Definition of Minila Programming Language

Lecture Note 3

Topics

- Minila – A mini programming language
- Expressions & statements
- Variables & environments
- Concrete syntax & abstract syntax
- Abstract syntax of expressions & statements
- Housekeeping modules
- Outline of a Minila Language Processor
  - Interpreter
  - Virtual machine
  - Compiler

Minila (1)

- Minila (a Mini-language) is an imperative programming language.
- The datatype available is only the natural number.
- Some functions/operations over natural numbers such as addition and multiplication are available.
- Program constructs are assignments, `if` statements, `while` statements, `for` statements, and sequential compositions.

Minila (2)

- The language processor for Minila consists of
  - An interpreter

    A function that interprets Minila programs directly.
  - A virtual machine

    It consists of a set of instructions and a function that executes a sequence of instructions.
  - A compiler

    A function that translates Minila programs into sequences of instructions.

Examples of Minila Programs (1)

- A program that computes the factorial of `10`:

```
v(0) := 1 ;
v(1) := 1 ;
while v(1) << 10 || v(1) === 10
do
     v(0) := v(0) ** v(1) ;
     v(1) := v(1) ++ 1 ;
od
```

- `v(0)` and `v(1)` are variables.
- It consists of three statements (two assignments and one `while` statement) that are sequentially composed.
- When the program terminates, `v(0)` contains the factorial of `10`.

Examples of Minila Programs (2)

- Another program that computes the factorial of `10`:

```
v(0) := 1 ;
for v(1) 1 10
do
    v(0) := v(1) ** v(0) ;
od
```

- It consists of two statements (one assignment and one `for` statement) that are sequentially composed.
- When the program terminates, `v(0)` contains the factorial of `10`.

Examples of Minila Programs (3)

- Note that a for statement can be expressed as a while statement. For example,

```
for v(1) 1 10
do
  v(0) := v(1) ** v(0) ;
od
```

is equivalent to (can be replaced with)

```
v(1) := 1 ;
while v(1) << 10 || v(1) === 10
do
   v(0) := v(0) ** v(1) ;
   v(1) := v(1) ++ 1 ;
od
```

- This fact can be used to define an interpreter and a compiler of Minila.

Examples of Minila Programs (4)

- A program that computes the greatest common divisor of 24 and 30 with the Euclidean algorithm:

```
v(0) := 24 ;
v(1) := 30 ;
while v(1) !== 0
do
    v(2) := v(0) %% v(1) ;
    v(0) := v(1) ;
    v(1) := v(2) ;
od
```

- When the program terminates, $\mathrm{v}(0)$ contains the result.

Examples of Minila Programs (5)

- A program that computes the integral part of the square root of $200{,}000{,}000$ with a binary search:

```
v(0) := 200000000 ;
v(1) := 0 ;
v(2) := v(0) ;
while v(1) !== v(2)
do
    if ((v(2) -- v(1)) %% 2) === 0
        then v(3) := v(1) ++ (v(2) -- v(1)) // 2 ;
        else v(3) := v(1) ++ ((v(2) -- v(1)) // 2) ++ 1 ; fi
    if v(3) ** v(3) >> v(0)
        then v(2) := v(3) -- 1 ;
        else v(1) := v(3) ; fi
od
```

- When the program terminates, `v(1)` contains the result.

Expressions (1)

- Expressions are inductively defined as follows:
  - Natural numbers `0, 1, 2, …` are expressions.
  - Variables `v(0), v(1), v(2), …` are expressions.
  - If E_1 and E_2 are expressions, so are the followings.

    `(`E_1`)`

    E_1 `++` E_2 E_1 `--` E_2 E_1 `**` E_2

    E_1 `//` E_2 E_1 `%%` E_2

    E_1 `===` E_2 E_1 `!==` E_2

    E_1 `<<` E_2 E_1 `>>` E_2

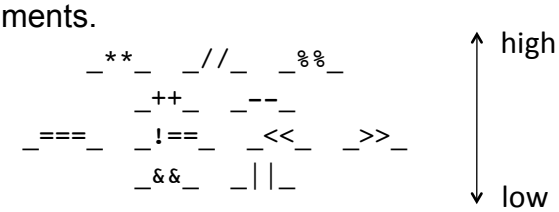
    E_1 `&&` E_2 E_1 `||` E_2

Expressions (2)

E_1 `++` E_2	Addition of E_1 and E_2.
E_1 `--` E_2	Absolute value of the difference between E_1 and E_2.
E_1 `**` E_2	Multiplication of E_1 and E_2.
E_1 `//` E_2	Quotient of dividing E_1 by E_2.
E_1 `%%` E_2	Remainder of dividing E_1 by E_2.
E_1 `===` E_2	1 if E_1 equals E_2 and 0 otherwise.
E_1 `!==` E_2	0 if E_1 equals E_2 and 1 otherwise.
E_1 `<<` E_2	1 if E_1 is less than E_2 and 0 otherwise.
E_1 `>>` E_2	1 if E_1 is greater than E_2 and 0 otherwise.
E_1 `&&` E_2	0 if at least one of E_1 and E_2 is 0 and 1 otherwise.
E_1 `||` E_2	1 if at least one of E_1 and E_2 is 1 and 0 otherwise.

Expressions (3)

- The operators are ordered w.r.t. the precedence in binding arguments.



- Examples:

```
3 ++ 4 ** 5    is parsed as   3 ++ (4 ** 5)

v(0) ++ v(1) << 10 || v(0) ++ v(1) === 10
     is parsed as
((v(0) ++ v(1)) << 10) || ((v(0) ++ v(1)) === 10)
```

Statements (1)

- Statements are inductively defined as follows:
  - The empty statement denoted by `estm` is a statement.
  - If X is a variable and E is an expression,
    X `:=` E `;` is a statement.
  - If E is an expression and S_1 and S_2 are statements,
    `if` E `then` S_1 `else` S_2 `fi` is a statement.
  - If E is an expression and S is a statement,
    `while` E `do` S `od` is a statement.
  - If X is a variable, E_1 and E_2 are expressions and S is a statement,
    `for` X E_1 E_2 `do` S `od` is a statement.
  - If S_1 and S_2 are statements, S_1 S_2 is a statement.

Statements (2)

`estm`	Does nothing.
X `:=` E `;`	Evaluates E and sets (or binds) X to the result.
`if` E `then` S_1 `else` S_2 `fi`	Executes S_1 if the result of evaluating E is not `0` and executes S_2 otherwise.
`while` E `do` S `od`	Executes S repeatedly while the result of evaluating E is not `0`.
`for` X E_1 E_2 `do` S `od`	Sets X to the result of evaluating E_1, then executes S and increments X by `1` repeatedly while X is less than or equal to the result of evaluating E_2.
S_1 S_2	Executes S_1 and then S_2.

Variables and Environments (1)

- An environment:
  - It records assignments, namely that what variables have been set to what values (natural numbers).
  - It can be implemented as a table from variables to values (natural numbers).
- Assignment: *X* `:=` *E* `;`
  - Updates an environment with *X* and the result of evaluating *E* under the environment.
- Evaluation of a variable *X*:
  - Looks for the value of the key `X` in an environment (a table).

Variables and Environments (2)

Examples:

```
1.  v(0) := 2 ;
2.  v(1) := v(0) ++ 1 ;
3.  v(0) := v(0) ++ v(1) ;
4.  v(2) := v(0) ++ v(1) ;
```

1.

variables	values
v(0)	2

2.

variables	values
v(1)	3
v(0)	2

3.

variables	values
v(1)	3
v(0)	5

4.

variables	values
v(2)	8
v(1)	3
v(0)	5

Concrete Syntax and Abstract Syntax

- Programs usually written by programmers are sequences of letters (strings).
- Such programs need to be scanned (lexically analyzed) and parsed according to a *concrete syntax* (a syntax for scanners and parsers) to construct a syntax tree.
- In the implementation of Minila in CafeOBJ, the CafeOBJ scanner and parser are used by defining the Minila syntax using mixfix function symbols.
- Such a syntax faithfully corresponds to syntax trees and is called an *abstract syntax* (a syntax for interpreters and compilers).

Abstract Syntax of Expressions (1)

- Module VAR:

```
mod! VAR {
  pr(NAT)
  [Var]
  op v : Nat -> Var {constr} .
}
```

The variables used in Minila are `v(0), v(1), …`

Abstract Syntax of Expressions (2)

- Module EXP:

```
mod! EXP {  pr(NAT)  pr(VAR)
  [Nat Var < Exp]
  op _**_    : Exp Exp -> Exp
                  {constr assoc comm prec: 29}
  op _//_      : Exp Exp -> Exp {constr prec: 29}
  op _%%_  : Exp Exp -> Exp {constr prec: 29}
  op _++_   : Exp Exp -> Exp
                 {constr assoc comm prec: 30}
  op _--_     : Exp Exp -> Exp {constr prec: 30}

  op _===_ : Exp Exp -> Exp {constr prec: 40}
  op _!==_  : Exp Exp -> Exp {constr prec: 40}
  op _<<_   : Exp Exp -> Exp {constr prec: 40}
  op _>>_   : Exp Exp -> Exp {constr prec: 40}
  op _&&_  : Exp Exp -> Exp {constr comm prec: 50}
  op _||_   : Exp Exp -> Exp {constr comm prec: 50}
}
```


Abstract Syntax of Statements

Module STM:

```
mod! STM {
  pr(EXP)
  [Stm]
  op estm : -> Stm {constr}
  op _:=_; : Var Exp -> Stm {constr}
  op if_then_else_fi : Exp Stm Stm -> Stm {constr}
  op while_do_od : Exp Stm -> Stm {constr}
  op for_ _ _do_od : Var Exp Exp Stm -> Stm {constr}
  op _ _ : Stm Stm -> Stm
                {constr assoc prec: 45 id: estm}
}
```

Housekeeping Modules (1)

The implementation of Minila uses the following housekeeping modules:

- `BOOL-ERR`
- `NAT-ERR`
- `TRIV-ERR`
- `PAIR(M :: TRIV,N :: TRIV)`
- `LIST(M :: TRIV-ERR)`
- `ENTRY`
  Pairs of variables and natural numbers.
- `ENV`
  Environments are implemented as tables from variables to natural numbers, which are implemented as lists of pairs of variables and natural numbers.
- `STACK`
  Stacks are implemented as lists of natural numbers.

Housekeeping Modules (2)

- For exception handling for Boolean values & natural numbers

```
[Bool ErrBool < Bool&Err]
op errBool : -> ErrBool {constr}

[Nat ErrNat < Nat&Err]
op errNat : -> ErrNat {constr}
```

- To this end, some functions and equations are also declared. Among them are as follows:

```
op _quo_ : Nat&Err Nat&Err -> Nat&Err
eq N quo 0 = errNat .
eq errNat quo NE = errNat .
eq NE quo errNat = errNat .
```

Housekeeping Modules (3)

- An environment is returned as the result when a program is executed.
- If an exception occurs, however, errEnv is used as the result.

```
[Env ErrEnv < Env&Err]
op errEnv : -> ErrEnv {constr}
op _|_ : ErrEntry Env&Err -> ErrEnv {constr}
op _|_ : Entry&Err ErrEnv -> ErrEnv {constr}
op _|_ : Entry&Err Env&Err -> Env&Err {constr}
eq errEntry | EV&E:Env&Err = errEnv .
eq ET&E:Entry&Err | errEnv = errEnv .
--
op update : Var Nat&Err Env&Err -> Env&Err
op lookup : Var Env&Err -> Nat&Err
```


Housekeeping Modules (4)

```
mod! STACK {
  pr(LIST(M <= TRIV-ERR2NAT-ERR)
     * {sort List -> Stack, op nil -> empstk} )
  [Stack ErrStack < Stack&Err]
  op _|_ : ErrNat Stack&Err -> ErrStack {constr}
  op _|_ : Nat&Err ErrStack -> ErrStack {constr}
  op _|_ : Nat&Err Stack&Err -> Stack&Err {constr}
  op errStack : -> ErrStack .
  eq errNat | S&E:Stack&Err = errStack .
  eq N&E:Nat&Err | errStack = errStack .
}

 view TRIV-ERR2NAT-ERR from TRIV-ERR to NAT-ERR {
   sort Elt -> Nat, sort Err -> ErrNat,
   sort Elt&Err -> Nat&Err, op err -> errNat
 }
```

Outline of Interpreter (1)

- Interprets Minila programs using an environment.
- Returns the environment used when a program terminates.
- When the interpreter takes the program:

```
v(0) := 1 ;
v(1) := 1 ;
while v(1) << 10 || v(1) === 10 do
     v(0) := v(0) ** v(1) ;
     v(1) := v(1) ++ 1 ;
od .
```

it returns the environment:

```
((< v(0) , 3628800 >) | ((< v(1) , 11 >) | empty))
```


Outline of Interpreter (2)

- When the interpreter takes the program

```
v(0) := 9999 ;
v(1) := 5 ;
while v(1) >> 0 do
    v(0) := v(0) // (v(1) -- 1) ;
    v(1) := v(1) -- 1 ;
od .
```

it returns (errEnv):ErrEnv because the exception "division by zero" occurs.

Outline of Compiler (1)

- Translates Minila programs into lists of instructions.
- When the compiler takes the program:

```
v(0) := 1 ;
v(1) := 1 ;
while v(1) << 10 || v(1) === 10 do
     v(0) := v(0) ** v(1) ;
     v(1) := v(1) ++ 1 ;
od
```

- it generates the list of instructions:

```
push(1) | store(v(0)) | push(1) | store(v(1)) |
load(v(1)) | push(10) | lessThan | load(v(1)) | push(10) |
equal | or | jumpOnCond(2) | jump(10) |
load(v(0)) | load(v(1)) | multiply | store(v(0)) |
load(v(1)) | push(1) | add | store(v(1)) |
bjump(17) | quit | clnil
```


Outline of Compiler (2)

- When the compiler takes the program

```
v(0) := 9999 ;
v(1) := 5 ;
while v(1) >> 0 do
    v(0) := v(0) // (v(1) -- 1) ;
    v(1) := v(1) -- 1 ;
od
```

it generates

```
push(9999) | store(v(0)) | push(5) | store(v(1)) |
load(v(1)) | push(0) | greaterThan | jumpOnCond(2) |
jump(12) | load(v(0)) | load(v(1)) | push(1) | minus |
divide | store(v(0)) | load(v(1)) | push(1) | minus |
store(v(1)) | bjump(15) | quit | clnil
```

Outline of Virtual Machine (1)

- Executes lists of instructions using an environment and a stack.
- Stacks (which are implemented as lists) are used to evaluate expressions.
- When it takes the list of instructions:

```
push(1) | store(v(0)) | push(1) | store(v(1)) |
load(v(1)) | push(10) | lessThan | load(v(1)) | push(10) |
equal | or | jumpOnCond(2) | jump(10) |
load(v(0)) | load(v(1)) | multiply | store(v(0)) |
load(v(1)) | push(1) | add | store(v(1)) |
bjump(17) | quit | clnil
```

- it returns the environment:

```
((< v(0) , 3628800 >) | ((< v(1) , 11 >) | empty))
```


Outline of Virtual Machine (2)

- When the virtual machine takes

```
push(9999) | store(v(0)) | push(5) | store(v(1)) |
load(v(1)) | push(0) | greaterThan | jumpOnCond(2) |
jump(12) | load(v(0)) | load(v(1)) | push(1) | minus |
divide | store(v(0)) | load(v(1)) | push(1) | minus |
store(v(1)) | bjump(15) | quit | clnil
```

it returns `(errEnv):ErrEnv` because the exception "division by zero" occurs.

Exercises

1. Declare a module for the abstract syntax of a calculator. The calculator can deal with the expressions:
   - E_1 `++` E_2 : Addition of E_1 and E_2.
   - E_1 `--` E_2 : Absolute value of $E_1 - E_2$.
   - E_1 `**` E_2 : Multiplication of E_1 and E_2.
   - E_1 `//` E_2 : Quotient of dividing E_1 by E_2.
   - E_1 `%%` E_2 : Remainder of dividing E_1 by E_2.
   - E `^2` : Square of E.
   - `/` E : Integral part of the square root of E.
   - E `!` : Factorial of E.

   The precedence is as follows: