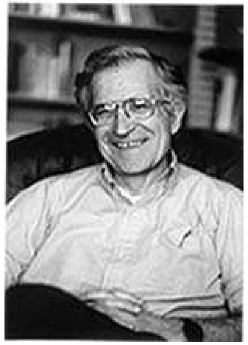
« Operational semantics »

Patrick Cousot
Jerome C. Hunsaker Visiting Professor
Massachusetts Institute of Technology
Department of Aeronautics and Astronautics

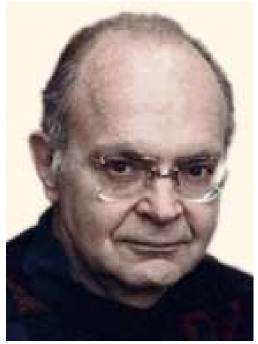
cousot@mit.edu
www.mit.edu/~cousot

Course 16.399: "Abstract interpretation"
http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/

Parsing and Translation



Noam Chomsky



Donald E. Knuth

Reference

[1] Noam Chomsky. "On certain formal properties of grammars". Information and Control 2 (1959), 137-167.

[2] Don E. Knuth. On the Translation of Languages from Left to Right. Information and Control, 8: 607–63, 1965.

Principle of Parsing and Translation

Principle of the lexer

– The *lexems*[1] (or *tokens*) are described by a regular expression;
– The regular expression is translated (by `ocamllex`) into a finite automaton and next into a lexer, that is a program transforming sequences of characters into a lexems
– In the lexer, the semantic actions are in charge of the translation of each lexem into a *semantic value*[2] (also called *attribute*)
– By composition the lexer maps the program file into a sequence of lexems with associated semantic values

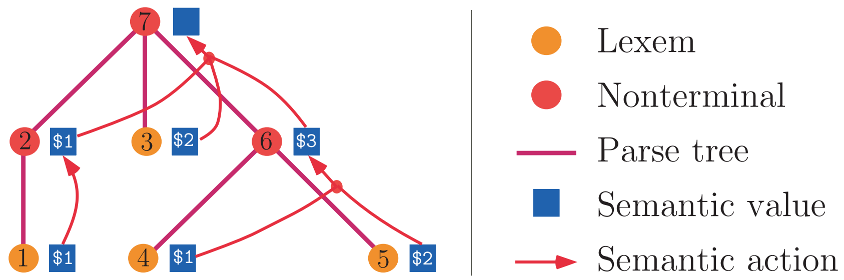
[1] i.e.: elementary meaningful pieces of text such as a reserved word like `begin`, a user-defined identifier, a symbol like <=, etc.
[2] For example the semantic value associated to a text 12345 is the numerical value of the number in base 10.

Principle of the parser

– The program structure is described by a grammar[3] specifying a syntax tree of the program (considered as a sequence of lexems)
– The grammar is translated (by `ocamlyacc`) into a stack automaton and next into a parser, that is a program exploring the program syntax tree from bottom-up and left to right
– In the parser, the semantic actions are in charge of the translation of each grammar rule into a semantic value (reusing the translation of the immediate derivatives of the rule)

[3] technically for YACC and `ocamlyacc` the context-free grammar must be in the class LALR(1), otherwise ambiguities have to be solved by hand.

Principle of the translation



In conclusion, the parser executes the semantic actions to compute semantic values in the order of exploration of the virtual parse tree from bottom-up and left to right

The calculator example

Parse and execute a sequence of commands. A command assigns a value to a variable or computes the value of an arithmetic expression.

```
% ./calculator
X:=1; Y:=2; Z:=3; (X+Y)*Z+4
X = 1;
Y = 2;
Z = 3;
13
X:=1; Y:=2; *X
X = 1;
Y = 2;
Syntax error ...
1
%
```

The calculator grammar

prog	::=	list	Program
list	::=	cmd ; list	List of commands
	|	cmd	
cmd	::=	assign	Command
	|	expr	
assign	::=	IDENT := expr	Assignment
expr	::=	expr + expr	Expression
	|	expr - expr	
	|	expr * expr	
	|	expr / expr	
	|	- expr	
	|	(expr)	
	|	IDENT	Identifier lexem $([a-z]\|[A{-}Z])([a{-}z]\|[A{-}Z]\|[0{-}9])^*$
	|	NUM	Number lexem $[0{-}9]^+$

The Lexer

The lexer generator

- The user provides a specification of the lexems (and of the corresponding semantic values) by an alternative regular expression (and semantic actions written in `Ocaml` for each alternative);
- The recognized lexem is the longest generated by an alternatice of the regular expression;
- When a lexem is recognized, a value (of type `token`) is returned, as specified by the semantic action associated with that alternative;
- The type `token` is defined later (with the grammar);
- The `ocamllex` version of `LEX` will automatically construct a lexer in `Ocaml` from that specification.

The calculator lexer

```
(* File calculatorLEX.mll *)
{
open CalculatorYACC;; (* Type token defined in CalculatorYACC.mli *)
exception Eof;;
}
rule token = parse
    [' ' '\t'] { token lexbuf } (* skip blanks and tabs *)
  | ['\n' ]    { EOL } (* end of line *)
  | (['a'-'z'] | ['A'-'Z'])(['a'-'z'] | ['A'-'Z'] | ['0'-'9'])* as idt
               { IDENT idt }
  | ['0'-'9']+ as num
               { NUM (int_of_string num) }
  | ';'        { SEMICOLON }
  | ':' '='    { ASSIGN }
```

```
    | '+'        { PLUS }
    | '-'        { MINUS }
    | '*'        { TIMES }
    | '/'        { DIV }
    | '('        { LPAREN }
    | ')'        { RPAREN }
    | eof        { raise Eof } (* end of file *)
```


Syntax of the lexical definitions

{ OCAML text [4] (* header *) }
rule lexem =
  parse regular expression$_1$ { semantic action$_1$ }
    | ...
    | regular expression$_2$ as lxm { semantic action$_2$ }
and lexem =
  parse ...
and ...
;;

[4] Including the declaration of the type of the semantic values returned by the lexems as generated by the compilation of the parser.

Regular expressions

'c'	character c;
_	any character;
eof	end of file;
" string "	character string;
[C]	set of characters;
[^ C]	character set complement;
ρ*	repetition zero or several times;
ρ+	repetition one or more times;
ρ?	empty or ρ;
ρ_1 | ρ_2	union of languages;
ρ_1 ρ_2	concatenation;
(ρ)	parentheses.

Character sets

' c '	character;
' c_1 ' - ' c_2 '	character interval [5];
C_1 ^ C_2	union of character sets.

[5] in ASCII order.

Lexer semantic actions

- After a regular expression, the clause `as ident` assigns the character string recognized by the corresponding alternative of the regular expression to `ident`. This value can then pbe use in the corresponding semantic action [6];
- In the Lexing module , `lexbuf` is a lexer buffer (of type `lexbuf`) created either from a file channel or from a character string:

```
val from_channel : in_channel -> lexbuf
val from_string  : string     -> lexbuf
```

[6] As `lxm` in `semantic action`$_2$.

Types of the lexem semantic values [7]

```
type token =
  | EOL
  | SEMICOLON
  | ASSIGN
  | PLUS
  | MINUS
  | TIMES
  | DIV
  | LPAREN
  | RPAREN
  | IDENT of ( string )
  | NUM of ( int )

val prog :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> unit
```

[7] The file `calculatorYACC.mli` is automatically generated (together with the parser `calculatorYACC.ml`) by `ocamlyacc` from `calculatorYACC.mly`. It is then used by the lexer generator `ocamllex` to determine the type of the semantic values returned by lexems.

The Parser

The parser generator

- The user provides a list of lexems and a grammar grammar [8] with semantic actions (written in`Ocaml`)
- The `ocamlyacc` version of `YACC` [3] in `Ocaml` will generate a bottom-up, left-to-right syntax analyzer and the type (`token`) for the lexem semantic values;

Reference

[3] Stephen C. Johnson. "YACC: Yet another compiler-compiler". Unix Programmer's Manual, Vol. 2b, 1979.

[8] technically for YACC and `ocamlyacc` the context-free grammar must be in the class LALR(1), otherwise ambiguities have to be solved by hand.

Parsing

– The parsing of a sequence of lexems:
  - corresponds to the traversal of the syntactic tree, if any [9], from left-to-right and bottom-up;
  - calls the lexer to successively get the lexems in sequence;
  - returns for each nonterminal the semantic value computed by the corresponding semantic action.

[9] If the sentence cannot be parsed by the grammar, the parser stops at the rist error.

Grammar semantic actions

– The semantic value of the lefthand side nonterminal X of a grammar rule

$$X ::= X_1 \ldots X_n \quad \{\mathrm{semantic\_action}(\$1, \ldots, \$n)\}$$

is defined as the value of $\mathrm{semantic\_action}(\$1, \ldots, \$n)$.

It depends upon the semantic values $\$1, \ldots, \$n$ of the lexems and nonterminals $X_1, \ldots, X_n$ appearing on the righthand side of the rule (and, maybe, on the values of global variables such as symbol tables [10]).

[10] The symbol table essentially records the user-defined identifiers.

The calculator parser

```
/* File calculatorYACC.mly */

%{ (* header *)

type symbTable = (string * int) list ;;

let sb = ref([] : symbTable) ;;

let getvalue x =
   if (List.mem_assoc x !sb) then
     (List.assoc x !sb)
   else
     0;;

```



```
let rec except x l = match l with
  []   -> []
| h::t -> if (h = x) then t
            else h::(except x t)

let setvalue x v =
  (print_string (x ^ " = "); print_int (v);
   print_string ";\n"; flush stdout;
   if (List.mem_assoc x !sb) then
     sb := (x, v) :: (except (x, (List.assoc x !sb)) !sb)
   else
     sb := (x, v) :: !sb
  );;

%} /* declarations */

%token EOL SEMICOLON ASSIGN PLUS /* lexer tokens */
%token MINUS TIMES DIV LPAREN RPAREN
```

```
%token < string > IDENT
%token < int > NUM
%start prog                    /* the entry point */
%type <unit> prog
%type <int> list
%type <int> cmd
%type <int> assign
%type <int> expr
%left PLUS MINUS             /* lowest precedence */
%left TIMES DIV              /* medium precedence */
%nonassoc UMINUS            /* highest precedence */

%% /* rules */

prog :
    list EOL  { print_int $1 ; print_newline(); flush stdout; () }

list :
```



```
    cmd SEMICOLON list  { $3 }
  | cmd                 { $1 }

cmd :
    assign  { $1 }
  | expr    { $1 }

assign :
    IDENT ASSIGN expr  { (setvalue $1 $3) ; $3 }

expr :
    expr PLUS expr           { $1 + $3 }
  | expr MINUS expr          { $1 - $3 }
  | expr TIMES expr          { $1 * $3 }
  | expr DIV expr            { $1 / $3 }
  | MINUS expr %prec UMINUS  { - $2 }
  | LPAREN expr RPAREN       { $2 }
  | IDENT                    { (getvalue $1) }
```



```
  | NUM                      { $1 }

%% (* trailer *)
```


Metasyntax of the syntactic definitions

```
%{
(* prelude, Ocaml code *)
%}
 /* declarations */
%%
 /* grammar rules */
%%
(* postlude, Ocaml code *)
```

The grammar must be in the LALR(1) class. Otherwise, ambiguities can be explicitly solved by priorities specified by precedence declarations.

Token declarations

Declarations :

`%token terminal...terminal` lexems [11];

`%token < type > terminal...terminal` type of the lexem semantic values [12];

[11] used as constructors in the `token` type.

[12] used as parameterized constructors in the `token` type with qualified type `type` as argument.

Axiom and non-terminal declarations

`%start non-terminal...non-terminal` axioms of the grammar [13];

`%type < type > non-terminal...non-terminal` type of the semantic values of the non-terminals [14];

[13] for which a syntax analysis function « non-terminal lexem `lexbuf` » is generated (where « lexem » is created by `LEX` and « `lexbuf` » is a lexical buffer).

[14] strictly required for the grammar axioms only.

Priority/precedence declarations

`%left terminal...terminal` lexems with same precedence, left-associative;

`%right terminal...terminal` lexems with same precedence, right-associative;

`%nonassoc terminal...terminal` lexems with same precedence, non associative;

These precedence declarations are given in order, from lowest to highest priorities.

Grammar rules

Rules:

```
non-terminal :
      symbol ... symbol { semantic action}
    | ...
    | symbol ... symbol %prec terminal { semantic action}
    | symbol ... symbol { semantic action}
    ;
```

Symbol: lexem or grammar non-terminal (their syntax is that of `Ocaml` identifiers);

Precedence[15] *:* « %prec terminal » indicates that the priority and 'associativity of the rule will be defined by the precedence and associativity declarations for the « terminal » (which can be fictious, as UMINUS);

Semantic actions: Ocaml expressions where $\$i$ denote the semantic value of the i^{th} lexical or syntactic symbol, counted from left to right, starting from 1;

[15] can be used to solve conflicts for ambiguous grammars. The option -v of ocamlyacc can be used to report on all conflicts. For specialists, a reduce/reduce conflict is solved according to the order of the grammar rules. A shift/reduction conflict is solved in favor of the shift. These default rules can be changed thanks to the precedence declarations.

Generation and compilation of the parser and lexer

Given the grammar parser.mly and lexer lexer.mll specifications:

1) The Ocaml lexer lexer.ml is generated by:

   ocamllex lexer.mll;

2) The Ocaml parser parser.ml and a file parser.mli containing the token type are generated by:

   ocamlyacc syntax.mly;

3) The Ocaml files parser.mli (defining the token type), lexer.ml (using the token type) and syntax.ml are compiled;

4) The parsing functions for the grammar axioms can then be used in the main program.

The calculator

```
(* File calculator.ml *)
open Parsing;;
try
  let lexbuf = Lexing.from_channel stdin in
  while true do
    try
      CalculatorYACC.prog CalculatorLEX.token lexbuf
    with Parse_error ->
      (print_string "Syntax error ..." ; print_newline ()) ;
    clear_parser ()
  done
with CalculatorLEX.Eof ->
  ()
;;
```


Compilation of the calculator (makefile)

```
# make delete
# script calculator.typescript
# make
# ^D

.PHONY : all
all :
	ls
	@echo '# Lexer specification:'
	cat calculatorLEX.mll
	@echo '# Lexer creation:'
	ocamllex calculatorLEX.mll
	ls
	@echo '# Parser specification:'
```

```
	cat calculatorYACC.mly
	@echo '# Parser creation:'
	ocamlyacc calculatorYACC.mly
	ls
	@echo '# Types of lexem returned values:'
	cat calculatorYACC.mli
	@echo '# Compilation of the lexer and parser:'
	ocamlc -c calculatorYACC.mli
	ocamlc -c calculatorLEX.ml
	ocamlc -c calculatorYACC.ml
	@echo '# Specification of the calculator'
	cat calculator.ml
	ocamlc -c calculator.ml
	@echo '# Linking and code generation for the lexer, '
	@echo '# parser and calculator:'
	ocamlc -o calculator calculatorLEX.cmo calculatorYACC.cmo \
	calculator.cmo
	ls
```



```
	@echo '# Using the calculator:'
	echo "X:=1; Y:=2; Z:= 3; (X+Y)*Z+4" | ./calculator
	echo "X:=1; Y:=2; *X; X" | ./calculator

.PHONY : clean
clean :
	@-/bin/rm -f *.cmo *.cmi calculatorLEX.ml calculatorYACC.mli \
	calculatorYACC.ml || true

.PHONY : delete
delete : clean
	@-/bin/rm -f calculator typescript
```


Examples of execution of the calculator

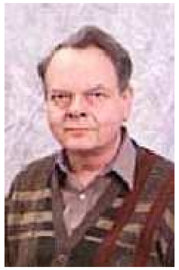
```
Script started on Sat Feb 26 14:52:00 2005
% make

ls
README         calculator.typescript   calculatorYACC.mly
calculator.ml      calculatorLEX.mll   makefile
# Lexer specification:
cat calculatorLEX.mll
(* File calculatorLEX.mll *)
{
open CalculatorYACC;; (* Type token defined in CalculatorYACC.mli *)
exception Eof;;
}
rule token = parse
```



```

echo "X:=1; Y:=2; *X; X" | ./calculator
X = 1;
Y = 2;
Syntax error ...

% ^Dexit

Script done on Sat Feb 26 14:52:12 2005
```

Operational Semantics

John McCarthy



John Reynolds

Gordon Plotkin

References

[4] John McCarthy. "Recursive Functions of Symbolic Expressions and their Computation by Machine (Part I)". *Communications of the ACM*, Volume 3, Issue 4 (April 1960), pp. 184–195.

[5] John Reynolds. "Definitional interpreters for higher-order programming languages". Proceedings of the ACM annual conference - Volume 2 table of contents Boston, Mass., pp. 717–740, 1972.

[6] Gordon Plotkin. "A Structural Approach to Operational Semantics". Lecture notes. Univ. of Århus, Denmark, 1981.

Concrete Syntax of the Simple Imperative Language (SIL)

Concrete Syntax of Numbers and Variabes

The Simple Imperative Language (SIL) has the following concrete syntax:

- Spaces, tabulations and end of lines are ignored
- Comments are between percent (%) characters with no % inside
- $\mathrm{NAT} ::= (0|1|\ldots|9)^+$ natural numbers [16]
- $\mathrm{LETTER} ::= a|\ldots|z|A|\ldots|Z$ letters
- $\mathrm{VAR} ::= \mathrm{LETTER}(\mathrm{LETTER}|\mathrm{NAT})^*$ variables [17]

[16] x^+ means $\underbrace{x\ldots x}_{n \text{ times}}$, $n \geq 1$

[17] x^* means $\underbrace{x\ldots x}_{n \text{ times}}$, $n \geq 0$ i.e. empty string when $n = 0$

Concrete Syntax of Arithmetic Expressions

Aexp ::=		Arithmetic expression
	?	random value
$\mid$	NAT	natural number
$\mid$	VAR	variable
$\mid$	Aexp + Aexp	addition
$\mid$	Aexp − Aexp	substraction
$\mid$	Aexp ∗ Aexp	multiplication
$\mid$	Aexp / Aexp	division
$\mid$	Aexp `mod` Aexp	modulo
$\mid$	+ Aexp	identity
$\mid$	− Aexp	opposite
$\mid$	(Aexp)	parentheses

Concrete Syntax of Boolean Expressions

Bexp ::=		Boolean expression
	`true`	truth
$\mid$	`false`	falsity
$\mid$	Aexp < Aexp	strictly less than
$\mid$	Aexp <= Aexp	less than or equal
$\mid$	Aexp = Aexp	equal
$\mid$	Aexp >= Aexp	greater than or equal
$\mid$	Aexp > Aexp	strictly greater than
$\mid$	Bexp $\mid$ Bexp	disjunction
$\mid$	Bexp & Bexp	conjunction
$\mid$	¬ Bexp	negation
$\mid$	(Bexp)	parentheses

Priority of operators

Operator	Priority	Associativity
$\mid$	1 (lowest)	left
&	2	left
¬	3	right (unary)
<, <=, =, >, >=	4	none
+, −	5	left (binary)
∗, /, `mod`	6	left
+, −	7 (hightest)	right (unary)

Concrete Syntax of Commands

Com ::=		Commands
	`skip`	void
$\mid$	VAR := Aexp	assignment
$\mid$	`if` Bexp `then` LCo `else` LCo `fi`	test
$\mid$	`while` Bexp `do` LCo `od`	iteration
LCo ::=		List of commands
	Com	
$\mid$	Com ; LCo	

Concrete Syntax of Programs

Prog ::= LCo; ;   Programs

Abstract Syntax of Numbers and Variables

Numbers	
$\mathtt{d} \in \mathsf{Digit} ::= 0 \mid 1 \mid \ldots \mid 9$	digits,
$\mathtt{n} \in \mathsf{Nat} ::= \mathsf{Digit} \mid \mathsf{Nat}\ \mathsf{Digit}$	numbers in decimal notation.
Variables	
$\mathtt{X} \in \mathbb{V}$	variables/identifiers.

Abstract Syntax of the Simple Imperative Language (SIL)

Abstract Syntax of Arithmetic Expressions

Arithmetic expressions	
$A \in \mathsf{Aexp} ::= \mathtt{n}$	numbers,
$\mid \mathtt{X}$	variables,
$\mid \mathtt{?}$	random machine integer,
$\mid +A \mid -A$	unary operators,
$\mid A_1 + A_2 \mid A_1 - A_2$	binary operators,
$\mid A_1 * A_2 \mid A_1 \mathbin{/} A_2$	
$\mid A_1 \ \mathtt{mod}\ A_2$.	

Abstract Syntax of Boolean Expressions

Arithmetic expressions

$A_1, A_2 \in \mathrm{Aexp}$.

Boolean expressions

$B, B_1, B_2 \in \mathrm{Bexp} ::=$	`true`	truth,
$\mid$	`false`	falsity,
$\mid$	$A_1 = A_2 \mid A_1 < A_2$	arithmetic comparison,
$\mid$	B_1 & B_2	conjunction,
$\mid$	$B_1 \mid B_2$	disjunction.

Abstract Syntax of Commands

Commands

$C \in \mathrm{Com} ::=$	`skip`	identity,
$\mid$	`X := `A	assignment,
$\mid$	`if` B `then` S_1 `else` S_2 `fi`	conditional,
$\mid$	`while` B `do` S `od`	iteration.

List of commands

$S, S_1, S_2 \in \mathrm{Seq} ::=$	C	command,
$\mid$	C ; S	sequence.

Abstract Syntax of Programs

Program

$P \in \mathrm{Prog} ::= S$;; program.

Implementation of the syntax of the Simple Imperative Language (SIL) in OCaml

Program Variables

Variables are implicitly declared. The free variables

$$\mathsf{Var} \in (\mathsf{Prog} \cup \mathsf{Com} \cup \mathsf{Seq} \cup \mathsf{Aexp} \cup \mathsf{Bexp}) \mapsto \wp(\mathbb{V})$$

are defined by structural induction [18] for *programs* ($S \in$ Seq)

$$\mathsf{Var}[\![S\ ;;]\!] \stackrel{\text{def}}{=} \mathsf{Var}[\![S]\!]$$

[18] Note that the relation "is an immediate component of" (e.g. S is an immedaite component of $P = S\ ;;$) is well-founded so that the recursive definition is well-defined and unique.

list of commands ($C \in$ Com, $S \in$ Seq)

$$\mathsf{Var}[\![C\ ;\ S]\!] \stackrel{\text{def}}{=} \mathsf{Var}[\![C]\!] \cup \mathsf{Var}[\![S]\!]$$

commands ($\mathtt{X} \in \mathbb{V}$, $A \in$ Aexp, $B \in$ Bexp, S, S_t, $S_f \in$ Seq)

$$\begin{aligned}
\mathsf{Var}[\![\mathtt{skip}]\!] &\stackrel{\text{def}}{=} \emptyset\\
\mathsf{Var}[\![\mathtt{X} := A]\!] &\stackrel{\text{def}}{=} \{\mathtt{X}\} \cup \mathsf{Var}[\![A]\!]\\
\mathsf{Var}[\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!] &\stackrel{\text{def}}{=} \mathsf{Var}[\![B]\!] \cup \mathsf{Var}[\![S_t]\!] \cup\\
&\qquad \mathsf{Var}[\![S_f]\!]\\
\mathsf{Var}[\![\mathtt{while}\ B\ \mathtt{do}\ S\ \mathtt{od}]\!] &\stackrel{\text{def}}{=} \mathsf{Var}[\![B]\!] \cup \mathsf{Var}[\![S]\!]
\end{aligned}$$

arithmetic expressions ($\mathtt{n} \in$ Nat, $\mathtt{X} \in \mathbb{V}$, $\mathtt{u} \in \{+, -\}$, A_1, $A_2 \in$ Aexp, $\mathtt{b} \in \{+, -, *, /, \mathtt{mod}\}$)

$$\begin{aligned}
\mathsf{Var}[\![\mathtt{n}]\!] &\stackrel{\text{def}}{=} \emptyset & \mathsf{Var}[\![\mathtt{u}\ A_1]\!] &\stackrel{\text{def}}{=} \mathsf{Var}[\![A_1]\!]\\
\mathsf{Var}[\![\mathtt{X}]\!] &\stackrel{\text{def}}{=} \{\mathtt{X}\} & \mathsf{Var}[\![A_1\ \mathtt{b}\ A_2]\!] &\stackrel{\text{def}}{=} \mathsf{Var}[\![A_1]\!] \cup \mathsf{Var}[\![A_2]\!]\\
\mathsf{Var}[\![?]\!] &\stackrel{\text{def}}{=} \emptyset
\end{aligned}$$

and *boolean expressions* (A_1, $A_2 \in$ Aexp, $\mathtt{r} \in \{=, <\}$, B_1, $B_2 \in$ Bexp, $\mathtt{l} \in \{\&, |\}$)

$$\begin{aligned}
\mathsf{Var}[\![\mathtt{true}]\!] &\stackrel{\text{def}}{=} \emptyset, & \mathsf{Var}[\![A_1\ \mathtt{r}\ A_2]\!] &\stackrel{\text{def}}{=} \mathsf{Var}[\![A_1]\!] \cup \mathsf{Var}[\![A_2]\!]\\
\mathsf{Var}[\![\mathtt{false}]\!] &\stackrel{\text{def}}{=} \emptyset, & \mathsf{Var}[\![B_1\ \mathtt{l}\ B_2]\!] &\stackrel{\text{def}}{=} \mathsf{Var}[\![B_1]\!] \cup \mathsf{Var}[\![B_2]\!]
\end{aligned}$$

Symbol table

The symbol table records the list of program variables (each being assigned a unique number).

```
(* symbol_Table.mli *)
(* initialization of the symbol table to empty              *)
val init_symb_table     : unit -> unit
(* variables are represented by their rank                  *)
type variable = int
(* if absent, add variable with given string to symbol table *)
(* and return its rank                                      *)
val add_symb_table      : string -> variable
(* number of variables in the symbol table                  *)
val number_of_variables : unit -> int
(* string of given variable in the symbol table             *)
val string_of_variable  : variable -> string
```

```
(* print given program variable                             *)
val print_variable       : variable -> unit
(* (for_all_variables f) iterates application of f to all    *)
(* variables in the symbol table                            *)
val for_all_variables    : (variable -> 'a) -> unit
(* (print_map_variables f) prints {...; vi : f vi;...} for   *)
(* all variables vi in the symbol table                     *)
val print_map_variables : (variable -> unit) -> unit
(* map_variables p = (p v0) (p v1) ... (p vn-2) (p vn-1)     *)
(* where v0, ..., vn-1 are the n >= 0 program variables     *)
val map_variables        : (variable -> unit) -> unit
(* map2_variables p q =                                     *)
(*   (p v0) (q v1) (p v1) ... (p vn-2) (q vn-1) (p vn-1)     *)
(* where v0, ..., vn-1 are the n >= 0 program variables     *)
val map2_variables       : (variable -> unit) ->
                                     (variable -> unit) -> unit
```

See `symbol_Table.ml` for the `OCaml` implementation.

Variables

Provides an abstract view of the symbol table where the only visible operations are thore on variables:

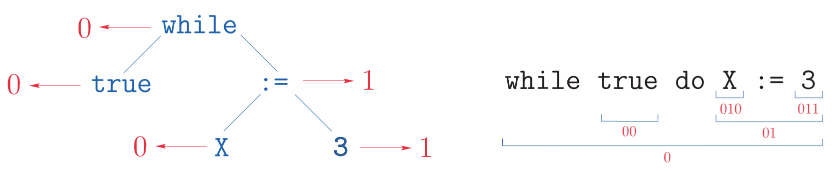
```
(* variables.mli *)
open Symbol_Table
type variable = Symbol_Table.variable
val number_of_variables : unit -> int
val for_all_variables   : (variable -> 'a) -> unit
val print_variable      : variable -> unit
val print_map_variables : (variable -> unit) -> unit
val map_variables       : (variable -> unit) ->  unit
val map2_variables      : (variable -> unit) ->
                                   (variable -> unit) -> unit
val string_of_variable  : variable -> string
```



```
(* variables.ml *)
open Symbol_Table
type variable = Symbol_Table.variable
let number_of_variables = number_of_variables
let for_all_variables   = for_all_variables
let print_variable      = print_variable
let print_map_variables = print_map_variables
let map_variables       = map_variables
let map2_variables      = map2_variables
let string_of_variable  = string_of_variable
```


Program Components

– Program components can be uniquely identified by the Dewey notation for trees:



written:

$$\lfloor \texttt{while}\ \lfloor \texttt{true} \rfloor_{00}\ \texttt{do}\ \lfloor \lfloor \texttt{X} \rfloor_{010} := \lfloor 3 \rfloor_{011} \rfloor_{01}\ \texttt{od} \rfloor_{0}$$

$$
\begin{aligned}
\mathrm{Cmp}[\![S\ ;;]\!] &\stackrel{\mathrm{def}}{=} \{\lfloor S\ ;;\rfloor_0\} \cup \mathrm{Cmp}^0[\![S]\!],\\
\mathrm{Cmp}^\pi[\![C_1\ ;\ \ldots\ ;\ C_n]\!] &\stackrel{\mathrm{def}}{=} \{\lfloor C_1\ ;\ \ldots\ ;\ C_n\rfloor_\pi\}\\
&\qquad \cup \bigcup_{i=1}^{n} \mathrm{Cmp}^{\pi.i-1}[\![C_i]\!],\\
\mathrm{Cmp}^\pi[\![\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}]\!] &\stackrel{\mathrm{def}}{=} \{\lfloor \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}\rfloor_\pi\}\\
&\qquad \cup \mathrm{Cmp}^{\pi.0}[\![S_1]\!] \cup \mathrm{Cmp}^{\pi.1}[\![S_2]\!],\\
\mathrm{Cmp}^\pi[\![\texttt{while}\ B\ \texttt{do}\ S_1\ \texttt{od}]\!] &\stackrel{\mathrm{def}}{=} \{\lfloor \texttt{while}\ B\ \texttt{do}\ S_1\ \texttt{od}\rfloor_\pi\}\\
&\qquad \cup \mathrm{Cmp}^{\pi.0}[\![S_1]\!],\\
\mathrm{Cmp}^\pi[\![\texttt{X} := A]\!] &\stackrel{\mathrm{def}}{=} \{\lfloor \texttt{X} := A\rfloor_\pi\},\\
\mathrm{Cmp}^\pi[\![\texttt{skip}]\!] &\stackrel{\mathrm{def}}{=} \{\lfloor \texttt{skip}\rfloor_\pi\}\ .
\end{aligned}
$$

For example:

$\mathrm{Cmp}[\![\texttt{skip}\ ;\ \texttt{skip}\ ;;]\!] = \{\lfloor \texttt{skip}\ ;\ \texttt{skip}\ ;;\rfloor_0, \lfloor \texttt{skip}\rfloor_{00}, \lfloor \texttt{skip}\rfloor_{01}\}$ so that the two occurrences of the same command `skip` within the program `skip ; skip ;;` can be formally distinguished.

Program labelling

– In the abstract syntax, it is assumed that all program components (in $\mathrm{Cmp}[\![P]\!]$) of a program P are uniquely labelled by labels $\ell \in \mathrm{Lab}$ designating program points ($P \in \mathrm{Prog}$):

$\mathrm{at}_P \in \mathrm{Cmp}[\![P]\!] \mapsto \mathrm{Lab},$	$\mathrm{at}_P[\![C]\!]$ is the label before component C
$\mathrm{after}_P \in \mathrm{Cmp}[\![P]\!] \mapsto \mathrm{Lab},$	$\mathrm{after}_P[\![C]\!]$ is the label after component C
$\mathrm{in}_P \in \mathrm{Cmp}[\![P]\!] \mapsto \wp(\mathrm{Lab})$	$\mathrm{in}_P[\![C]\!]$ is the set of labels of the subcomponents of component C

Program components labelling is defined as follows (for short we leave positions implicit, writing C for $\lfloor C\rfloor_\pi$ and assuming that the rules for designating subcomponents of a component are clear from page 65)

$$\forall C \in \mathrm{Cmp}[\![P]\!] : \mathrm{at}_P[\![C]\!] \neq \mathrm{after}_P[\![C]\!]\ . \tag{1}$$

If $C = \texttt{skip} \in \mathrm{Cmp}[\![P]\!]$ or $C = \texttt{X} := A \in \mathrm{Cmp}[\![P]\!]$ then

$$\mathrm{in}_P[\![C]\!] = \{\mathrm{at}_P[\![C]\!], \mathrm{after}_P[\![C]\!]\}\ . \tag{2}$$

If $S = C_1; \ldots ; C_n \in \mathsf{Cmp}[\![P]\!]$ where $n \geq 1$ is a sequence of commands, then

$$
\begin{aligned}
\mathsf{at}_P[\![S]\!] &= \mathsf{at}_P[\![C_1]\!] \\
\mathsf{after}_P[\![S]\!] &= \mathsf{after}_P[\![C_n]\!] \\
\mathsf{in}_P[\![S]\!] &= \bigcup_{i=1}^{n} \mathsf{in}_P[\![C_i]\!] \\
&\forall i \in [1, n[: \mathsf{after}_P[\![C_i]\!] = \mathsf{at}_P[\![C_{i+1}]\!] = \\
&\quad \mathsf{in}_P[\![C_i]\!] \cap \mathsf{in}_P[\![C_{i+1}]\!], \qquad (3) \\
&\forall i, j \in [1, n] : (j \neq i - 1 \wedge j \neq i + 1) \Longrightarrow \\
&\quad (\mathsf{in}_P[\![C_i]\!] \cap \mathsf{in}_P[\![C_j]\!] = \emptyset)
\end{aligned}
$$

If $C = \mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi} \in \mathsf{Cmp}[\![P]\!]$ is a conditional command, then

$$\mathsf{in}_P[\![C]\!] = \{\mathsf{at}_P[\![C]\!], \mathsf{after}_P[\![C]\!]\} \cup \mathsf{in}_P[\![S_t]\!] \cup \mathsf{in}_P[\![S_f]\!]$$

$$\{\mathsf{at}_P[\![C]\!], \mathsf{after}_P[\![C]\!]\} \cap (\mathsf{in}_P[\![S_t]\!] \cup \mathsf{in}_P[\![S_f]\!]) = \emptyset \qquad (4)$$

$$\mathsf{in}_P[\![S_t]\!] \cap \mathsf{in}_P[\![S_f]\!] = \emptyset$$

If $C = \mathtt{while}\ B\ \mathtt{do}\ S\ \mathtt{od} \in \mathsf{Cmp}[\![P]\!]$ is an iteration command, then

$$\mathsf{in}_P[\![C]\!] = \{\mathsf{at}_P[\![C]\!], \mathsf{after}_P[\![C]\!]\} \cup \mathsf{in}_P[\![S]\!] \qquad (5)$$

$$\{\mathsf{at}_P[\![C]\!], \mathsf{after}_P[\![C]\!]\} \cap \mathsf{in}_P[\![S]\!] = \emptyset$$

If $P = S\ \mathtt{;;} \in \mathsf{Cmp}[\![P]\!]$ is a program, then

$$
\begin{aligned}
\mathsf{at}_P[\![P]\!] &= \mathsf{at}_P[\![S]\!], \\
\mathsf{after}_P[\![P]\!] &= \mathsf{after}_P[\![S]\!], \\
\mathsf{in}_P[\![P]\!] &= \mathsf{in}_P[\![S]\!]
\end{aligned}
$$

Abstract Syntax

The abstract syntax is a program representation as a tree structure:

```
(* abstract_Syntax.ml *)
type variable = int
and aexp =
  | NAT of string
  | VAR of variable
  | RANDOM
  | UMINUS of aexp
  | UPLUS of aexp
  | PLUS of aexp * aexp
  | MINUS of aexp * aexp
  | TIMES of aexp * aexp
  | DIV of aexp * aexp
  | MOD of aexp * aexp
```



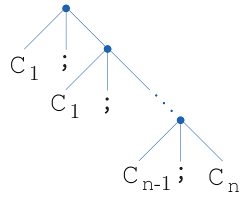
```
and bexp =
  | TRUE
  | FALSE
  | EQ of aexp * aexp
  | LT of aexp * aexp
  | AND of bexp * bexp
  | OR of bexp * bexp
and label = int
and com =
  | SKIP of label * label
  | ASSIGN of label * variable * aexp * label
  | SEQ of label * (com list) * label
  | IF of label * bexp * bexp * com * com * label
  | WHILE of label * bexp * bexp * com * label
```

The first label of `COM of label * ... * label` in command `COM` is the `at` label and the second is the `after` label.

Translation of concrete into abstract syntax

– Identity for arithmetic expressions

– Trees representing sequences of commands are linearized into a list of commands:



$$\Longrightarrow \quad [C_1; C_2; \ldots; C_n]$$

– Boolean expressions are rewritten in equivalent form (up to erroneous behaviors) [19]:

$$
\begin{array}{rcl@{\qquad}rcl}
T(\texttt{true}) & \stackrel{\text{def}}{=} & \texttt{true}, & T(\neg\texttt{true}) & \stackrel{\text{def}}{=} & \texttt{false},\\
T(\texttt{false}) & \stackrel{\text{def}}{=} & \texttt{false}, & T(\neg\texttt{false}) & \stackrel{\text{def}}{=} & \texttt{true},\\
T(A_1 < A_2) & \stackrel{\text{def}}{=} & A_1 < A_2, & T(\neg(A_1 < A_2)) & \stackrel{\text{def}}{=} & T(A_1 >= A_2),\\
T(A_1 <= A_2) & \stackrel{\text{def}}{=} & (A_1 < A_2) \mid (A_1 = A_2), & T(\neg(A_1 <= A_2)) & \stackrel{\text{def}}{=} & T(A_1 > A_2),\\
T(A_1 = A_2) & \stackrel{\text{def}}{=} & A_1 = A_2, & T(\neg(A_1 = A_2)) & \stackrel{\text{def}}{=} & T(A_1 <> A_2),\\
T(A_1 <> A_2) & \stackrel{\text{def}}{=} & (A_1 < A_2) \mid (A_2 < A_1), & T(\neg(A_1 <> A_2)) & \stackrel{\text{def}}{=} & A_1 = A_2,\\
T(A_1 > A_2) & \stackrel{\text{def}}{=} & A_2 < A_1, & T(\neg(A_1 > A_2)) & \stackrel{\text{def}}{=} & T(A_1 <= A_2),\\
T(A_1 >= A_2) & \stackrel{\text{def}}{=} & (A_1 = A_2) \mid (A_2 < A_1), & T(\neg(A_1 >= A_2)) & \stackrel{\text{def}}{=} & A_1 < A_2,\\
T(B_1 \mid B_2) & \stackrel{\text{def}}{=} & T(B_1) \mid T(B_2) & T(\neg(B_1 \mid B_2)) & \stackrel{\text{def}}{=} & T(\neg(B_1)) \,\&\, T(\neg(B_2)),\\
T(B_1 \,\&\, B_2) & \stackrel{\text{def}}{=} & T(B_1) \,\&\, T(B_2), & T(\neg(B_1 \,\&\, B_2)) & \stackrel{\text{def}}{=} & T(\neg(B_1)) \mid T(\neg(B_2)),\\
 & & & T(\neg(\neg(B))) & \stackrel{\text{def}}{=} & T(B).
\end{array}
$$

[19] This is only to greatly simplify the design of the abstract interpreter.

Translation from concrete to abstract syntax

```
(* concrete_To_Abstract_Syntax.mli *)
open Abstract_Syntax
(* abstract syntax *)
type variable = Abstract_Syntax.variable
and aexp = Abstract_Syntax.aexp
and bexp = Abstract_Syntax.bexp
and label = Abstract_Syntax.label
and com = Abstract_Syntax.com
(* concrete syntax *)
type c_bexp =
 | C_TRUE
 | C_FALSE
 | C_LT of aexp * aexp
 | C_LEQ of aexp * aexp
```



```
 | C_EQ of aexp * aexp
 | C_NEQ of aexp * aexp
 | C_GT of aexp * aexp
 | C_GEQ of aexp * aexp
 | C_OR of c_bexp * c_bexp
 | C_AND of c_bexp * c_bexp
 | C_NEG of c_bexp
type c_com =
 | C_SKIP
 | C_ASSIGN of variable * aexp
 | C_SEQ of c_com list
 | C_IF of c_bexp * c_com * c_com
 | C_WHILE of c_bexp * c_com

(* normalization of boolean expressions *)
val tbexp    : c_bexp -> bexp
    val tnotbexp : c_bexp -> bexp

```



```
(* labels *)
val at               : com   -> label        (* command entry label *)
val after            : com   -> label        (* command exit label  *)
val incom            : label -> com -> bool  (* label in command    *)
val number_of_labels : unit  -> int
val entry            : unit  -> label        (* program entry label *)
val exit             : unit  -> label        (* program exit label  *)
val print_label      : label -> unit
val string_of_label  : label -> string

(* program labelling *)
val label_normalize_com : c_com -> com
```



```
(* concrete_To_Abstract_Syntax.ml *)
open Abstract_Syntax
(* abstract syntax *)
type variable = Abstract_Syntax.variable
and aexp = Abstract_Syntax.aexp
and bexp = Abstract_Syntax.bexp
and label = Abstract_Syntax.label
and com = Abstract_Syntax.com
(* concrete syntax *)
type c_bexp =
| C_TRUE
| C_FALSE
| C_LT of aexp * aexp
| C_LEQ of aexp * aexp
| C_EQ of aexp * aexp
| C_NEQ of aexp * aexp
| C_GT of aexp * aexp
```

```
| C_GEQ of aexp * aexp
| C_OR of c_bexp * c_bexp
| C_AND of c_bexp * c_bexp
| C_NEG of c_bexp
type c_com =
| C_SKIP
| C_ASSIGN of variable * aexp
| C_SEQ of c_com list
| C_IF of c_bexp * c_com * c_com
| C_WHILE of c_bexp * c_com

(* normalization of boolean expressions *)
let rec tbexp b = match b with
| C_TRUE            -> TRUE
| C_FALSE           -> FALSE
| (C_LT  (v1, v2)) -> (LT (v1,v2))
| (C_LEQ (v1, v2)) -> (OR ((LT (v1,v2)),(EQ (v1,v2))))
| (C_EQ  (v1, v2)) -> (EQ (v1,v2))
```



```
| (C_NEQ (v1, v2)) -> (OR ((LT (v1,v2)),(LT (v2,v1))))
| (C_GT  (v1, v2)) -> (LT (v2,v1))
| (C_GEQ (v1, v2)) -> (OR ((LT (v2,v1)),(EQ (v1,v2))))
| (C_OR (b1,b2))   -> (OR ((tbexp b1),(tbexp b2)))
| (C_AND (b1,b2))  -> (AND ((tbexp b1),(tbexp b2)))
| (C_NEG b')       -> tnotbexp b'
and tnotbexp b = match b with
| C_TRUE            -> FALSE
| C_FALSE           -> TRUE
| (C_LT  (v1, v2)) -> tbexp (C_GEQ (v1, v2))
| (C_LEQ (v1, v2)) -> tbexp (C_GT (v1, v2))
| (C_EQ  (v1, v2)) -> tbexp (C_NEQ (v1, v2))
| (C_NEQ (v1, v2)) -> tbexp (C_EQ (v1, v2))
| (C_GT  (v1, v2)) -> tbexp (C_LEQ (v1, v2))
| (C_GEQ (v1, v2)) -> tbexp (C_LT (v1, v2))
| (C_OR  (b1,b2))  -> (AND ((tnotbexp b1),(tnotbexp b2)))
| (C_AND (b1,b2))  -> (OR ((tnotbexp b1),(tnotbexp b2)))
| (C_NEG b')       -> tbexp b'
```



```

(* variables of arithmetic expressions *)
let leq x y = x <= y
let rec reduce l = match l with
  (* eliminate duplicates in sorted list of variables *)
| [] -> []
| [v] -> [v]
|  h1 :: h2 :: t -> let q = (reduce (h2 :: t)) in
            if (h1 = h2) then q else h1 :: q
let rec varaexp a = match a with
| (NAT i)          -> []
| (VAR v)          -> [v]
| RANDOM           -> []
| (UMINUS a1)      -> (varaexp a1)
| (UPLUS  a1)      -> (varaexp a1)
| (PLUS  (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (MINUS (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (TIMES (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
```



```
| (DIV   (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (MOD   (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
(* variables of boolean expressions *)
let rec varbexp b = match b with
| C_TRUE           -> []
| C_FALSE          -> []
| (C_LT  (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (C_LEQ (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (C_EQ  (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (C_NEQ (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (C_GT  (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (C_GEQ (a1, a2)) -> reduce (Sort.merge leq (varaexp a1) (varaexp a2))
| (C_OR  (b1,b2))  -> reduce (Sort.merge leq (varbexp b1) (varbexp b2))
| (C_AND (b1,b2))  -> reduce (Sort.merge leq (varbexp b1) (varbexp b2))
| (C_NEG b')       -> (varbexp b')

(* program labelling *)
let at c = match c with
```

```
| SKIP (l,m)           -> l
| ASSIGN (l,x,a,m)     -> l
| SEQ (l,s,m)          -> l
| IF (l,b,nb,st,sf,m)  -> l
| WHILE (l,b,nb,s,m)   -> l
let after c = match c with
| SKIP (l,m)           -> m
| ASSIGN (l,x,a,m)     -> m
| SEQ (l,s,m)          -> m
| IF (l,b,nb,st,sf,m)  -> m
| WHILE (l,b,nb,s,m)   -> m
let rec incom l c = match c with
| SKIP (l1,l2)          -> (l=l1) or (l=l2)
| ASSIGN (l1,x,a,l2)    -> (l=l1) or (l=l2)
| SEQ (l1,s,l2)         -> (l=l1) or (l=l2) or (inseq l s)
| IF (l1,b,nb,st,sf,l2) -> (l=l1) or (l=l2) or (incom l st)
                                      or (incom l sf)
| WHILE (l1,b,nb,s,l2)  -> (l=l1) or (l=l2) or (incom l s)
```



```
and inseq l s = match s with
| [] -> false
| h::t -> (incom l h) or (inseq l t)
exception Error_label_normalize_com of string
let rec label_normalize_seq_from l s = match s with
| []  -> raise (Error_label_normalize_com "empty sequence of commands")
| [c] -> let m, c' = label_normalize_com_from l c in
        m, [c']
| h :: t -> let m, h' = label_normalize_com_from l h in
        let n, t' = label_normalize_seq_from m t in
          n, (h' :: t')
and label_normalize_com_from l c = match c with
| C_SKIP           -> l+1, (SKIP (l, (l+1)))
| (C_ASSIGN (v,a)) -> l+1, (ASSIGN (l, v, a, (l+1)))
| (C_IF (b,t,f))   -> let m, t' = label_normalize_com_from (l+1) t in
             let n, f' = label_normalize_com_from (m+1) f in
               n+1, (IF (l, (tbexp b), (tnotbexp b), t', f', (n+1)))
| (C_SEQ s)        -> let m, s' = label_normalize_seq_from l s in
```



```
             m, (SEQ (l, s', m))
| (C_WHILE (b,c))  -> let m, c' = label_normalize_com_from (l+1) c in
             m+1, (WHILE  (l, (tbexp b), (tnotbexp b), c', (m+1)))
let last_label = ref 0
let label_normalize_com c =
  let m, c' = label_normalize_com_from 0 c in
   last_label := m;
   c'
let number_of_labels () = (!last_label + 1)
let entry () = 0
let exit () = !last_label
let print_label l = (print_int l)
let string_of_label l = (string_of_int l)
```


Labels

Provide an abstract view of the abstract syntax where the operations on labels are visible (but not their internal implementation):

```
(* labels.mli *)
open Abstract_Syntax
(* labels *)
val at               : com   -> label       (* command entry label *)
val after            : com   -> label       (* command exit label  *)
val incom            : label -> com -> bool (* label in command    *)
val number_of_labels : unit  -> int
val entry            : unit  -> label       (* program entry label *)
val exit             : unit  -> label       (* program exit label  *)
val print_label      : label -> unit
val string_of_label  : label -> string
```

See `labels.ml` for the immediate implementation.

Lexer

```
{ (* from file lexer.mll (lexical analysis) *)
open Parser             (* The type token is defined in parser.mli *)
exception Eof
}
rule token = parse
  [' ' '\t' '\n' '\r'] { token lexbuf }
| ( '%' [^'%']* '%')   { token lexbuf }
| ['0'-'9']+           { (T_NAT (Lexing.lexeme lexbuf)) }
| '('                  { T_LPAR }
| ')'                  { T_RPAR }
| '?'                  { T_RANDOM }
| '+'                  { T_PLUS }
| '-'                  { T_MINUS }
| '*'                  { T_TIMES }
```



```
| '/'                  { T_DIV }
| "mod"                { T_MOD }
| "true"               { T_TRUE }
| "false"              { T_FALSE }
| '<'                  { T_LT }
| "<="                 { T_LEQ }
| '='                  { T_EQ }
| "<>"                 { T_NEQ }
| '>'                  { T_GT }
| ">="                 { T_GEQ }
| '¬'                  { T_NEG }
| '|'                  { T_OR }
| '&'                  { T_AND }
| "skip"               { T_SKIP }
| ":="                 { T_ASSIGN }
| ';'                  { T_SEQ }
| "if"                 { T_IF }
| "then"               { T_THEN }
```



```
| "else"               { T_ELSE }
| "fi"                 { T_FI }
| "while"              { T_WHILE }
| "do"                 { T_DO }
| "od"                 { T_OD }
| (['a'-'z'] | ['A'-'Z'])(['a'-'z'] | ['A'-'Z'] | ['0'-'9'])*
                       { (T_VAR (Lexing.lexeme lexbuf)) }
| ";;"                 { T_EOP }
| eof                  { raise Eof }
```


Parser

```
/* file parser.mly, parsing & concrete to abstract syntax translation */

%{ (* header *)
(* make a sequence of commands from two commands or *)
(* subsequences with flattening of the subsequences *)
let makeseq l r = match l, r with
  | Concrete_To_Abstract_Syntax.C_SEQ h,
    Concrete_To_Abstract_Syntax.C_SEQ t
    -> Concrete_To_Abstract_Syntax.C_SEQ (h @ t)
  | h, Concrete_To_Abstract_Syntax.C_SEQ t
    -> Concrete_To_Abstract_Syntax.C_SEQ (h :: t)
  | Concrete_To_Abstract_Syntax.C_SEQ h, t
    -> Concrete_To_Abstract_Syntax.C_SEQ (h @ [t])
  | h,t
```

```
    -> Concrete_To_Abstract_Syntax.C_SEQ [h; t]

%} /* declarations */

%token <string> T_NAT
%token T_LPAR T_RPAR
%token T_RANDOM T_PLUS T_MINUS T_TIMES T_DIV T_MOD
%token T_TRUE T_FALSE T_LT T_LEQ T_EQ T_NEQ T_GT T_GEQ T_OR T_AND T_NEG
%token T_SKIP T_ASSIGN T_SEQ T_IF T_THEN T_ELSE T_FI T_WHILE T_DO T_OD
%token T_AINITIAL T_FINAL T_ALWAYS T_SOMETIME
%token <string> T_VAR
%token T_EOP

%start n_Prog /* grammar axiom non terminal */
%type <Abstract_Syntax.com> n_Prog /* program */
%type <Concrete_To_Abstract_Syntax.c_com>  n_Lco /* list of commands */
%type <Concrete_To_Abstract_Syntax.c_com>  n_Com /* command          */
%type <Concrete_To_Abstract_Syntax.c_bexp> n_Bexp /* boolean expr.   */
```



```
%type <Abstract_Syntax.aexp> n_Aexp /* arithmetic expression         */

%left     T_OR              /* lowest precedence */
%left     T_AND
%right    T_NEG
%nonassoc T_LT T_LEQ T_EQ T_NEQ T_GT T_GEQ
%left     T_PLUS T_MINUS
%left     T_TIMES T_DIV T_MOD
%right    T_UPLUS T_UMINUS   /* highest precedence */

%% /* grammar rules */

n_Aexp:
  T_RANDOM                   { Abstract_Syntax.RANDOM            }
| T_NAT                      { (Abstract_Syntax.NAT $1)          }
| T_VAR { (Abstract_Syntax.VAR (Symbol_Table.add_symb_table $1)) }
| n_Aexp  T_PLUS   n_Aexp  { (Abstract_Syntax.PLUS  ($1, $3))    }
| n_Aexp  T_MINUS  n_Aexp  { (Abstract_Syntax.MINUS ($1, $3))    }
```



```
| n_Aexp  T_TIMES  n_Aexp  { (Abstract_Syntax.TIMES ($1, $3))    }
| n_Aexp  T_DIV    n_Aexp  { (Abstract_Syntax.DIV   ($1, $3))    }
| n_Aexp  T_MOD    n_Aexp  { (Abstract_Syntax.MOD   ($1, $3))    }
| T_PLUS  n_Aexp %prec T_UPLUS  { (Abstract_Syntax.UPLUS  $2)    }
| T_MINUS n_Aexp %prec T_UMINUS { (Abstract_Syntax.UMINUS $2)    }
| T_LPAR  n_Aexp  T_RPAR        { $2               }
;

n_Bexp:
  T_TRUE               { Concrete_To_Abstract_Syntax.C_TRUE           }
| T_FALSE              { Concrete_To_Abstract_Syntax.C_FALSE          }
| n_Aexp T_LT  n_Aexp { (Concrete_To_Abstract_Syntax.C_LT  ($1, $3)) }
| n_Aexp T_LEQ n_Aexp { (Concrete_To_Abstract_Syntax.C_LEQ ($1, $3)) }
| n_Aexp T_EQ  n_Aexp { (Concrete_To_Abstract_Syntax.C_EQ  ($1, $3)) }
| n_Aexp T_NEQ n_Aexp { (Concrete_To_Abstract_Syntax.C_NEQ ($1, $3)) }
| n_Aexp T_GT  n_Aexp { (Concrete_To_Abstract_Syntax.C_GT  ($1, $3)) }
| n_Aexp T_GEQ n_Aexp { (Concrete_To_Abstract_Syntax.C_GEQ ($1, $3)) }
| n_Bexp T_OR  n_Bexp { (Concrete_To_Abstract_Syntax.C_OR  ($1, $3)) }
```



```
| n_Bexp T_AND n_Bexp { (Concrete_To_Abstract_Syntax.C_AND ($1, $3)) }
| T_NEG n_Bexp        { (Concrete_To_Abstract_Syntax.C_NEG  $2)      }
| T_LPAR  n_Bexp T_RPAR { $2               }
;

n_Com:
  T_SKIP           { Concrete_To_Abstract_Syntax.C_SKIP              }
| T_VAR T_ASSIGN n_Aexp
                   { (Concrete_To_Abstract_Syntax.C_ASSIGN
                              ((Symbol_Table.add_symb_table $1), $3)) }
| T_IF n_Bexp T_THEN n_Lco  T_ELSE n_Lco T_FI
                   { (Concrete_To_Abstract_Syntax.C_IF ($2, $4, $6))  }
| T_WHILE n_Bexp T_DO n_Lco T_OD
                   { (Concrete_To_Abstract_Syntax.C_WHILE ($2, $4))   }
;

n_Lco:
     n_Com           { $1              }
```

```
| n_Com T_SEQ n_Lco  { (makeseq $1 $3) }
;

n_Prog:
     N_init n_Lco T_EOP
              { (Concrete_To_Abstract_Syntax.label_normalize_com $2) }
;

N_init:
     { Symbol_Table.init_symb_table () }

%%
(* trailer *)
```


Converting the program file into abstract syntax

The lexer and parser are called on the program input file to get the abstract syntax of the program:

```
(* program_To_Abstract_Syntax.mli *)
open Abstract_Syntax
(* parsing and concrete to abstract syntax translation *)
exception Syntax_Error
val abstract_syntax_of_program : string -> com
```



```
(* program_To_Abstract_Syntax.ml *)
open Lexing
open Abstract_Syntax
exception Syntax_Error
let abstract_syntax_of_program f =
   let input_channel = if f = "" then stdin else open_in f in
      try
          let lexbuf = Lexing.from_channel input_channel  in
            (Parser.n_Prog Lexer.token lexbuf)
      with
          | Failure s -> print_string s; print_newline ();
                         flush stdout;
                         raise Syntax_Error
          | Lexer.Eof -> print_string "lexical error\n";
                         flush stdout;
                         raise Syntax_Error
          | Parsing.Parse_error ->
```



```
                         print_string "syntax error\n";
                         flush stdout;
                         raise Syntax_Error
```

Program pretty-printing

```
(* pretty_Print.mli *)
open Abstract_Syntax
val pretty_print : com -> unit
```

See `pretty_Print.ml` for the implementation.

Reading and pretty-printing the program

```
(* main.ml *)
open Program_To_Abstract_Syntax
open Pretty_Print
let _ =
  let arg = Sys.argv.(1) in
    let p = (abstract_syntax_of_program arg) in
      pretty_print p
```


Compilation makefile

```
# makefile

SOURCES = \
symbol_Table.mli \
symbol_Table.ml \
variables.mli \
variables.ml \
abstract_Syntax.ml \
concrete_To_Abstract_Syntax.mli \
concrete_To_Abstract_Syntax.ml \
labels.mli \
labels.ml \
parser.mli \
parser.ml \
```



```
lexer.ml \
program_To_Abstract_Syntax.mli \
program_To_Abstract_Syntax.ml \
pretty_Print.mli \
pretty_Print.ml \
main.ml

.PHONY : help
help :
	@echo ""
	@echo "make help          : this help"
	@echo "make compile       : compile"
	@echo "./a.out filename   : execute"
	@echo "make examples      : run examples"
	@echo "make clean         : remove auxiliary files"
	@echo ""

.PHONY : compile
```

```
compile:
   ocamlyacc parser.mly
   ocamllex lexer.mll
   ocamlc $(SOURCES)

.PHONY : examples
examples :
   ./a.out ../Examples/example01.sil

.PHONY : clean
clean :
   /bin/rm -f *.cmi *.cmo *~ a.out lexer.ml parser.ml

.PHONY : delete
delete : clean
   /bin/rm -f parser.mli
```


Example

```
Script started on Sat Feb 26 12:27:11 2005
% make compile
ocamlyacc parser.mly
ocamllex lexer.mll
62 states, 3001 transitions, table size 12376 bytes
ocamlc symbol_Table.mli symbol_Table.ml variables.mli variables.ml
abstract_Syntax.ml concrete_To_Abstract_Syntax.mli
concrete_To_Abstract_Syntax.ml labels.mli labels.ml parser.mli
parser.ml lexer.ml program_To_Abstract_Syntax.mli
program_To_Abstract_Syntax.ml pretty_Print.mli pretty_Print.ml
main.ml
% ./a.out ../Examples/example1.sil

0:
```



```
  x := 1;
1:
  while (x < 100) do
    2:
      x := (x + 1)
    3:
  od {((100 < x) | (x = 100))}
4:

% make clean
/bin/rm -f *.cmi *.cmo *~ a.out lexer.ml parser.mli parser.ml
% ^Dexit
Script done on Sat Feb 26 12:27:52 2005
```


Operational Semantics of the Simple Imperative Language (SIL)

Values

Basic values are bounded machine integers:

$$
\begin{array}{ll}
\texttt{max\_int} > 9, & \text{greatest machine integer;} \\
\texttt{min\_int} \stackrel{\text{def}}{=} -\texttt{max\_int} - 1, & \text{smallest machine integer;} \\
z \in \mathbb{Z}, & \text{mathematical integers;} \\
i \in \mathbb{I} \stackrel{\text{def}}{=} [\texttt{min\_int}, \texttt{max\_int}], & \text{bounded machine integers.}
\end{array} \tag{6}
$$

Errors

The semantics keeps track of uninitialized variables (e.g. by means of a reserved value) and of arithmetic errors (overflow, division by zero, ..., e.g. by means of exceptions):

$$
\begin{array}{ll}
\Omega_{\mathrm{i}}, & \text{initialization error;} \\
\Omega_{\mathrm{a}}, & \text{arithmetic error;} \\
e \in \mathbb{E} \stackrel{\text{def}}{=} \{\Omega_{\mathrm{i}}, \Omega_{\mathrm{a}}\}, & \text{errors;} \\
v \in \mathbb{I}_\Omega \stackrel{\text{def}}{=} \mathbb{I} \cup \mathbb{E}, & \text{machine values.}
\end{array} \tag{7}
$$

Concrete environments

An environment ρ records the value $\rho(\mathtt{X})$ of program variables $\mathtt{X} \in \mathbb{V}$.

$$\rho \in \mathbb{R} \stackrel{\text{def}}{=} \mathbb{V} \mapsto \mathbb{I}_\Omega, \qquad \text{environments.}$$

Assignment/substitution notation ($f \in D \mapsto E$):

$$
\begin{array}{rcll}
f[d := e](x) & \stackrel{\text{def}}{=} & f(x), & \text{if } x \neq d\,; \\
f[d := e](d) & \stackrel{\text{def}}{=} & e\,; & \\
f[d_1 := e_1; d_2 := e_2; \ldots; d_n := e_n] & \stackrel{\text{def}}{=} & (f[d_1 := e_1])[d_2 := e_2; \ldots; d_n := e_n]\,. &
\end{array} \tag{8}
$$

Machine arithmetic: numbers

$\underline{\mathrm{n}} \in \mathbb{I}_\Omega$: machine natural number
$\mathrm{n} \in \mathbb{N}$: corresponding mathematical natural number

Decimal notation ($\mathrm{d} \in \mathrm{Digit}$, $\mathrm{n} \in \mathrm{Nat}$):

$$
\begin{array}{lll}
\underline{\mathrm{d}} \stackrel{\text{def}}{=} \mathrm{d}; & & \\
\underline{\mathrm{nd}} \stackrel{\text{def}}{=} \Omega_{\mathrm{a}}, & & \text{if } 10\,\underline{\mathrm{n}} + \mathrm{d} > \texttt{max\_int}; \\
\underline{\mathrm{nd}} \stackrel{\text{def}}{=} 10\,\underline{\mathrm{n}} + \mathrm{d}, & & \text{if } 10\,\underline{\mathrm{n}} + \mathrm{d} \leq \texttt{max\_int}\,.
\end{array}
$$

Machine arithmetic: unary operators

For unary arithmetic operators $\mathrm{u} \in \{+, -\}$:

$$\begin{array}{ll} \underline{\mathrm{u}} \in \mathbb{I}_\Omega \mapsto \mathbb{I}_\Omega & : \text{machine arithmetic operation} \\ \mathrm{u} \in \mathbb{Z} \mapsto \mathbb{Z} & : \text{corresponding mathematical operation} \end{array}$$

Error when the mathematical result is not machine-representable ($e \in \mathbb{E}$, $i \in \mathbb{I}$):

$$\begin{array}{lll} \underline{\mathrm{u}}\, \Omega_e \stackrel{\text{def}}{=} \Omega_e; & & \\ \underline{\mathrm{u}}\, i \stackrel{\text{def}}{=} \mathrm{u}\, i, & \text{if } \mathrm{u}\, i \in \mathbb{I}; & (9) \\ \underline{\mathrm{u}}\, i \stackrel{\text{def}}{=} \Omega_\mathrm{a}, & \text{if } \mathrm{u}\, i \notin \mathbb{I}\,. & \end{array}$$

Machine arithmetic: binary operators

– For binary arithmetic operators $\mathrm{b} \in \{+, -, *, /, \mathrm{mod}\}$:

$$\begin{array}{ll} \underline{\mathrm{b}} \in \mathbb{I}_\Omega \times \mathbb{I}_\Omega \mapsto \mathbb{I}_\Omega & : \text{machine arithmetic operation} \\ \mathrm{b} \in \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z} & : \text{corresponding math. operation} \end{array}$$

– division and modulo are defined only for non-negative first argument and positive second argument

We have ($\mathbb{N}_+$ is the set of positive naturals, $e \in \mathbb{E}$, $v \in \mathbb{I}_\Omega$, $i, i_1, i_2 \in \mathbb{I}$)

$$\begin{array}{lll} \Omega_e\, \underline{\mathrm{b}}\, v \stackrel{\text{def}}{=} \Omega_e; & & \\ i\, \underline{\mathrm{b}}\, \Omega_e \stackrel{\text{def}}{=} \Omega_e; & & \\ i_1\, \underline{\mathrm{b}}\, i_2 \stackrel{\text{def}}{=} i_1\, \mathrm{b}\, i_2, & \text{if } \mathrm{b} \in \{+, -, *\} \wedge i_1\, \mathrm{b}\, i_2 \in \mathbb{I}; & (10) \\ i_1\, \underline{\mathrm{b}}\, i_2 \stackrel{\text{def}}{=} i_1\, \mathrm{b}\, i_2, & \text{if } \mathrm{b} \in \{/, \mathrm{mod}\} \wedge i_1 \in \mathbb{I} \cap \mathbb{N} \wedge & (11) \\ & \quad i_2 \in \mathbb{I} \cap \mathbb{N}_+ \wedge i_1\, \mathrm{b}\, i_2 \in \mathbb{I}; & \\ i_1\, \underline{\mathrm{b}}\, i_2 \stackrel{\text{def}}{=} \Omega_\mathrm{a}, & \text{if } i_1\, \mathrm{b}\, i_2 \notin \mathbb{I} \vee (\mathrm{b} \in \{/, \mathrm{mod}\} \wedge & (12) \\ & \quad (i_1 \notin \mathbb{I} \cap \mathbb{N} \vee i_2 \notin \mathbb{I} \cap \mathbb{N}_+))\,. & \end{array}$$

Operational semantics of arithmetic expressions

The big-step operational semantics [7] [20] of arithmetic expressions involves judgements:

$$\rho \vdash A \Rightarrow v$$

meaning that in environment ρ, the arithmetic expression A may evaluate to $v \in \mathbb{I}_\Omega$.

Reference

[7] G.D. Plotkin. A structural approach to operational semantics. Tech. rep. DAIMI FN-19, Aarhus University, Denmark, Sep. 1981.

[8] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pp. 237–258. Elsevier, 1988.

[20] renamed natural semantics by [8].

$$\rho \vdash \mathrm{n} \Mapsto \underline{\mathrm{n}} \qquad \text{decimal numbers;} \tag{13}$$

$$\rho \vdash \mathrm{X} \Mapsto \rho(\underline{\mathrm{X}}) \qquad \text{variables;} \tag{14}$$

$$\frac{i \in \mathbb{I}}{\rho \vdash \texttt{?} \Mapsto i} \qquad \text{random;} \tag{15}$$

$$\frac{\rho \vdash A \Mapsto v}{\rho \vdash \mathrm{u}\, A \Mapsto \underline{\mathrm{u}}\, v} \qquad \text{unary arithmetic operations;}^{21} \tag{16}$$

$$\frac{\rho \vdash A_1 \Mapsto v_1 \quad \rho \vdash A_2 \Mapsto v_2}{\rho \vdash A_1 \mathrm{b}\, A_2 \Mapsto v_1\, \underline{\mathrm{b}}\, v_2} \qquad \text{binary arithmetic operations.} \tag{17}$$

[21] Observe that if m and M are the strings of digits respectively representing the absolute value of min_int and max_int then $\underline{\mathrm{m}} >$ max_int so that $\rho \vdash \mathrm{m} \Mapsto \Omega_a$ whence $\rho \vdash$ - m $\Mapsto \Omega_a$. However $\rho \vdash$ (- M) - 1 $\Mapsto$ min_int.

Machine booleans

$$\begin{aligned} \mathbb{B} &: \text{logical boolean values} \\ \mathbb{B}_\Omega &: \text{machine truth values}^{22} \end{aligned}$$

$$\begin{aligned} \mathbb{B} &\stackrel{\text{def}}{=} \{\mathtt{tt}, \mathtt{ff}\} \\ \mathbb{B}_\Omega &\stackrel{\text{def}}{=} \mathbb{B} \cup \mathbb{E} \end{aligned}$$

[22] including errors $\mathbb{E} = \{\Omega_i, \Omega_a\}$.

Arithmetic comparison

For the binary arithmetic comparison operators:

$$\mathrm{c} \in \{<, <=, =, <>, >=, >\}$$

$$\begin{aligned} \underline{\mathrm{c}} \in \mathbb{I}_\Omega \times \mathbb{I}_\Omega \mapsto \mathbb{B}_\Omega &: \text{machine arithmetic comparison} \\ \mathrm{c} \in \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{B} &: \text{mathematical comparison operation} \end{aligned}$$

Evaluation of operands, whence error propagation is left to right ($e \in \mathbb{E}$, $v \in \mathbb{I}_\Omega$, $i, i_1, i_2 \in \mathbb{I}$):

$$\begin{aligned} \Omega_e\, \underline{\mathrm{c}}\, v &\stackrel{\text{def}}{=} \Omega_e, \\ i\, \underline{\mathrm{c}}\, \Omega_e &\stackrel{\text{def}}{=} \Omega_e, \\ i_1\, \underline{\mathrm{c}}\, i_2 &\stackrel{\text{def}}{=} i_1\, \mathrm{c}\, i_2\, . \end{aligned} \tag{18}$$

Unary boolean operations

Boolean unary operators $\mathrm{u} \in \{\neg\}$:

$$\begin{aligned} \underline{\mathrm{u}} \in \mathbb{B}_\Omega \mapsto \mathbb{B}_\Omega &: \text{machine boolean operation} \\ \mathrm{u} \in \mathbb{B} \mapsto \mathbb{B} &: \text{mathematical operation} \end{aligned}$$

Errors are propagated, so that we have ($e \in \mathbb{E}$, $b \in \mathbb{B}$):

$$\begin{aligned} \underline{\mathrm{u}}\, \Omega_e &\stackrel{\text{def}}{=} \Omega_e, \\ \underline{\mathrm{u}}\, b &\stackrel{\text{def}}{=} \mathrm{u}\, b\, . \end{aligned}$$

Binary boolean operations

Binary boolean operators $\mathsf{b} \in \{\&, |\}$:

$$\underline{\mathsf{b}} \in \mathbb{B}_\Omega \times \mathbb{B}_\Omega \mapsto \mathbb{B}_\Omega \quad : \text{machine boolean operation}$$
$$\mathrm{b} \in \mathbb{B} \times \mathbb{B} \mapsto \mathbb{B} \quad : \text{mathematical boolean operation}$$

Evaluation of operands, whence error propagation is left to right ($e \in \mathbb{E}$, $w \in \mathbb{B}_\Omega$, $b, b_1, b_2 \in \mathbb{B}$):

$$\begin{aligned} \Omega_e \,\underline{\mathsf{b}}\, w &\stackrel{\text{def}}{=} \Omega_e, \\ b \,\underline{\mathsf{b}}\, \Omega_e &\stackrel{\text{def}}{=} \Omega_e, \\ b_1 \,\underline{\mathsf{b}}\, b_2 &\stackrel{\text{def}}{=} b_1 \,\mathrm{b}\, b_2 \;. \end{aligned} \tag{19}$$

Operational semantics of boolean expressions

A judgement $\rho \vdash B \Rrightarrow b$ means that in environment ρ, the boolean expression b may evaluate to $b \in \mathbb{B}_\Omega$:

$\rho \vdash \mathtt{true} \Rrightarrow \mathtt{tt}$	a truth;	(20)
$\rho \vdash \mathtt{false} \Rrightarrow \mathtt{ff}$	a falsity;	(21)
$\dfrac{\rho \vdash A_1 \Rrightarrow v_1 \quad \rho \vdash A_2 \Rrightarrow v_2}{\rho \vdash A_1 \,\mathsf{c}\, A_2 \Rrightarrow v_1 \,\underline{\mathsf{c}}\, v_2}$	arithmetic comparisons	(22)
$\dfrac{\rho \vdash B \Rrightarrow w}{\rho \vdash \mathsf{u}\, B \Rrightarrow \underline{\mathsf{u}}\, w}$	unary boolean operations	
$\dfrac{\rho \vdash B_1 \Rrightarrow w_1 \quad \rho \vdash B_2 \Rrightarrow w_2}{\rho \vdash B_1 \,\mathsf{b}\, B_2 \Rrightarrow w_1 \,\underline{\mathsf{b}}\, w_2}$	binary boolean operations	

Semantics of normalized boolean expressions

The semantics of a boolean expression B may not be the same as the semantics of its transformed form $T(B)$, but only in case of error [23]:

$$\begin{gathered} \forall b \in \mathbb{B} : \rho \vdash B \Rrightarrow b \iff \rho \vdash T(B) \Rrightarrow b, \\ (\exists e \in \mathbb{E} : \rho \vdash B \Rrightarrow e) \iff (\exists e' \in \mathbb{E} : \rho \vdash T(B) \Rrightarrow e') \;. \end{gathered}$$

[23] e.g. the rewriting rule $T(A_1 > A_2) = A_2 < A_1$ does not respect left to right evaluation whence the error propagation order.

Environments

During execution of program $P \in \mathrm{Prog}$, an environment $\rho \in \mathrm{Env}[\![P]\!] \subseteq \mathbb{R}$ maps program variables $\mathtt{X} \in \mathrm{Var}[\![P]\!]$ to their value $\rho(\mathtt{X})$:

$$\begin{aligned} \mathrm{Env} &\in \mathrm{Prog} \mapsto \wp(\mathbb{R}), \\ \mathrm{Env}[\![P]\!] &\stackrel{\text{def}}{=} \mathrm{Var}[\![P]\!] \mapsto \mathbb{I}_\Omega \;. \end{aligned}$$

Programs states

States $\langle \ell,\ \rho \rangle \in \Sigma[\![P]\!]$ record a program point $\ell \in \mathrm{in}_P[\![P]\!]$ and an environment $\rho \in \mathrm{Env}[\![P]\!]$ assigning values to variables:

$$\begin{aligned} \Sigma &\in \mathrm{Prog} \mapsto \wp(\mathbb{V} \times \mathbb{R}), \\ \Sigma[\![P]\!] &\stackrel{\mathrm{def}}{=} \mathrm{in}_P[\![P]\!] \times \mathrm{Env}[\![P]\!]\ . \end{aligned} \tag{23}$$

Small-step operational semantics of commands

The small-step operational semantics [7] of commands, sequences and programs $C \in \mathrm{Com} \cup \mathrm{Seq} \cup \mathrm{Prog}$ within a program $P \in \mathrm{Prog}$ involves transition judgements [24]

$$\langle \ell,\ \rho \rangle \Longmapsto\!\![\![C]\!]\!\!\Longrightarrow \langle \ell',\ \rho' \rangle\ .$$

[24] Such judgements mean that if execution is at control point $\ell \in \mathrm{in}_P[\![C]\!]$ in environment $\rho \in \mathrm{Env}[\![P]\!]$ then the next computation step within command C leads to program control point $\ell' \in \mathrm{in}_P[\![C]\!]$ in the new environment $\rho' \in \mathrm{Env}[\![P]\!]$.

Small-step operational semantics of commands and programs

Identity $C = \mathtt{skip}$ $(\mathrm{at}_P[\![C]\!] = \ell$ and $\mathrm{after}_P[\![C]\!] = \ell')$

$$\langle \ell,\ \rho \rangle \Longmapsto\!\![\![\mathtt{skip}]\!]\!\!\Longrightarrow \langle \ell',\ \rho \rangle \tag{24}$$

Assignment [25] $C = \mathtt{X} := A$ $(\mathrm{at}_P[\![C]\!] = \ell$ and $\mathrm{after}_P[\![C]\!] = \ell')$

$$\frac{\rho \vdash A \Rrightarrow i}{\langle \ell,\ \rho \rangle \Longmapsto\!\![\![\mathtt{X} := A]\!]\!\!\Longrightarrow \langle \ell',\ \rho[\mathtt{X} := i] \rangle},\ i \in \mathbb{I} \tag{25}$$

[25] According to axiom schema (25), program execution is blocked in error state at the assignment $\mathtt{X} := A$ if the arithmetic expression A evaluates to an error, i.e. $\rho \vdash A \Rrightarrow \Omega_e$, $e \in \mathbb{E}$. This option corresponds to an implementation where uninitialization is implemented using a special value which is checked at runtime whenever a variable is used in arithmetic (or boolean) expressions.

Conditional [26] $C = \mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}$ $(\mathrm{at}_P[\![C]\!] = \ell$ and $\mathrm{after}_P[\![C]\!] = \ell')$

$$\frac{\rho \vdash B \Rrightarrow \mathtt{tt}}{\langle \ell,\ \rho \rangle \Longmapsto\!\![\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!]\!\!\Longrightarrow \langle \mathrm{at}_P[\![S_t]\!],\ \rho \rangle} \tag{26}$$

$$\frac{\rho \vdash T(\neg B) \Rrightarrow \mathtt{tt}}{\langle \ell,\ \rho \rangle \Longmapsto\!\![\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!]\!\!\Longrightarrow \langle \mathrm{at}_P[\![S_f]\!],\ \rho \rangle} \tag{27}$$

$$\frac{\langle \ell_1,\ \rho_1 \rangle \Longmapsto\!\![\![S_t]\!]\!\!\Longrightarrow \langle \ell_2,\ \rho_2 \rangle}{\langle \ell_1,\ \rho_1 \rangle \Longmapsto\!\![\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!]\!\!\Longrightarrow \langle \ell_2,\ \rho_2 \rangle} \tag{28}$$

$$\frac{\langle \ell_1,\ \rho_1 \rangle \Longmapsto\!\![\![S_f]\!]\!\!\Longrightarrow \langle \ell_2,\ \rho_2 \rangle}{\langle \ell_1,\ \rho_1 \rangle \Longmapsto\!\![\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!]\!\!\Longrightarrow \langle \ell_2,\ \rho_2 \rangle} \tag{29}$$

$$\langle \mathrm{after}_P[\![S_t]\!],\ \rho \rangle \Longmapsto\!\![\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!]\!\!\Longrightarrow \langle \ell',\ \rho \rangle \tag{30}$$

$$\langle \mathrm{after}_P[\![S_f]\!],\ \rho \rangle \Longmapsto\!\![\![\mathtt{if}\ B\ \mathtt{then}\ S_t\ \mathtt{else}\ S_f\ \mathtt{fi}]\!]\!\!\Longrightarrow \langle \ell',\ \rho \rangle \tag{31}$$

Iteration[26] $C = \texttt{while}\ B\ \texttt{do}\ S\ \texttt{od}$ ($\mathrm{at}_P[\![C]\!] = \ell$, $\mathrm{after}_P[\![C]\!] = \ell'$ and $\ell_1, \ell_2 \in \mathrm{in}_P[\![S]\!]$)

$$\frac{\rho \vdash T(\neg B) \Rightarrow \mathtt{tt}}{\langle \ell,\ \rho\rangle \Longmapsto[\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{od}]\!]\Longrightarrow \langle \ell',\ \rho\rangle} \quad (32)$$

$$\frac{\rho \vdash B \Rightarrow \mathtt{tt}}{\langle \ell,\ \rho\rangle \Longmapsto[\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{od}]\!]\Longrightarrow \langle \mathrm{at}_P[\![S]\!],\ \rho\rangle} \quad (33)$$

$$\frac{\langle \ell_1,\ \rho_1\rangle \Longmapsto[\![S]\!]\Longrightarrow \langle \ell_2,\ \rho_2\rangle}{\langle \ell_1,\ \rho_1\rangle \Longmapsto[\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{od}]\!]\Longrightarrow \langle \ell_2,\ \rho_2\rangle} \quad (34)$$

$$\langle \mathrm{after}_P[\![S]\!],\ \rho\rangle \Longmapsto[\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{od}]\!]\Longrightarrow \langle \ell,\ \rho\rangle \quad (35)$$

[26] In conditional and iteration commands, execution is blocked when a boolean expression is erroneous i.e. evaluates to $\rho \vdash B \Rightarrow \Omega_e$, $e \in \mathbb{E}$. Another possible semantics would be a nondeterministic choice of the chosen branch. This option corresponds to an implementation where the initial variable can be any value.

Transition system of a program

The transition system of a program $P = S\ ;;$ is

$$\langle \Sigma[\![P]\!],\ \tau[\![P]\!]\rangle$$

where $\Sigma[\![P]\!]$ is the set (23) of program states and $\tau[\![C]\!]$, $C \in \mathrm{Cmp}[\![P]\!]$ is the transition relation for component C of program P, defined by

$$\tau[\![C]\!] \stackrel{\text{def}}{=} \{\langle\langle \ell,\ \rho\rangle,\ \langle \ell',\ \rho'\rangle\rangle \mid \langle \ell,\ \rho\rangle \Longmapsto[\![C]\!]\Longrightarrow \langle \ell',\ \rho'\rangle\} \quad (38)$$

Sequence[27] $C_1\ ;\ \ldots\ ;\ C_n$, $n > 0$ ($i \in [1, n]$:$\ell_i, \ell_{i+1} \in \mathrm{in}_P[\![C_i]\!]$)

$$\frac{\langle \ell_i,\ \rho_i\rangle \Longmapsto[\![C_i]\!]\Longrightarrow \langle \ell_{i+1},\ \rho_{i+1}\rangle}{\langle \ell_i,\ \rho_i\rangle \Longmapsto[\![C_1\ ;\ \ldots\ ;\ C_n]\!]\Longrightarrow \langle \ell_{i+1},\ \rho_{i+1}\rangle} \quad (36)$$

Program $P = S\ ;;$

$$\frac{\langle \ell,\ \rho\rangle \Longmapsto[\![S]\!]\Longrightarrow \langle \ell',\ \rho'\rangle}{\langle \ell,\ \rho\rangle \Longmapsto[\![S\ ;;]\!]\Longrightarrow \langle \ell',\ \rho'\rangle} \quad (37)$$

[26] Note that in the definition (36) of the small-step operational semantics of sequences, the proper sequencing directly follows from the labelling scheme (3) since $\mathrm{after}_P[\![C_i]\!] = \mathrm{at}_P[\![C_{i+1}]\!]$.

Initial States

Execution starts at the program entry point with all variables uninitialized:

$$\mathrm{Entry}[\![P]\!] \stackrel{\text{def}}{=} \{\langle \mathrm{at}_P[\![P]\!],\ \lambda \mathtt{X} \in \mathrm{Var}[\![P]\!] \cdot \Omega_{\mathrm{i}}\rangle\}\ . \quad (39)$$

Final States

Execution ends without error when control reaches the program exit point

$$\mathrm{Exit}[\![P]\!] \stackrel{\mathrm{def}}{=} \{\mathrm{after}_P[\![P]\!]\} \times \mathrm{Env}[\![P]\!] \ .$$

When the evaluation of an arithmetic or boolean expression fails with a runtime error, the program execution is blocked so that no further transition is possible.

Program transition relation structuration

A basic result on the program transition relation is that it is not possible to jump into or out of program components ($C \in \mathrm{Cmp}[\![P]\!]$)

$$\langle\langle \ell,\ \rho\rangle,\ \langle \ell',\ \rho'\rangle\rangle \in \tau[\![C]\!] \Longrightarrow \{\ell, \ell'\} \subseteq \mathrm{in}_P[\![C]\!] \ . \quad (40)$$

The proof, by structural induction on C, is trivial whence omitted.

Big-step operational semantics of a program

- The big-step operational semantics of a program P is

$$\langle \Sigma[\![P]\!],\ t^*[\![P]\!]\rangle$$

  where $t^*[\![P]\!] \stackrel{\mathrm{def}}{=} (t[\![P]\!])^*$ is the reflexive transitive closure of the transition relation $t[\![P]\!]$
- Infinite executions are not considered with this semantics

Reachability semantics of a program

- One can also condider the restriction to entry states (to get the forward/reachability semantics) [28]:

$$\mathrm{Entry}[\![P]\!] \upharpoonleft t^*[\![P]\!]$$

  to exit states (to get the backward reachability semantics) [29]:

$$t^*[\![P]\!] \upharpoonright \mathrm{Exit}[\![P]\!]$$

  or both (this is the natural semantics):

$$\mathrm{Entry}[\![P]\!] \upharpoonleft t^*[\![P]\!] \upharpoonright \mathrm{Exit}[\![P]\!]$$

28 $X \upharpoonleft r \stackrel{\mathrm{def}}{=} \{\langle x,\ y\rangle \mid x \in X \wedge r(x,y)\}$

29 $r \upharpoonright Y \stackrel{\mathrm{def}}{=} \{\langle x,\ y\rangle \mid r(x,y) \wedge y \in Y\}$

Standard Interpreter of the Simple Imperative Language (SIL) in OCaml

Values

The Values module provides the description of machine operations on values.

```
(* values.mli *)
type error_type = INITIALIZATION | ARITHMETIC
(* machine integers *)
type machine_int = ERROR_NAT of error_type | NAT of int
val machine_int_of_string : string -> machine_int
val machine_unary_random : unit -> machine_int
val machine_unary_plus : machine_int -> machine_int
val machine_unary_minus : machine_int -> machine_int
val machine_binary_plus : machine_int -> machine_int -> machine_int
val machine_binary_minus : machine_int -> machine_int -> machine_int
val machine_binary_times : machine_int -> machine_int -> machine_int
val machine_binary_div : machine_int -> machine_int -> machine_int
```



```
val machine_binary_mod : machine_int -> machine_int -> machine_int

(* machine booleans *)
type machine_bool = ERROR_BOOL of error_type | BOOLEAN of bool
val machine_eq : machine_int -> machine_int -> machine_bool
val machine_lt : machine_int -> machine_int -> machine_bool
val machine_and : machine_bool -> machine_bool -> machine_bool
val machine_or : machine_bool -> machine_bool -> machine_bool

(* printing *)
val print_machine_int : machine_int -> unit
```

See values.ml for the OCaml implementation.

Environments

The Env module provides the description of environments assigning machine values to variables.

```
(* env.mli *)
open Abstract_Syntax
open Variables
open Values
type env                                                        (* environments
val initerr   : unit -> env                                     (* uninitializati
val copy      : env -> env                                      (* copy
val get       : env -> variable -> machine_int                  (* r(X)
val set       : env -> variable -> machine_int -> unit          (* r[X <- v]
val print_env : env -> unit                                     (* printing
```

See env.ml for the OCaml implementation.

Evaluation of Arithmetic Expressions

The `Aexp` module provides an evaluator of arithmetic expressions.

```
(* aexp.mli *)
open Abstract_Syntax
open Values
open Env
(* evaluation of arithmetic operations *)
val eval_aexp : aexp ->  env -> machine_int
```



```
(* aexp.ml *)
open Abstract_Syntax
open Values
open Env
(* evaluation of arithmetic operations *)
let rec eval_aexp e r = match e with
| Abstract_Syntax.NAT i -> machine_int_of_string i
| VAR v        -> get r v
| RANDOM       -> machine_unary_random ()
| UPLUS a      -> machine_unary_plus (eval_aexp a r)
| UMINUS a     -> machine_unary_minus (eval_aexp a r)
| PLUS (a, b)  -> machine_binary_plus (eval_aexp a r) (eval_aexp b r)
| MINUS (a, b) -> machine_binary_minus (eval_aexp a r) (eval_aexp b r)
| TIMES (a, b) -> machine_binary_times (eval_aexp a r) (eval_aexp b r)
| DIV (a, b)   -> machine_binary_div (eval_aexp a r) (eval_aexp b r)
| MOD (a, b)   -> machine_binary_mod (eval_aexp a r) (eval_aexp b r)
```


Evaluation of Boolean Expressions

The `Bexp` module provides an evaluator of normalized boolean expressions.

```
(* bexp.mli *)
open Abstract_Syntax
open Values
open Env
(* evaluation of boolean operations *)
val eval_bexp : bexp -> env -> machine_bool
```



```
(* bexp.ml *)
open Abstract_Syntax
open Values
open Env
open Aexp
(* evaluation of boolean operations *)
let rec eval_bexp b r =
  match b with
   | TRUE           -> (BOOLEAN true)
   | FALSE          -> (BOOLEAN false)
   | (EQ (a1, a2))  -> machine_eq (eval_aexp a1 r) (eval_aexp a2 r)
   | (LT (a1, a2))  -> machine_lt (eval_aexp a1 r) (eval_aexp a2 r)
   | (AND (b1, b2)) -> machine_and (eval_bexp b1 r) (eval_bexp b2 r)
   | (OR (b1, b2))  -> machine_or (eval_bexp b1 r) (eval_bexp b2 r)
```

Small-step operational semantics

The `Smallstep` module provides the execution of one program step.

```
(* smallstep.mli *)
open Abstract_Syntax
open Labels
open Env
(* program states *)
type state = label * env
(* small-step oàperation semantics of commands *)
val trans : com -> state -> state
(* run-tme errors *)
exception Error of string
```



```
(* smallstep.ml *)
open Abstract_Syntax
open Labels
open Values
open Env
open Aexp
open Bexp
(* program states *)
type state = label * env
(* small-step operational semantics of commands *)

exception Error of string
let rec trans c (l, r) = match c with
| (SKIP (l', l'')) ->
    if (l = l') then (l'', r)
    else (raise (Error "SKIP incoherence"))
| (ASSIGN (l',x,a,l'')) ->
```



```
    if (l = l') then
      (let v = (eval_aexp a r) in
         (set r x v;
          (l'', r)))
    else (raise (Error "ASSIGN incoherence"))
| (SEQ (l', s, l'')) ->
    (transseq s (l, r))
| (IF (l', b, nb, t, f, l'')) ->
    (if (l = l') then
       (match (eval_bexp b r) with
        | ERROR_BOOL e -> (raise (Error ("runtime error in \"if\" at "
                                         ^ (string_of_label l))))
        | BOOLEAN true -> ((at t), r)
        | BOOLEAN false -> match (eval_bexp nb r) with
            | ERROR_BOOL e ->
              (raise (Error ("runtime error in \"if\" at "
                             ^ (string_of_label l))))
            | BOOLEAN true -> ((at f), r)
```



```
            | BOOLEAN false -> (raise (Error "IF test incoherence")))
     else if (l = (after t)) then (l'', r)
     else if (l = (after f)) then (l'', r)
     else if (incom l t) then
        (trans t (l, r))
     else if (incom l f) then
        (trans f (l, r))
     else (raise (Error "IF incoherence")))
| (WHILE (l', b, nb, c', l'')) ->
   (if (l = l') then
      (match (eval_bexp b r) with
       | ERROR_BOOL e -> (raise (Error
        ("runtime error in \"while\" loop at " ^ (string_of_label l))))
       | BOOLEAN true -> ((at c'), r)
       | BOOLEAN false -> match (eval_bexp nb r) with
         | ERROR_BOOL e ->
           (raise (Error ("runtime error in \"while\" loop at "
                          ^ (string_of_label l))))
```

```
          | BOOLEAN true -> (l'', r)
          | BOOLEAN false -> (raise (Error "WHILE test incoherence")))
      else if (l = (after c')) then (l', r)
      else if (incom l c') then
        (trans c' (l, r))
      else (raise (Error "WHILE incoherence")))
and transseq s (l, r) = match s with
| []   -> raise (Error "empty SEQ incoherence")
| [c]  -> if (incom l c) then
            (trans c (l, r))
          else (raise (Error "SEQ incoherence"))
| h::t -> if (l = (after h)) then (transseq t (l, r))
          else if (incom l h) then (trans h (l, r))
          else (transseq t (l, r))
```


Big-step operational semantics

The Bigstep module provides the natural program semantics.

```
(* bigstep.mli *)
open Abstract_Syntax
(* program execution *)
val run :  com -> unit
```

- We record only the (initial and) final state(s), not all intermediate states.
- The implementation may not terminate for nonterminating programs (no attempt is made to detect nontermination, which is undecidable).


```
(* bigstep.ml *)
open Abstract_Syntax
open Labels
open Values
open Env
open Smallstep

(* big-step operational semantics of commands *)
let run p =
  let rec exec (l, r) =
    if l = (exit ())
      then (print_env r; print_newline ())
      else
        let (l', r') = trans p (l, r) in
          exec (l', r')
  in
     (try exec ((entry ()), (initerr ()))
```



```
       with Error s -> print_string ("Fatal error:" ^ s ^ ".\n"))
```

The Standard Interpretor

The `Main` module provides the standard interpreter.

```
(* main.ml *)
open Program_To_Abstract_Syntax
open Pretty_Print
open Bigstep
(* read, parse and execute the program *)
let _ =
  let arg = if (Array.length  Sys.argv) = 1 then ""
            else Sys.argv.(1) in
     Random.self_init ();
    let p = (abstract_syntax_of_program arg) in
      (pretty_print p;
       run p)
```


Example

Input:

```
x := 1;
while (x < 100) do
  x := x + 1
od;;
```

Result:

```
0:
  x := 1;
1:
  while (x < 100) do
    2:
      x := (x + 1)
    3:
  od {((100 < x) | (x = 100))}
4:

x = 100;
```

See the `makefile` and the other examples.

Standardization of programming languages

– For professional programming languages, the definition is often pseudo-formal, in english;
– See for example the standardization of C [9].

Reference

[9] JTC 1/SC 22. Programming languages — C. Technical report, ISO/IEC 9899:1999, 16 Dec. 1999.

THE END

My MIT web site is http://www.mit.edu/~cousot/

The course web site is http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/.