City University London

IN2009
Language Processors

Session 7

The Source PL version 1.0 Compiling: the basics

Igor Siveroni

Session Plan

Session 7:
- The Source Programming Language (SPL)
- Syntax of a subset of SPL
- Semantics: An Interpreter for SPL
- Compiling SPL to TPL

Next Test: Monday, March 29. 1-2pm
Topic: Grammars, and a bit of SPL/TPL.
Coursework out this week

Source Programming Language (SPL)

An imperative typed programming language made of functions and statements. We introduce a subset

Program$_S$ → *MainDecl*

MainDecl → `void main(){` *VarDecl*$^+$ *Statement*$^+$ `}`

VarDecl → `int` *id* `;`

Statement → *AssignStm* | *PrintStm* | *IfStm* | *WhileStm*

AssignStm → *id* `:=` *exp* `;`

PrintStm → `print(` *exp* `)` `;`

SPL: Syntax

IfStm → `if (` *exp* `)` `then` *Block* `else` *Block*

WhileStmr → `while(` *exp* `)` *Block*

Block → `{` *Statement*$^+$ `}`

exp → *exp AOp exp* | *exp BOp exp* | *exp COp exp* |
id | *integer* | `!(`*exp*`)` | `true` | `false`

Aop → + | - | * | /   *Bop* → `and` | `or`

Cop → > | >= | ==

SPL: Example

```
void main() {
  int x; int y;
  x := 1;
  while (5 > x) { print(y+2); x := x+1; }
}
```

Basic Features:

- A single main method. All variables of type int.
- Undeclared variables are reported.
- Runtime checking e.g.
  `while (5)` is not allowed, test must be a boolean

SPL: Abstract Syntax

We will use a special type of abstract syntax:

Program(MainDecl m)
MainDecl(List<id> vars, Statement+ ls)
AssignStm(Id v,Exp e)
PrintStm(Exp e)
IfStm(Exp e, Statement+ ls1, Statement+ ls2)
While(Exp e, Statement+ body)
OpExp(Exp e1, Aop op, Exp e2)
BoolExp(Exp e1, Bop op, Exp e2)
CmpExp(Exp e1, Cop op, Exp e2)
IdExp(Id x)   IntLiteralExp(int n)   BoolLiteralExp(bool b)

SPL: Semantics

We will model:

• Memory, to keep track of the values of variables. We use a lookuptable **table** of class **Table**:

  • table.update(x,v): Variable x has now value v.

  • table.lookup(x): returns the value associated to x.

• Execution of statements: **execStm(s,table,stdout),** where table t and stdout can be updated.

• Evaluation of expressions: **evalExp(e,table)=v**. No side-effects, just returns the value of expression e.

SPL: Semantics

Now we can easily model the execution of the program. We start with an empty Table **table**, and an empty standard output **stdout**:

execProgram(P,table,stdout):

P is Program(MainDecl m) // this is abstract syntax
m is MainDecl(Id* vars, Statement* ls)

For each x in vars: **table.update(v,0)**
  (initialise each variable to 0)

For each s in ls: **execStm(s,table,stdout)**
  (execute each statement in main body)

SPL: Semantics

execStm(AssignStm(Id x, Exp e),table,stdout):
The execution of “x := e” is defined by:

v = evalExp(e,table) // evaluate e
ReportError if v is not integer
table.update(x,v) // update variable with new value

execStm(PrintStm(Exp e), table, stdout)
The execution of “print(e)”

v = evalExp(e,table) // evaluate e
print value **v** to standard output **stdout**

SPL: Semantics

**execStm(IfStm(Exp e,Stm+ ls1,Stm+ ls2),
table,stdout):**
The execution of “if(e) then { ls1 } else { ls2 }” is defined by:

v = evalExp(e,table) // evaluate test e
ReportError if v is not boolean (true,false)
if v is true
  for each s in ls1: **execStm(s, table, stdout)**
else
  for each s in ls2: **execStm(s, table, stdout)**

Note the runtime check on the type of v.

SPL: Semantics

**execStm(WhileStm(Exp e,Stm+ body),
table,stdout):**
The execution of “while(e) then { body }” is defined by:

start:
v = evalExp(e,table) // evaluate test e
ReportError if v is not boolean (true,false)
if v is true
  for each s in ls1: **execStm(s, table, stdout)**
  goto **start**

SPL: Semantics

evalExp(OpExp(Exp e1,Aop op, Exp e2), table):
The execution of “e1 op e2” is defined by:

v1 = evalExp(e1,table)
v2 = evalExp(e2,table)
ReportError if v1 or v2 are not integers
if (op is +) result = v1 + v2
if (op is -) result = v1 - v2
…

return result of type int

SPL: Semantics

evalExp(BoolExp(Exp e1,BOp op, Exp e2), table):
The execution of “e1 op e2” is defined by:

v1 = evalExp(e1,table)
v2 = evalExp(e2,table)
ReportError if v1 or v2 are not boolean
if (op is and) result = v1 && v2
if (op is or) result = v1 || v2

return result of type boolean

SPL: Semantics

evalExp(CmpExp(Exp e1,COp op, Exp e2), table):
The execution of “e1 op e2” is defined by:

v1 = evalExp(e1,table)
v2 = evalExp(e2,table)
ReportError if v1 or v2 are not integer
if (op is >) result = (v1 > v2)
if (op is >) result = (v1 >= v2)
if (op is ==) result = (v1 == v2)

return result of type boolean

SPL: Semantics

evalExp(NumLiteralExp(int n), table) :

return n of type int

evalExp(IdExp(Id x), table) :

ReportError if x is not in table
v = table.lookup(x)
return v of type int

evalExp(BoolLiteralExp(bool t), table) :

return t of type bolean

SPL: Runtime errors

```
void main() {
int x; int y;
x := (y < 5);     // type error
y := 10*z;        // variable undefined
x := x + true;    // type error
}
```


SPL Semantics: Why?

- In order to compile SPL we need to fully understand its semantics I.e. the meaning of SPL programs.
- Once we’ve understood this, we can define translations from SPL to TPL
- The understanding of runtime errors is important. We don’t want to generate code that crashes at runtime. This justifies the inclusion of a typechecker (next session)
- We will start by defining the translation (compilation) of statements.

Target Programming Language (TPL)

Program$_T$ → *Instruction*$^+$

Instruction → *StoreInstr* | *BinopInstr* | *UopInstr* |
JumpInstr | *IOInstr* | **`STACKALLOC`** ***n***
StoreInstr → `STORE` *Arg*, *Res*
BinopInstr → *Op*$_l$ *Arg1*, *Arg2*, *Res*
UopInstr → `UMINUS` *Arg*, *Res*| `NOT` *Arg*, *Res*
JumpInstr → `LABEL` *Lname* | `JMP` *Lname* |
`JMP0` *Arg*, *Lname* | `JMP1` *Arg*, *Lname*
IOInstr → `WRITEI` *Arg* | `READI` *Res*

Op→ `ADDI` | `SUBI` | `MULTI` | `DIVI`| `AND` | `OR` | `XOR`
`EQ` | `NE` | `GT` | `GE` | `LT` | `LE`

Memory and registers

Values can be stored into (and read from) memory or registers:

Reference → *$Location* | *Register*(*Offset*)
Location ::= Integer

Memory locations start at address 0.

• `STACKALLOC` **n** allocates n words of memory. The program can only access allocated memory.

•This subset of the language works only with integers - we only have the integer versions of operations e.g. ADDI.

Translating SPL into TPL

We will use the following functions:

• Symbol table **stable:** It contains information collected and computed by the compiler, such as memory addreses and temporaries. Details later.

• **genCode(P,stable), genCode(ls,stable) genCode (s,stable), genCode(e, stable):**
Returns a sequence of TPL instructions, where P, ls, s and e are a TPL Program, statement list, statement and expression, respectively.

• **genLabel():** Returns a fresh program label.

Translating SPL: Program

genCode(Program(MainDecl m), stable):

m is MainDecl(Id* vars, Statement* ls)
x1,…,xn in vars // n vars
s1,…,sm in ls

Code = STACKALLOC n
STORE 0,$0
….
STORE 0,$(n-1)
codeGen(s1, stable)
…
codeGen(sm, stable)

Translating SPL: Assignment

genCode(AssignStm(Id x, Exp e), stable):

t = stable.getTemp(e) // get temporary location
a = stable.getAddress(x) // address of x

Code = genCode(e,stable)
`STORE` t, a

• Variable assignment: z = x + y + 10

```
ADDI $2, $4, R1
ADDI R1, 10, R2
STORE R2, $6
```

`Where $6 was the address returned by the symbol table for z.`

Example: If Conditional

genCode(IfStm(Exp e, Stm+ ls1, Stm+ ls2), stable)
t = stable.getTemp(e) // get temporary location
L1 = genLabel()
L2 = genLabel()

Code = genCode(e,stable)
`JMP0` t, L1
genCode(ls1,stable)
`JMP` L2
`LABEL` L1
genCode(ls2, stable)
`LABEL` L2

Example: If Conditional

"if (x >= 15) then y = 20 else y = 30"

Assuming x and y are stored in locations 10 and 12, the translation of the code above may look like:

```
GE $10,15,R1
JMP0 R1,L1
STORE 20,$12
JMP L2
LABEL L1
ADDI 30,$12
LABEL L2
```

Translating TPL: While loop

genCode(WhileStm(Exp e, Stm+ body), stable)

t = stable.getTemp(e) // get temporary location
L1 = genLabel()
L2 = genLabel()

Code = `LABEL` L1
genCode(e,stable)
`JMP0` t, L2
genCode(body,stable)
`JMP` L1
`LABEL` L2

Translating TPL: While loop

"x = 0; while (x < 5) { x = x + 1; print(x); }"

An equivalent program in TPL:

```
STORE 0,$2
LABEL L1
LT $2,5,R1
JMP0 R1,L2
ADDI $2,1,$2
WRITEI $2
JMP L1
LABEL L2
```


What's next

- Define genCode(Exp e, SymbolTable stable)
- Define the symbol table.
- Extend SPL and introduce typechecking.
- Extend TPL and complete translation with types.
- Translate Function definitions and calls.