COMP3012 - Compilers
Coursework 2

Venanzio Capretta

Tuesday 19 November 2024

Extend the compiler you wrote in Coursework 1 with type declarations, function definitions, and calls to functions inside expressions.

When compiling a program, the compiler should check if all the types are correct. If there is a type error, the compiler should report it. It should also report an error if an identifier is declared more than once. If there is no type error, the compiler should report that the program is type-correct.

Extented Grammar

We change the grammar for declarations to include types and functions:

declaration	::=	`var` **identifier** `:` **type**
	|	`var` **identifier** `:` **type** `:=` **expr**
	|	`fun` **identifier** `(` **vardecls** `)` `:` **type** `:=` **expr**
type	::=	`Integer` | `Boolean`
vardecl	::=	**identifier** `:` **type**
vardecls	::=	**vardecl** | **vardecl** `,` **vardecls**

We also change the grammar for expressions to allow function application. Function application binds stronger than all the operators, so we add it to the **term** grammar:

term	::=	**int** | **bool** | **identifier** | `-` **expr** | `(` **expr** `)`
	|	**identifier** `(` **exprs** `)`
exprs	::=	**expr** | **expr** `,` **exprs**

Extension of TAM

We add to TAM new instructions that will allow us to generate code for function definition and application.

The TAM running state will have a new parameter LB (Local Base). It is used when calling a function. It contains the address in the stack below which the function arguments are stored. It can be used in stack addresses: if the function has m arguments, their local values during function execution are in addresses [LB - m], [LB - $(m-1)$], ..., [LB - 1].

Siminarly, there will be a parameter SB (Stack Base) pointing at the base of the stack. Therefore, if the program has k local variables, their addresses will be [SB - $(k-1)$], [SB - $(k-2)$], ..., [SB - 0] (or just [SB]).

- CALL f
  A call to the function f: it sets up the activation record and jumps to the label f (where the code for the function is stored). The arguments of the function are assumed to be at the top of the stack in reverse order. The call will push the present value of LB onto the stack, then push the *return address* onto the stack (that is the point in the TAM program where the computation should continue on return from the function call).

- RETURN n m
  Clears the activation record (m is the number of arguments of the function) and leaves the n return values on top of the stack. It must get the return address from the activation record and set the program counter to it, get the old local base from the activation record and reset it as the present local base, clear the m local arguments from the stack, leave the top n values on the stack as the result of the function call (we ill only use it with $n = 1$).

Example

Here is an example of a program written in MiniTriangle containing type declarations and functions:

```
let
  fun square (n : Integer) : Integer = n*n;
  var x : Integer := 7;
  var y : Integer := 3
in
printint(square(x) + square(square (y)))
```

And here is its compilation into TAM code (your code does not need to be exactly the same, as long as it is correct):

```
LOADL 7
LOADL 3
LOAD [SB + 0]
CALL "square"
LOAD [SB + 1]
CALL "square"
CALL "square"
ADD
PUTINT
HALT
Label "square"
LOAD [LB - 1]
LOAD [LB - 1]
MUL
RETURN 1 1
```