow$ `Lo`	`r1 = op r1`
`Li: binop op, r1, r2` $\longrightarrow$ `Lo`	`r2 = r2 op r1`
`Li: ubranch op, r1` $\longrightarrow$ `Lo1, Lo2`	unary branch: `op r1`
`Li: bbranch, r1, r2` $\longrightarrow$ `Lo1, Lo2`	binary branch: `r1 op r2`
`Li: goto` $\longrightarrow$ `Lo`	unconditional jump
`Li: newframe` $\longrightarrow$ `Lo`	allocate new frame (size TBD)
`Li: delframe` $\longrightarrow$ `Lo`	deallocate current frame
`Li: load_param n, r` $\longrightarrow$ `Lo`	load stack param #`n` into pseudo
`Li: copy r1, r2` $\longrightarrow$ `Lo`	copy between pseudos
`Li: copy mr, r` $\longrightarrow$ `Lo`	copy machine reg into pseudo
`Li: copy r, mr` $\longrightarrow$ `Lo`	copy pseudo into machine reg
`Li: push r` $\longrightarrow$ `Lo`	push pseudo onto stack
`Li: pop r` $\longrightarrow$ `Lo`	pop pseudo from stack
`Li: call f(n)` $\longrightarrow$ `Lo`	call with `n` **non-stack** args
`Li: return`	terminate the call

`n`	number/immediate (no `$`)	`Li`, `Lo`, …	labels
`mr`	machine register	`r`, `r1`, `r2`, …	pseudo-registers
`op`	opcode		

Figure 3: ERTL instructions