New York University, CIMS, CS, Course CSCI-GA.3140-001, Spring 2024

"Abstract Interpretation"

Ch. 5, Parsing

(absolute prerequisite for the optional project)

Patrick Cousot

pcousot@cs.nyu.edu cs.nyu.edu/~pcousot

Class 2

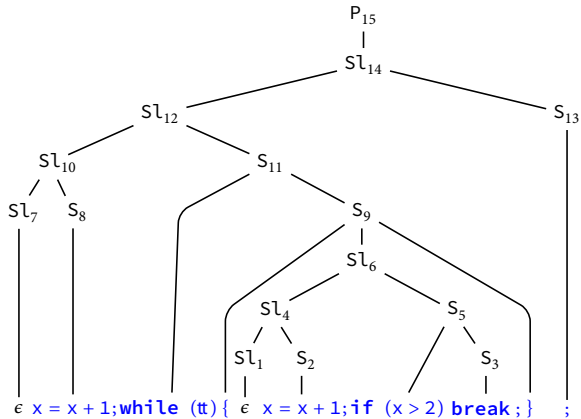
These slides are available at
http://cs.nyu.edu/~pcousot/courses/spring24/CSCI-GA.3140-001/slides/02--2024-01-29-syntax-semantics-traces-oo/
/slides-05--parsing-AI.pdf

Chapter 5

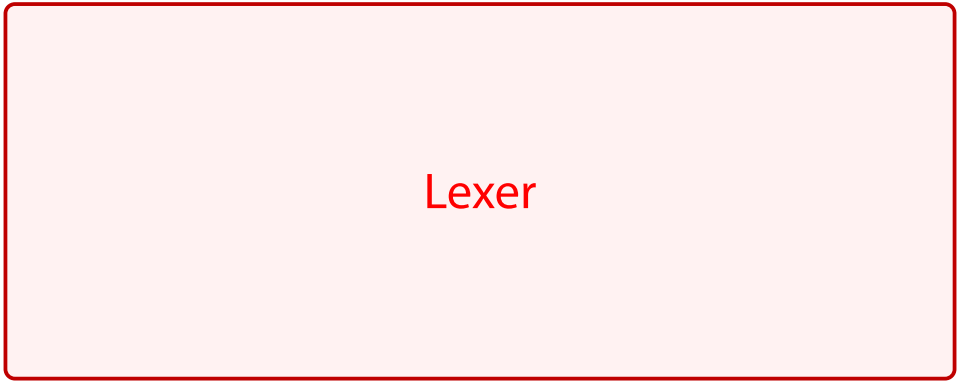
Ch. 5, Parsing

Parsing consists in building an abstract syntax tree (AST) of a program recognized by a context-free grammar (or rejecting that sentence because of a syntax error).

```
S  ::=  x = A ;
    |   ;
    |   if (B) S
    |   if (B) S else S
    |   while (B) S
    |   break ;
    |   { Sl }
Sl ::=  Sl S | ϵ
P  ::=  Sl
```



en.wikipedia.org/wiki/Context-free_grammar
en.wikipedia.org/wiki/Abstract_syntax_tree



Lexing

- Consider the program

```
if (x<0)  x = -x;.
```

- In a first lexing phase, a program (called a lexer) will abstract the terminals in this input sentence by lexemes, as follows.

```
IF LPARENT IDENT LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END
if    (      x    <  0     )      x     =      -     x       ;
```

- The END lexeme represents the end of the program, for example the end of the input file.

en.wikipedia.org/wiki/Lexical_analysis

Lexer specification

Regular expressions

- The lexemes can be described by a regular expressions
- The syntax of regular expressions is

$$R \;::=\; \epsilon \;\Big|\; \texttt{'letter'} \;\Big|\; \texttt{"text"} \;\Big|\; R_1 R_2 \;\Big|\; R_1 \mid R_2 \;\Big|\; R^+ \;\Big|\; R^* \;\Big|\; (R)$$

- The semantics of regular expressions is a set of strings, as follows
  - ϵ denotes the empty string
  - `'letter'` denotes the string of one letter `letter`
  - `"text"` denotes the string `text`
  - $R_1 R_2$ denotes the concatenation of the strings denoted by R_1 and R_2
  - $R_1 \mid R_2$ denotes the string denoted either by R_1 or by R_2
  - R^+ denotes any concatenation of several strings denoted by R
  - R^* denotes the strings denoted by $\epsilon \mid R^+$

`en.wikipedia.org/wiki/Regular_expression`

Lexer generation

Lexer specification

The lexer can be generated automatically from a formal description of the lexemes by regular expressions as follows.

```
en.wikipedia.org/wiki/Lex_(software)
en.wikipedia.org/wiki/Flex_(lexical_analyser_generator)
caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html
```

```
(* File lexer.mll *)
{
open Parser
exception Error of string
}
rule token = parse
    [' ' '\t' '\n'] { token lexbuf } (* skip blanks, tabs and newlines *)
  | "nand"      { NAND }
  | "if"        { IF }
  | "else"      { ELSE }
  | "while"     { WHILE }
  | "break"     { BREAK }
  | (['a'-'z'] | ['A'-'Z'])(['a'-'z'] | ['A'-'Z'] | ['0'-'9'])* as idt
                { IDENT idt }
  | ['0'-'9']+ as num
                { NUM (int_of_string num) }
  | '-'         { MINUS }
  | '<'         { LT }
  | '('         { LPAREN }
  | ')'         { RPAREN }
  | '='         { ASSIGN }
  | ';'         { SEMICOLON }
  | '{'         { LBRACKET }
  | '}'         { RBRACKET }
  | eof         { END }
  | _
    { raise (Error (Printf.sprintf "At offset %d: unexpected character.\n"
                                    (Lexing.lexeme_start lexbuf))) }
```

Comments on the lexer specification I

7,28: The `lexbuf` contains the input sentence which is a sequence of characters.

6: Each regular expression is considered in order.

6: For the first that applies a lexeme is returned and lexing goes on with the rest of the input sentence `lexbuf`.

7: the alternative `[' ' '\t' '\n']` is used to avoid repeating the same rule

7: `token lexbuf` moves to the next character in `lexbuf` so that space, tabulation, and end of line characters are skipped.

8–12: Then reserved keywords are checked.

13: Otherwise, an attempt is made to recognize an identifier that is the longest possible string, is any, starting by a lower or upper letter followed by zero or more letters or digits.

14: If an identifier is recognized, this string of characters is returned as part of the lexeme.

Comments on the lexer specification II

15: Otherwise an attempt is made to recognize a natural number as a nonempty sequence of digits.

16: The value returned with the lexeme is the value of that sequence of digits considered with the decimal encoding.

17–24: Otherwise an attempt is made to recognize symbols, where the end of file `eof` is considered to be the end of the program marqued by the lexeme END.

27,27: If none of the above cases applies, an error exception is raised.

Error exceptions

- For example for the input

```
x=10; while (x>0) x=x-1;
```

we get the error

```
At offset 14: unexpected character.
```

since `>` is not an integer comparison operator in our language.

Lexer generation

- A lexer is an OCaml program which, whenever called, will return the next lexeme recognized (in `lexbuf`).
- The lexemes will be OCaml values of the following

```
type token =  WHILE | SEMICOLON | RPAREN | RBRACKET | NUM of (int) | NAND | MINUS | LT
            | LPAREN | LBRACKET | IF | IDENT of (string) | END | ELSE | BREAK | ASSIGN
```

  (which is automatically generated from the parser specification in the file `parser.mly` below).

- The `lexbuf` buffer can be chosen to be a file, the input keyboard, or a string.
- The lexer generator `ocamllex` will take this specification and automatically produce a lexer, returning lexemes of that type `token`.

ocaml.org

Grammar

Expressions

The following grammar specifies a subset of C expressions

$x, y, \ldots$	$\in$	$\mathbb{V}$			variables ($\mathbb{V}$ not empty)
A	$\in$	$\mathbb{A}$	$::=$	$1 \mid x \mid A_1 - A_2$	arithmetic expressions
B	$\in$	$\mathbb{B}$	$::=$	$A_1 < A_2 \mid B_1$ nand B_2	boolean expressions
E	$\in$	$\mathbb{E}$	$::=$	$A \mid B$	expressions

en.wikipedia.org/wiki/Context-free_grammar
en.wikipedia.org/wiki/C_(programming_language)

Programs

The following grammar specifies a subset of C programs

S	::=		statement $S \in \mathbb{S}$
		x = A ;	assignment
	|	;	skip
	|	**if** (B) S	conditional
	|	**if** (B) S **else** S	
	|	**while** (B) S	iteration
	|	**break** ;	iteration break
	|	{ Sl }	compound statement
Sl	::=	Sl S | ϵ	statement list $Sl \in \mathbb{S}\mathbb{l}$
P	::=	Sl	program $P \in \mathbb{P}$
$\mathbb{P}c$	$\triangleq$	$\mathbb{S} \cup \mathbb{S}\mathbb{l} \cup \mathbb{P}$	program component $S \in \mathbb{P}c$

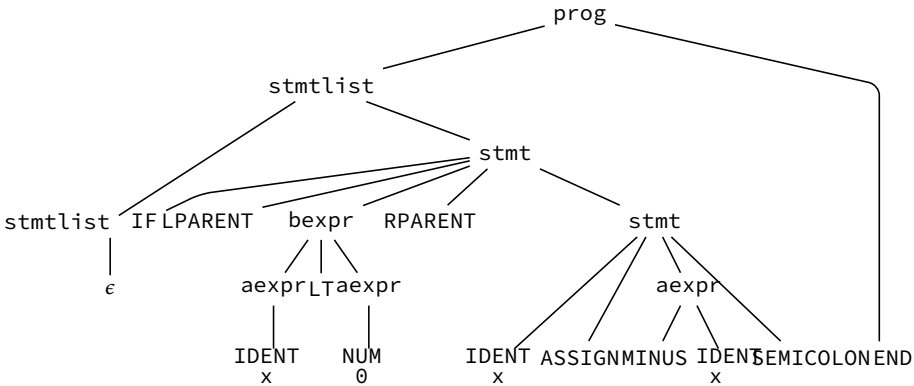
Grammar specification

- The grammar of the language is specified by the grammar specification for the parser (on next slide)[1].
- Terminals in the input sentence have been replaced by lexemes so that the grammar input sentence is a sequence of lexemes.
- More precisely, whenever the next lexeme is required by the parser it is obtained by calling the lexer (for a specific input buffer `lexbuf`).

[1] There are many other different syntaxes for grammars such as `en.wikipedia.org/wiki/Backus-Naur_form`

```
prog:
  | stmtlist END

stmt:
  | IDENT ASSIGN aexpr SEMICOLON
  | SEMICOLON
  | IF LPAREN bexpr RPAREN stmt
  | IF LPAREN bexpr RPAREN stmt ELSE stmt
  | WHILE LPAREN bexpr RPAREN stmt
  | BREAK SEMICOLON
  | LBRACKET stmtlist RBRACKET

stmtlist:
  | stmtlist stmt
  |

aexpr:
  | NUM
  | IDENT
  | aexpr MINUS aexpr
  | MINUS aexpr
  | LPAREN aexpr RPAREN

bexpr:
  | aexpr LT aexpr
  | bexpr NAND bexpr
  | LPAREN bexpr RPAREN
```

- The objective of a parser is to built the parse/syntax tree of a program
- For the example program `if (x<0) x = -x ;`, the parse tree is the following

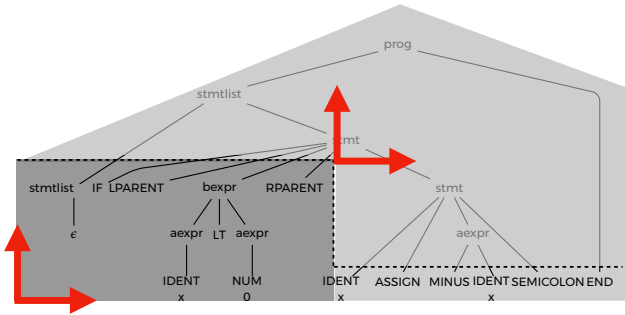


By the way, this program is not the absolute value with machine integers (because of the overflow with the smallest integer)

en.wikipedia.org/wiki/Parsing

Parser I

- We consider left-to-right, bottom-up parsing so that the syntax tree is built in prefix order from the input sentence of lexemes.
- Initially, the syntax tree is empty and the input sentence is the sequence of lexemes to be recognized.



en.wikipedia.org/wiki/LALR_parser

Parser II

An intermediate state of the parser would be the following.

- Only the top of the prefix of the parsing tree that has been recognized until now is used so that it can be stored in a stack.
- The input is the rest of the sequence of lexemes to be recognized.

Parser III

Depending on the state of the stack and the input, the next action of the parser will be either

- a shift (going right),
- a reduce (going up),
- an acceptance, or
- a rejection

of the sentence.

Parser IV

- A *reduce* consists in replacing the righthand side of a grammar rule recognized on top of the stack by the lefthand side of the rule.
- Example
  - stack: `stmtlist IF LPARENT aexpr LT aexpr`
  - input: `RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END`

  A reduction replaces the righthand side `aexpr LTaexpr` of the grammar rule by its lefthand side `bexpr`
  - stack: `stmtlist IF LPARENT bexpr`
  - input: unchanged `RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END`

Parser V

- A *shift* is for the case when the stack contains part of the righthand side of a rule in construction and a lexeme is pushed from the input to the stack.
- Example
  - stack: `stmtlist IF LPARENT bexpr`
  - input: `RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END`

  The next lexeme `RPARENT` is shifted to the stack which becomes
  - stack: `stmtlist IF LPARENT bexpr RPARENT`
  - input:`IDENT ASSIGN MINUS IDENT SEMICOLON END`

Parser VI

- An *acceptance* is when the stack is reduced to the grammar axiom and the input is empty.
- Otherwise it is a *rejection* i.e. the stack cannot be extended by the input to a meaningful sentence of the language defined by the grammar.

Parser VII

The parsing of the input sentence

IF LPARENT IDENT LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

proceeds as follows (a state has the form stack | input, $\xrightarrow{S}$ is a shift and $\xrightarrow{R}$ is a reduction).

| IF LPARENT IDENT LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{R}$ stmtlist | IF LPARENT IDENT LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF | LPARENT IDENT LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT | IDENT LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT IDENT | LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{R}$ stmtlist IF LPARENT aexpr | LT NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT aexpr LT | NUM RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

Parser VIII

$\xrightarrow{S}$ stmtlist IF LPARENT aexpr LT NUM | RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{R}$ stmtlist IF LPARENT aexpr LT aexpr | RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{R}$ stmtlist IF LPARENT bexpr | RPARENT IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT bexpr RPARENT | IDENT ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT bexpr RPARENT IDENT | ASSIGN MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT bexpr RPARENT IDENT ASSIGN | MINUS IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT bexpr RPARENT IDENT ASSIGN MINUS | IDENT SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT bexpr RPARENT IDENT ASSIGN MINUS IDENT | SEMICOLON END

$\xrightarrow{R}$ stmtlist IF LPARENT bexpr RPARENT IDENT ASSIGN aexpr | SEMICOLON END

$\xrightarrow{S}$ stmtlist IF LPARENT bexpr RPARENT IDENT ASSIGN aexpr SEMICOLON | END

$\xrightarrow{R}$ stmtlist IF LPARENT bexpr RPARENT stmt | END

Parser IX

$\xrightarrow{R}$ `stmtlist stmt` | `END`

$\xrightarrow{R}$ `stmtlist` | `END`

$\xrightarrow{S}$ `stmtlist END` |

$\xrightarrow{R}$ `prog` |

A syntactically valid program has been recognized.

`en.wikipedia.org/wiki/Shift-reduce_parser`

Parsers

There are numerous deterministic parsers available for LALR(k) and LR(k) grammars.

en.wikipedia.org/wiki/LALR_parser
en.wikipedia.org/wiki/LR_parser

en.wikipedia.org/wiki/Yacc
en.wikipedia.org/wiki/GNU_Bison
caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html (ocamlyacc)
gallium.inria.fr/~fpottier/menhir/

Ambiguities, shift-reduce conflicts

Ambiguity I

- The grammar is ambiguous in that a sentence can be parsed in different ways.
- For example `1-2-3` can be parsed as `((1-2)-3)` or `(1-(2-3))` which does not return the same result.
- In mathematics, `1-2-3` means `((1-2)-3)`
- The binary minus operation `-` is left-associative and evaluated left-to-right.
- The ambiguity is solved by the following declaration in the grammar specification.

```
%left MINUS
```

- This declares `MINUS` to be left-associative (i.e. do a reduce rather than a shift).

en.wikipedia.org/wiki/Ambiguous_grammar

Ambiguity II

- Another ambiguity is for expressions like `1 - -2` which should be evaluated as `(1 - (-2))` i.e. the unary operator `-` in `-2` should be evaluated before the binary operator `-` in `1 - ␣`.
- Another difficulty is that the unary and binary symbols are the same `-`.
- These two problems are solved by declaring that

  (1) the unary minus is evaluated before the binary minus

```
%left MINUS          /* evaluated second */
%nonassoc UMINUS     /* evaluated first */
```

  (2) the unary minus has the syntax of the binary minus but the precedence of the unary minus (which should be evaluated first)

```
aexpr:
    | NUM
    | IDENT
    | aexpr MINUS aexpr
    | MINUS aexpr %prec UMINUS
    | LPAREN aexpr RPAREN
```

Ambiguity III

- Another ambiguity is for expressions like `1-2<3` which should be parsed as `((1-2)<3)` and not `(1-(2<3))`

- The binary `-` should be evaluated before the logical comparison `<`, as indicated by the declaration

```
%left NAND          /* evaluated fourth */
%left LT            /* evaluated third */
%left MINUS         /* evaluated second */
%nonassoc UMINUS    /* evaluated first */
```

- The `NAND` is left-associative and should be evaluated last.

Shift-reduce conflict

- All this situations yield a shift-reduce conflict.
- For example during the analysis of `1-2-x`, we reach the state

  $aexpr\ \texttt{MINUS}\ aexpr \mid \texttt{MINUS}\ \texttt{IDENT}$

- We can either perform
  - a reduce

    $\xrightarrow{R} aexpr \mid \texttt{MINUS}\ \texttt{IDENT}$

    in case of left-associativity or
  - a shift

    $\xrightarrow{S} aexpr\ \texttt{MINUS}\ aexpr\ \texttt{MINUS} \mid \texttt{IDENT}$

    in case of right-associativity.
- The declaration indicates that the `MINUS` operator is left-associative so the reduce should be performed, not the shift.

Ambiguity IV (conditional)

- The grammar for the conditional is also ambiguous.
- For example

  $\texttt{if (bexpr}_1\texttt{) if (bexpr}_2\texttt{) stmt}_1\texttt{ else stmt}_2$

  can be understood either as

  $\texttt{(if (bexpr}_1\texttt{)(if (bexpr}_2\texttt{) stmt}_1\texttt{)else stmt}_2\texttt{)}$

  or as

  $\texttt{(if (bexpr}_1\texttt{)(if (bexpr}_2\texttt{) stmt}_1\texttt{ else stmt}_2\texttt{)).}$

- In most programming languages, the `else` refers to the closest enclosing `if` i.e. the ambiguity is solved according to the second parsing.

en.wikipedia.org/wiki/Dangling_else

Ambiguity IV (conditional)

Again precedences can be used as follows.

```
%nonassoc NO_ELSE     /* evaluated sixth */
%nonassoc ELSE        /* evaluated fifth */
%left NAND            /* evaluated fourth */
%left LT              /* evaluated third */
%left MINUS           /* evaluated second */
%nonassoc UMINUS      /* evaluated first */
```

The grammar is now

```
stmt:
  | ...
  | IF LPAREN b = bexpr RPAREN s = stmt %prec NO_ELSE
  | IF LPAREN b = bexpr RPAREN s1 = stmt ELSE s2 = stmt
  | ...
```

The `%prec NO_ELSE` states that the lexeme `NO_ELSE` is fake (an empty string in the input). Moreover it does not associate with itself and is evaluated last.

Ambiguity IV (conditional)

- Therefore an input

  $\texttt{if (bexpr}_1\texttt{) if (bexpr}_2\texttt{) stmt}_1\texttt{ else if (bexpr}_3\texttt{) stmt}_2\texttt{ else stmt}_3$,

  is parsed as

  $(\texttt{if (bexpr}_1\texttt{) if (bexpr}_2\texttt{) stmt}_1\texttt{ else if (bexpr}_3\texttt{) stmt}_2\texttt{ else stmt}_3\ \texttt{NO\_ELSE})$

- Moreover the ELSE does not associate with itself
- i.e. interpreting $\_$ ELSE $\_$ ELSE $\_$ as $((\_\ \texttt{ELSE}\ \_)\ \texttt{ELSE}\ \_)$ or $(\_\ \texttt{ELSE}\ (\_\ \texttt{ELSE}\ \_))$ is meaningless

- The parsing is

  $(\texttt{if (bexpr}_1\texttt{)}\quad(\texttt{if (bexpr}_2\texttt{) stmt}_1\texttt{ else}\quad(\texttt{if (bexpr}_3\texttt{) stmt}_2\texttt{ else stmt}_3\texttt{))}\,\texttt{NO\_ELSE})$

Ambiguities, reduce-reduce conflicts

Reduce-reduce conflict

A reduce-reduce conflict occurs during parsing when a reduction can be applied with two or more different grammar rule righthand sides, for the same input token.

Example of reduce-reduce conflict

```
/* File parser.mly */
%{ %}
%token NUM
%start s /* the grammar entry point */
%type  <unit> s
%type  <unit> t

%% /* rules */
s:
  | t       { () }
  | NUM     { () }

t:
  | NUM     { () }

%%
```

- A `NUM` can be reduced to a `s` or to a `t` and then to a `s`.

Error message for the reduce-reduce conflict

The menhir parser generator produces the following error message,

```
Warning: one state has reduce/reduce conflicts.
Warning: one reduce/reduce conflict was arbitrarily resolved.
File "parser.mly", line 14, characters 4-7:
Warning: production t -> NUM is never reduced.
```

- menhir explains the problem in the parser.conflicts file.
- menhir arbitrarily chooses the first alternative
- The second alternative is never used, which is most certainly an error in the design of the context free-grammar.

Solving ambiguities by grammar rewriting

It is always possible to resolve the ambiguities by rewriting the grammar.

For example, the grammar of arithmetic expressions would be

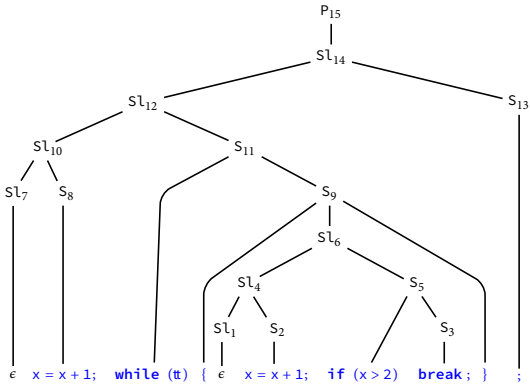
```
<Exp> ::= <Exp> + <Term> |
          <Exp> - <Term> |
          <Term>

<Term> ::= <Term> * <Factor> |
           <Term> / <Factor> |
           <Factor>

<Factor> ::= <Variable> |
             <Constant> |
             ( <Exp> ) |
             - <Factor>
```

Using priorities is usually much simpler.

Abstract syntax

Abstract syntax

- A program can be represented by a tree called its abstract syntax.



- For parsing only, the abstract syntax is unit ()

en.wikipedia.org/wiki/Abstract_syntax_tree

Abstract syntax

In the programming language OCaml [Leroy, Doligez, Frisch, Garrigue, Rémy, and Vouillon, 2021], the abstract syntax type is

```
1  (* file abstractSyntax.ml *)
2
3  type variable = string
4  type aexpr = Num of int | Var of string | Minus of aexpr * aexpr
5  type bexpr = Lt of aexpr * aexpr | Nand of bexpr * bexpr
6  type 'a tree =
7    | Prog of 'a tree list * 'a
8    | Assign of variable * aexpr * 'a
9    | Emptystmt of 'a
10   | If of bexpr * 'a tree * 'a
11   | Ifelse of bexpr * 'a tree * 'a tree * 'a
12   | While of bexpr * 'a tree * 'a
13   | Break of 'a
14   | Stmtlist of 'a tree list * 'a (* trees in inverse order *)
```

The abstract syntax tree type is parameterized by the type `'a` of the attributes attached to the nodes of the tree

ocaml.org
en.wikipedia.org/wiki/OCaml

Construction of the abstract syntax tree

- The abstract syntax tree can be constructed during parsing
- The stack elements ⟨`nt, ast`⟩ are pairs of a non-terminal `nt` (or terminal) and an abstract syntax subtree `ast`
- Each shift extends the tree left-to-right
- Each reduction extends the tree bottom-up

`en.wikipedia.org/wiki/Attribute_grammar`

Example, cont'd I

...

$\xrightarrow{S}$ ⟨stmtlist, Emptystmt⟩ IF LPARENT ⟨bexpr, LT (Var "x",Num 0)⟩ RPARENT
IDENT ASSIGN MINUS ⟨IDENT, Var "x"⟩ | SEMICOLON END
"x"

$\xrightarrow{R}$ ⟨stmtlist, Emptystmt⟩ IF LPARENT ⟨bexpr, LT (Var "x",Num 0)⟩ RPARENT
IDENT ASSIGN ⟨aexpr, Minus (Num 0, Var "x")⟩ | SEMICOLON END
"x"

$\xrightarrow{S}$ ⟨stmtlist, Emptystmt⟩ IF LPARENT ⟨bexpr, LT (Var "x",Num 0)⟩ RPARENT
IDENT ASSIGN ⟨aexpr, Minus (Num 0, Var "x")⟩ SEMICOLON | END
"x"

$\xrightarrow{R}$ ⟨stmtlist, Emptystmt⟩ IF LPARENT ⟨bexpr, LT (Var "x",Num 0)⟩ RPARENT ⟨stmt,
Assign ("x",Minus (Num 0, Var "x"))⟩ | END

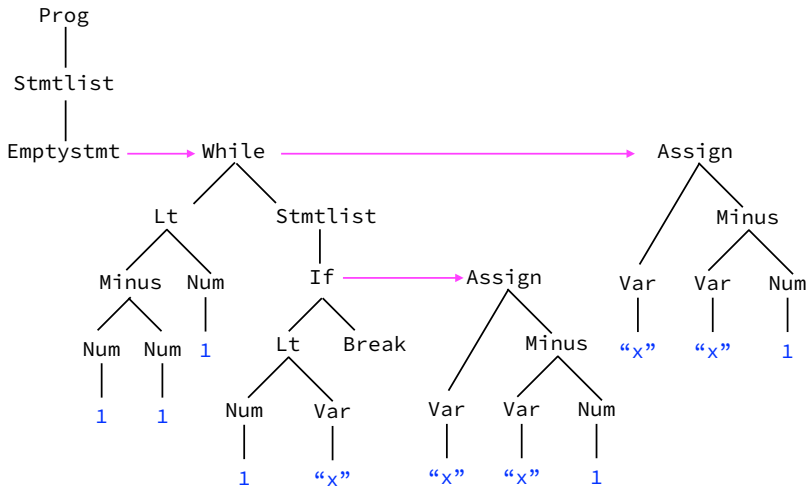
Example, cont'd II

$\xrightarrow{R}$ ⟨stmtlist, Emptystmt⟩ ⟨stmt,

If (LT (Var "x",Num 0), Assign ("x",Minus (Num 0, Var "x")))⟩ | END

...

Example of OCaml Abstract Syntax Tree




```
x = x + 1; while ((1 - 1) < 1) { x = x + 1; if (1 < x) break; } ;
```

Parser specification

Parser specification

- The parser specification on next slide lists
  - the lexemes (`%token`), possibly with an associated OCaml type,
  - the precedences (`%left`, `%right`, `%nonassoc`) in increasing priority order,
  - the grammar axiom (`%start`),
  - the OCaml type of the abstract syntax tree associated to the grammar nonterminals (`%type`)
  - the grammar rules.

```
en.wikipedia.org/wiki/Yacc
dev.realworldocaml.org/parsing-with-ocamllex-and-menhir.html
```

```
1  /* File parser.mly */
2
3  %{ (* header *)
4  open AbstractSyntax
5
6  %} /* declarations */
7
8  %token MINUS LT NAND LPAREN RPAREN ASSIGN /* lexer tokens */
9  %token SEMICOLON IF ELSE WHILE BREAK LBRACKET RBRACKET END
10 %token <string> IDENT
11 %token <int> NUM
12
13 %nonassoc NO_ELSE    /* evaluated fifth */
14 %nonassoc ELSE       /* evaluated fourth */
15 %left NAND           /* evaluated third */
16 %left MINUS          /* evaluated second */
17 %nonassoc UMINUS     /* evaluated first */
18
19 %start prog          /* the grammar entry point */
20 %type  <unit AbstractSyntax.tree> prog
21 %type  <aexpr> aexpr
22 %type  <bexpr> bexpr
23 %type  <unit AbstractSyntax.tree> stmt
```

```
24  %type  <unit AbstractSyntax.tree list> stmtlist
25
26  %% /* rules */
27
28  prog:
29    | l = stmtlist END                          { Prog (l, ()) }
30  stmt:
31    | x = IDENT ASSIGN a = aexpr SEMICOLON      { Assign (x, a, ()) }
32    | SEMICOLON                                 { Emptystmt () }
33    | IF LPAREN b = bexpr RPAREN s = stmt %prec NO_ELSE   { If (b, s, ()) }
34    | IF LPAREN b = bexpr RPAREN s1 = stmt
35      ELSE s2 = stmt                            { Ifelse (b, s1, s2, ()) }
36    | WHILE LPAREN b = bexpr RPAREN s = stmt { While (b, s, ()) }
37    | BREAK SEMICOLON                           { Break () }
38    | LBRACKET l = stmtlist RBRACKET            { Stmtlist (l, ()) }
39  stmtlist:
40    | l = stmtlist s = stmt                     { s :: l (* nodes in inverse order *) }
41    |                                           { [] }
42  aexpr:
43    | n = NUM                                   { Num n }
44    | x = IDENT                                 { Var x }
45    | a1 = aexpr MINUS a2 = aexpr               { Minus (a1, a2) }
46    | MINUS a = aexpr                           { Minus ((Num 0), a) } %prec UMINUS
```

```
47    | LPAREN a = aexpr RPAREN                { a }
48  bexpr:
49    | a1 = aexpr LT a2 = aexpr               { Lt (a1, a2) }
50    | b1 = bexpr NAND b2 = bexpr             { Nand (b1, b2) }
51    | LPAREN b = bexpr RPAREN                { b }
52
53  %% (* trailer *)
```

Comments on the parser specification

- The types are defined in the module `AbstractSyntax` (in file `abstractSyntax.ml`)
- The `AbstractSyntax` module is opened in the header to avoid explicit qualification (such as `AbstractSyntax.aexpr`).
- The grammar rules give a name to the abstract syntax trees already constructed (e.g. `l` in `l = stmtlist`)
- They use these named abstract syntax subtrees to construct the abstract syntax tree of the rule lefthand side non terminal upon reduction (e.g. `{ Prog l }`).

Parser generation

Automatic parser generation

- Given parser specification, parser generators such as `ocamlyacc` or `menhir` can automatically generate parsers in the OCaml programming language.
- Similar tools exist for others languages (`yacc`, `bison`, etc.)
- The parser can be called in a main program (called `main.ml` in our example).

```
en.wikipedia.org/wiki/Yacc
en.wikipedia.org/wiki/GNU_Bison
```

```
(* File main.ml *)

open AbstractSyntax

let rec print_aexpr a = ...;;
let rec print_bexpr a = ...;;

let rec print_program p = ...
and print_stmtlist sl = ...
and print_stmt s = ...;;

let lexbuf = Lexing.from_channel stdin in
  try
    print_program (Parser.prog Lexer.token lexbuf)
  with
  | Lexer.Error msg ->
      Printf.fprintf stderr "%s%!" msg
  | Parser.Error ->
      Printf.fprintf stderr "At offset %d: syntax error.\n%!"
                                    (Lexing.lexeme_start lexbuf);;
```

Comments on the `main` program

- The lexing buffer is created by `lexbuf = Lexing.from_channel stdin` from the keyboard standard input (`stdin`)
- The lexing buffer `lexbuf` is passed as a parameter to the parser `Parser.prog` of programs.
- The `Lexer` module is created by `ocamllex` from the lexer specification and `Lexer.token` is the set of lexemes to be recognized.
- An exception is raised and captured in case of lexing or parsing error.
- Else, the parser returns an abstract syntax tree which is printed (in parenthesized form).

Compilation I

- The compilation through `make` [Feldman, 1979]

```
1  # file "makefile"
2
3  OCAMLBUILD := ocamlbuild -use-menhir -menhir "menhir --infer --explain"
4
5  all: delete built examples
6  built:
7          $(OCAMLBUILD) main.native
8
9  examples:
10         @echo "# using the parser:"
11         @echo ";"| ./main.native
12         @echo "x = 42;"| ./main.native
13         @echo "break;"| ./main.native
14         @echo "break; x = 7;"| ./main.native
15         @echo "x = 7; ; break; "| ./main.native
16         @echo "{}" | ./main.native
```

Compilation II

```
17          @echo "x=-10-20--30;" | ./main.native
18          @echo "x=1; y=2;" | ./main.native
19          @echo "{x=10; ; y=20;}" | ./main.native
20          @echo "if (1-2<3-4-5) x=-x;" | ./main.native
21          @echo "if (0<x) if (x<0) x=1; else if (x<0) { x=2; x=3; } else
                { x=4; x=5; x=6; }" | ./main.native
22          @echo "while (0<1){x = x - 1;} x = 42;" | ./main.native
23          @echo "while (0<1){}" | ./main.native
24          @echo "x=x-1;while (0<1){x=x-1;if(x<2)break;};" | ./main.native
25          @echo "x=-10; while (x<0) if (x<0) if (0<x) x=-x;" | ./main.
               native
26          @echo "x=-10; while (x<0) { x=x-1; break; }; x= 10;" | ./main.
               native
27          @echo "x=0; while (x<0) { while (x<0) x=x-1; x= 10; }; x= 100;"
                 | ./main.native
28          @echo "x=0; while (x<0) { while (x<0) x=x-1; break; }; x= 100;"
                 | ./main.native
```

Compilation III

```
29          @echo "x=x-1; while (0<1) { x=x-1; if (2 < x) break; };" | ./
                main.native
30          -@echo "x=10; while (x>0) x=x-1;" | ./main.native
31          @echo "# end"
32
33  delete:
34          rm -f *~ .*~ main.native
35          ocamlbuild -clean
```

- The result is the following (syntax trees are printed in parenthesized form)

```
# using the parser:
...
x = (x - 1); (while (0 < 1) { x = (x - 1); (if (x < (0 - 2)) break; ) } ) ;
At offset 14: unexpected character.
# end
```

en.wikipedia.org/wiki/Make_(software)

Builder I

- `ocamlbuild` [Durak and Pouillard, 2008] automatically determines the sequence of calls to the OCaml compiler to build a native OCaml-centric software project.
- This can be done manually by the following
- `makefile`

```
all:
	ocamllex -q lexer.mll
	menhir --infer --explain parser.mly
	ocamlc -c parser.mli
	ocamlc -c lexer.ml
	ocamlc -c parser.ml
	ocamlc -c main.ml
	ocamlc parser.cmo lexer.cmo main.cmo -o main
```

`ocaml.org/learn/tutorials/ocamlbuild/`

Conclusion

Conclusion

- Popular lexer and parser generators are `lex` [Lesk, 1975] and `yacc` [Johnson, 1975] i.e. `ocamllex` and `ocamlyacc` [Leroy, Doligez, Frisch, Garrigue, Rémy, and Vouillon, 2021, Chapter 12] for the OCaml functional programming language [Leroy, Doligez, Frisch, Garrigue, Rémy, and Vouillon, 2021].
- We prefer `menhir` [Pottier and Régis–Gianas, 2006, 2020] to `ocamlyacc` since it reports errors with respect to the grammar (and not respect to the more obscure parser code).
- We have introduced parsing from an empirical practical user point of view.
- For the theoretic background, see [Sippu and Soisalon–Soininen, 1988, 1990].
- Parsers are abstract interpretations of the semantics of grammars [P. Cousot and R. Cousot, 2003, 2011].

Bibliography

Bibliography I

Cousot, Patrick and Radhia Cousot (2003). "Parsing as Abstract Interpretation of Grammar Semantics.". *Theor.Comput.Sci.* 290.1, pp. 531–544.

— (2011). "Grammar Semantics, Analysis and Parsing by Abstract Interpretation.". *Theor.Comput.Sci.* 412.44, pp. 6135–6192.

Durak, Berke and Nicolas Pouillard (Sept. 2008). "The `Ocamlbuild` Users Manual.". Institut National de Recherche en Informatique et en Automatique. `https://ocaml.org/learn/tutorials/ocamlbuild/`.

Feldman, Stuart I. (1979). "Make—A Program for Maintaining Computer Programs.". *Softw., Pract.Exper.* 9.4, pp. 255–65.

Bibliography II

Johnson, Stephen C. (1975). "Yacc: Yet Another Compiler–Compiler.". *AT&T Bell Laboratories Computer Science Technical Reports.* 32.

Leroy, Xavier, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon (Apr. 7, 2021). "The OCaml System, Release 4.12, Documentation and User's Manual.". Institut National de Recherche en Informatique et en Automatique. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

Lesk, Michael E. (Oct. 1975). "Lex—a Lexical Analyzer Generator.". *AT&T Bell Laboratories Computer Science Technical Reports.* 39.

Pottier, François and Yann Régis–Gianas (2006). "Towards Efficient, Typed LR Parsers.". *Electr.Notes Theor.Comput.Sci.* 148.2, pp. 155–180.

Bibliography III

Pottier, François and Yann Régis–Gianas (Nov. 22, 2020). "Menhir Reference Manual.". Institut National de Recherche en Informatique et en Automatique. http://gallium.inria.fr/~fpottier/menhir/manual.pdf.

Sippu, Seppo and Eljas Soisalon–Soininen (1988). *Parsing Theory—Volume I: Languages and Parsing*. Vol. 15. EATCS Monographs on Theoretical Computer Science. Springer. https://doi.org/10.1007/978-3-642-61345-6.

— (1990). *Parsing Theory—Volume II: LR(k) and LL(k) Parsing*. Vol. 20. EATCS Monographs on Theoretical Computer Science. Springer.

The End, Thank you