Virtual Machines
Lecture 9
–
Lexical Analysis and Syntactic Analysis and Fourth Assignment

Compiler/Interpreter Architecture

Architecture of a compiler is pipe and filter

- Compiler is one long chain of filters, which can be split into two phases
- **Front end:** translate source code into a tree data structure called ***abstract syntax tree*** (AST)
- **Back end:** translate AST into machine code
-

Front end of compilers and interpreters largely the same:

- ***Lexical analysis*** with lexer
- ***Syntactic analysis*** with parser
- *Semantic analysis-TypeSystem*

Lexical Analysis

Character stream:

```
if x=0 then  1 else fact(x-1)
```

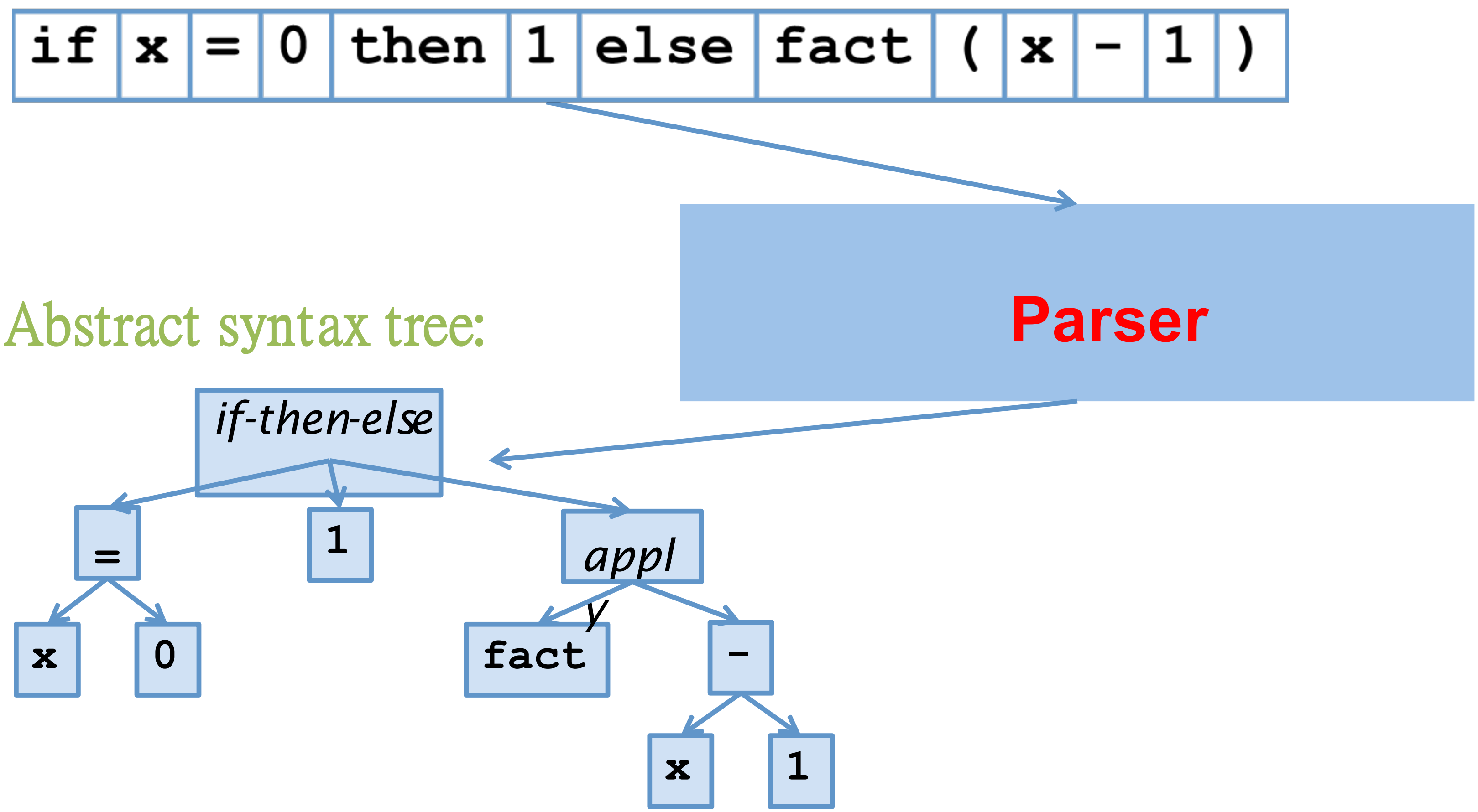
Lexer

Token stream:

`if`	`x`	`=`	`0`	`then`	`1`	`else`	`fact`	`(`	`x`	`-`	`1`	`)`

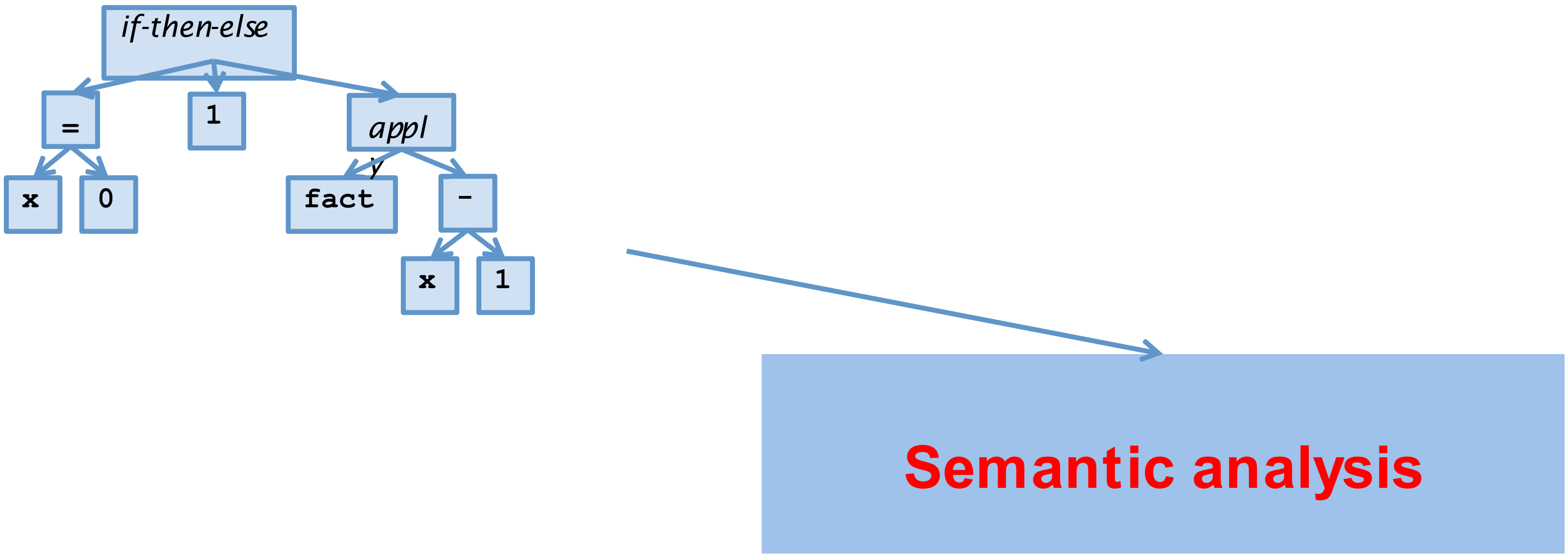
Syntactic Analysis

Token stream:



Semantic Analysis –Type System

Abstract syntax tree:



- accept or reject program
- ***decorate*** AST with types
- etc.

Fourth Assignment

–

CoreJava Lexer and Parser

Fourth Assignment – 25% of the final grade

Please use ocamllex and menhir in order to implement in Ocaml a lexer and a parser for CoreJava language. The input is a text file that contains the CoreJava program. The output of the lexical and syntactic analysis is the CoreJava AST(Abstract Syntax Tree) corresponding to the input CoreJava program.

The following slides present some examples of using ocamllex and menhir that help you to do your assignment.

References

- RealWorld Ocaml, Chapter16. Parsing with Ocamllex and Menhir. https://realworldocaml.org/v1/en/html/parsing-with-ocamllex-and-menhir.html
- Menhir reference manual. http://gallium.inria.fr/~fpottier/menhir/
- Some discussion groups. http://stackoverflow.com/questions/9897358/ocaml-menhir-compiling-writing
-

OCamllex and Menhir

OCamllex

- is a lexer/scanner generator
- Input: a sequence of token definitions
- Output: a lexer/scanner
    - available as an OCaml module

Menhir

- is a parser generator
- Input: an LR(1) context-free grammar
- Output: a parser for the defined language
    - available as an OCaml module

Expression Language Example

- We are going to use the same very simple expression language.
- Please find attached the following files:
  - ast.ml: the AST of the expression language
  - main.ml: the interpreter for the expression language
  - parser.mly: the input file to Menhir. It describes the syntax of the expression language.
  - lexer.mll: the input file to ocamllex. It describes the tokens of the expression language

Expression Language AST (see ast.ml)

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Let of string*expr*expr
```

Expression Language interpreter (see main.ml)

- The first three functions in this file, **subst, step, and multistep** should be familiar.
- (* **[parse s]** is the AST corresponding to the concrete syntax of expression [s]. *)

  val parse : string -> expr

- (* **[interp s]** parses the string [s] into an AST, interprets the AST, and yields the resulting integer value. *)

  val interp : string -> int

Expression Language interpreter (see main.ml)

```
let parse s = let lexbuf = Lexing.from_string s in
              let ast = Parser.prog Lexer.read lexbuf in
                    ast
```

- uses the standard library's Lexing module to create a lexer buffer from a string (use Lexing.from_channel to read from a file text, see Chapter 16 examples)
- then lexes and parses the string into an AST, using Lexer.read and Parser.prog.
- Lexer and Parser modules are code that is generated automatically during the compilation process by ocamllex and menhir:
  - ocamllex produces lexer.ml from input file lexer.mll.
  - menhir produces parser.ml from input file parser.mly.

Expression Language Interpreter (see main.ml)

```
let interp e = e |> parse |> multistep |> extract_value
```

- the main function of the program
- It uses the library composition operator:

```
val (|>) : 'a -> ('a -> 'b) -> 'b
```

  - Reverse-application operator: x |> f |> g is exactly equivalent to g (f (x)).

Expression Language Lexer (see lexer.mll)

- lexer.mll is the input for ocamllex
- Please read carefully the comments from the file lexer.mll
- For detailed explanations please see Chapter 16, section: Defining a Lexer

Expression Language Parser (see parser.mly)

- parser.mly is the input for menhir
- Please read carefully the comments from the file parser.mly
- For detailed explanations please see Chapter 16,especially the sections:
  - Parsing Sequences
  - Bringing it all together (for errors treatment. However error treatment is not required in your assignment)

Generated Parser Conflicts

- Please consult the reference manual of Menhir
- In the following slides we will see some conflicts examples and the solutions for them
- For your assignment please introduce as many terminals as possible in order to avoid the conflicts

Our Favorite Grammar in ocamllex and menhir

```
{ open Parser
  let get = Lexing.lexeme
}
(* Helpers *)
 let tab   = '\009'
 let cr    = '\013'
 let lf    = '\010'
 let eol   = cr | lf | cr lf


rule  token = parse
 | eol              {   token lexbuf }
 | (' ' | tab)      {  token lexbuf }
 | eof              {  EOF }
 | '+'              {  PLUS }
 | '-'              {  MINUS }
 | '*'              {  STAR }
 | '/'              {  SLASH }
 | '('              {  LPAR }
 | ')'              {  RPAR }
 | ('x'|'y'|'z')    {  ID(get lexbuf) }
```

```
%{
  (* Put OCaml helper functions here *)
%}

%token EOF
%token PLUS MINUS STAR SLASH LPAR RPAR
%token <string>ID

%start <unit> start  /* entry point */
%%


/* Productions */
start   :  expr EOF                  { };

expr      :  expr  PLUS term         { }
          |  expr  MINUS term        { }
          |  term                    { };


term      :  term STAR factor      { }
          |  term SLASH factor     { }
          |   factor               { };


factor   :  ID                       { }
         |  LPAR expr RPAR           { };
```

Generated Modules

```
-rw-r--r-- 1 jmi users 6384 Sep  1 22:53 lexer.ml
-rw-r--r-- 1 jmi users  594 Sep  1 22:20 lexer.mll
-rw-r--r-- 1 jmi users  317 Sep  1 22:20 main.ml
-rw-r--r-- 1 jmi users 8554 Sep  1 22:53 parser.ml
-rw-r--r-- 1 jmi users  170 Sep  1 22:53 parser.mli
-rw-r--r-- 1 jmi users  480 Sep  1 22:20 parser.mly
-rw-r--r-- 1 jmi users 2416 Sep  1 22:53 parser.automaton
```

These are the input files

Generated Modules

```
-rw-r--r-- 1 jmi users 6384 Sep  1 22:53 lexer.ml

-rw-r--r-- 1 jmi users  594 Sep  1 22:20 lexer.mll
-rw-r--r-- 1 jmi users  317 Sep  1 22:20 main.ml
-rw-r--r-- 1 jmi users 8554 Sep  1 22:53 parser.ml
-rw-r--r-- 1 jmi users  170 Sep  1 22:53 parser.mli
-rw-r--r-- 1 jmi users  480 Sep  1 22:20 parser.mly
-rw-r--r-- 1 jmi users 2416 Sep  1 22:53 parser.automaton
```

These are the input files

These are the output files (run menhir with `-v`)

Generated Modules

```
-rw-r--r-- 1 jmi users 6384 Sep  1 22:53 lexer.ml
-rw-r--r-- 1 jmi users  594 Sep  1 22:20 lexer.mll
-rw-r--r-- 1 jmi users  317 Sep  1 22:20 main.ml
-rw-r--r-- 1 jmi users 8554 Sep  1 22:53 parser.ml
-rw-r--r-- 1 jmi users  170 Sep  1 22:53 parser.mli
-rw-r--r-- 1 jmi users  480 Sep  1 22:20 parser.mly
-rw-r--r-- 1 jmi users 2416 Sep  1 22:53 parser.automaton
-rw-r--r-- 1 jmi users 2416 Sep  1 22:53 parser.conflicts
```

These are the input files

These are the output files (run menhir with $-v$)

Menhir generates a conflicts-file in case of conflicts.

These two are useful for debugging conflicts.

The Main Application

```
let lexbuf = Lexing.from_channel stdin in
try
  Parser.start Lexer.token lexbuf
with
 | Failure msg  -> print_endline ("Failure in " ^ msg)
 | Parser.Error -> print_endline "Parse error"
 | End_of_file  ->
              print_endline "Parse error: unexpected     end of string"
```

An Ambiguous Grammar

$$X \rightarrow \Lambda \mid \mathtt{a}\ X \mid \mathtt{a}\ \mathtt{a}\ X$$

Any string in this language has exponentially many different parse trees

$\underbrace{\mathtt{a}\ \mathtt{a}\ \ldots\ \mathtt{a}}_{n}$ has exactly *Fib(n)* parse trees

The ocamllex + menhir version

```
{ open Parser }

rule token = parse
  | 'a'        { A }
```

```
%{ %}
%token  A EOF
%start  <unit>  main
%%

main : x EOF    { };

x    :          { }
     | A x      { }
     | A A x    { };
```

menhir is not happy...

```
$ menhir -v parser.mly
Warning: one state has reduce/reduce conflicts.
Warning: one reduce/reduce conflict was arbitrarily resolved. File
"parser.mly", line 11, characters 6-14:
Warning: production x -> A A x is never reduced.
Warning: in total, 1 productions are never reduced.
$
```

The parser table in `parser.automaton` contains conflicting actions!

`parser.conflicts` is more informative:

```
** Conflict (reduce/reduce) in state 3.
** Token involved: EOF
** This state is reached from main after reading:

A A x

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin to differ.)

main
x EOF // lookahead token appears (?)

** In state 3, looking ahead at EOF, reducing production
** x -> A x
** is permitted because of the following sub-derivation:

A x // lookahead token is inherited A x .

** In state 3, looking ahead at EOF, reducing production
** x -> A A x
** is permitted because of the following sub-derivation:

A A x .
```

Solution: Less Stupid Grammar

```
{ open Parser }

rule token = parse
  | 'a'        { A }
```

```
%{ %}
%token  A EOF
%start  <unit>  main
%%

main :    x EOF    { };

x     :              { }
      |  A x         { };
```

A Grammar for If-Statements

```
{ open Parser }

 let tab   = '\009'
 let lf    = '\010'
 let cr    = '\013'
 let eol   = cr | lf   | cr lf

rule  token  = parse
   | (eol|' '|tab)    { token lexbuf }
   | eof              { EOF }
   | "exp"            { EXP }
   | "if"             { IF }
   | "then"           { THEN }
   | "else"           { ELSE }
   | "assign"         { ASSIGN }
```

```
%{ %}

%token EOF
%token EXP IF THEN ELSE ASSIGN

%start <unit> main
%%

main : stm EOF                     { };

 stm
  : IF EXP THEN stm                { }
  | IF EXP THEN stm ELSE stm       { }
  | ASSIGN                         { };
```

menhir is not happy...

```
$ menhir -v parser.mly
Warning: one state has shift/reduce conflicts.
Warning: one shift/reduce conflict was arbitrarily resolved.
$
```

But the grammar does not appear to be stupid...

Again `parser.conflicts` is your friend:

```
** Conflict (shift/reduce) in state 5.
** Token involved: ELSE
** This state is reached from main after reading:

IF EXP THEN IF EXP THEN stm

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin to differ.)

main stm EOF (?)

** In state 5, looking ahead at ELSE, reducing production
** stm -> IF EXP THEN stm
** is permitted because of the following sub-derivation:

IF EXP THEN stm ELSE stm // lookahead token appears IF EXP
            THEN stm .

** In state 5, looking ahead at ELSE, shifting is permitted
** because of the following sub-derivation:

IF EXP THEN stm
            IF EXP THEN stm . ELSE stm
```

Solution: Less Natural Grammar

```
{ open Parser }

 let tab   = '\009'
 let lf    = '\010'
 let cr    = '\013'
 let eol   = cr | lf   | cr lf

rule  token  = parse
   | (eol|' '|tab)    { token lexbuf }
   | eof              { EOF }
   | "exp"            { EXP }
   | "if"             { IF }
   | "then"           { THEN }
   | "else"           { ELSE }
   | "assign"          { ASSIGN }
```

```
%{ %}

%token EOF
%token EXP IF THEN ELSE ASSIGN

%start <unit> main
%%

main : stm EOF                    { };

 stm
  : IF EXP THEN stm               { }
  | IF EXP THEN stm2 ELSE stm     { }
  | ASSIGN                        { };

stm2
  : IF EXP THEN stm2 ELSE stm2 { }
  | ASSIGN                        { };
```

Dangling Else Problem

- An example statement:

```
if exp then if exp then assign else assign
```

- To which `if` does the `else` belong?
- The first grammar is ambiguous
- Our modified grammar parses the string as:

```
if exp then (if exp then assign else assign)
```

The Palindrome Grammar

```
{ open Parser }

rule token = parse
  | '0'      { Zero }
  | '1'      { One }
```

```
%{ %}

%token  Zero One EOF

%start  <unit> main
%%

main :  pal EOF              { };

pal   :                      { }
      | One                  { }
      | Zero                 { }
      | One pal One          { }
      | Zero pal Zero      { };
```

menhir is not happy...

```
$ menhir -v parser.mly
Warning: 2 states have shift/reduce conflicts.
Warning: 2 states have reduce/reduce conflicts.
Warning: 6 shift/reduce conflicts were arbitrarily resolved.
$
```

Again `parser.conflict` is descriptive…

```
** Conflict (shift/reduce) in    state 2.
** Tokens involved: Zero One
** The following explanations    concentrate on token One.
** This state is reached from    main after reading:

One

** The derivations that appear below have the following common    factor:
** (The question mark symbol (?) represents the spot where the    derivations begin to
   differ.)

main pal EOF
(?)

** In state 2, looking ahead at One, reducing production
** pal ->
** is permitted because of the following sub-derivation:

One pal One // lookahead token appears
      .
…
```

No Solution!

- There is no LR(1) grammar for this language
-
- Some **grammars** are not LR(1)
- And some **languages** are not LR(1)
-
- Some **grammars** are ambiguous
- And some **languages** are ambiguous