Towards LR parsing in Miking - key ideas and challenges

Miking Workshop 2023

John Wikman
jwikman@kth.se

KTH Royal Institute of Technology

2023-11-23

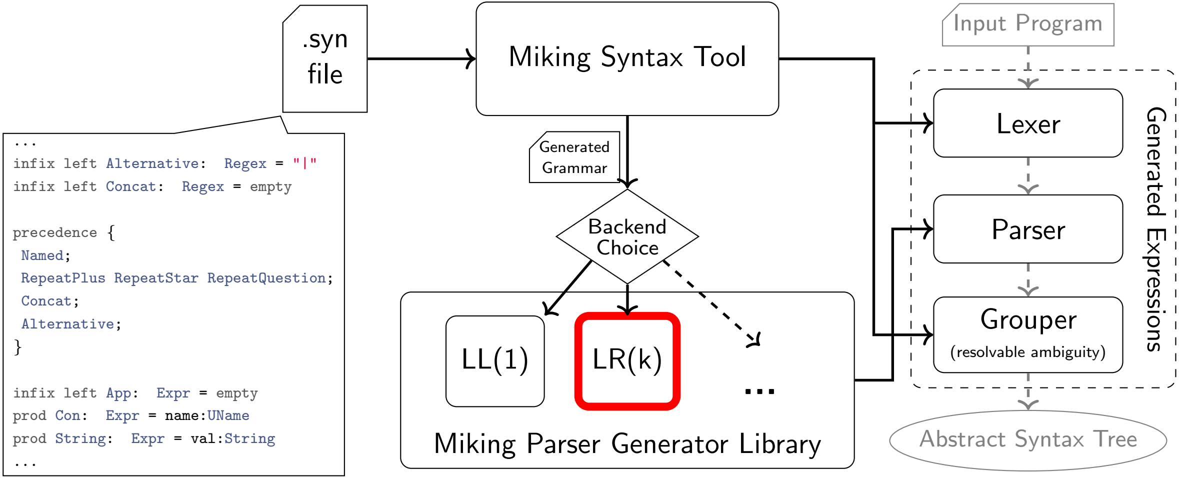
Outline

Parsing in Miking

What is a LR(k) Parser?

The Challenges of a LR(k) Parser

The Parsing Chain in Miking



```
...
infix left Alternative:  Regex = "|"
infix left Concat:  Regex = empty

precedence {
 Named;
 RepeatPlus RepeatStar RepeatQuestion;
 Concat;
 Alternative;
}

infix left App:  Expr = empty
prod Con:  Expr = name:UName
prod String:  Expr = val:String
...
```

Goals With Miking’s LR(k) Parser

1. Type-Safe

2. Side-Effect Free

3. Minimal Overhead

4. Manageable Code Size

What is a LR(k) Parser?

(a) $E \rightarrow S$ EOF

(b) $S \rightarrow (\ S\ )\ S$

(c) $\mid \epsilon$

States: Progress indicators in productions.

(a) $.S$ EOF

(a) $S.$EOF

(a) S EOF.

(b) $.(\ S\ )\ S$

(b) $(.S\ )\ S$

(b) $(\ S.)\ S$

(b) $(\ S\ ).S$

(b) $(\ S\ )\ S.$

(c) $\epsilon.$

What is a LR(k) Parser?

(a) $E \rightarrow S\ \mathtt{EOF}$

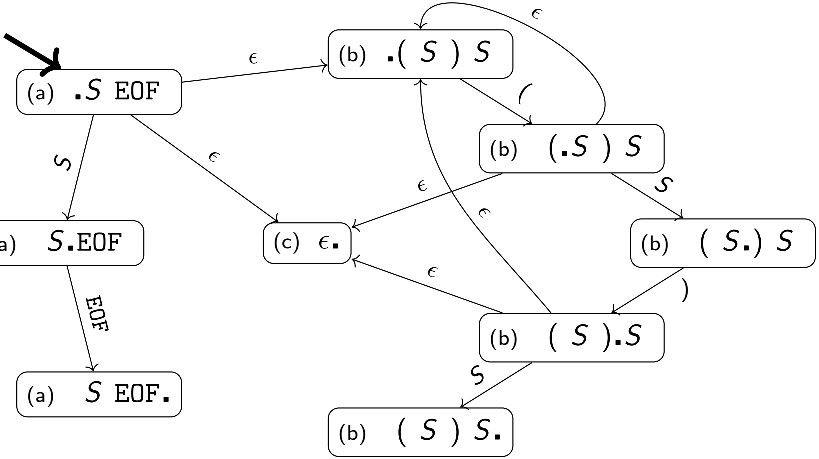
(b) $S \rightarrow (\ S\ )\ S$

(c) $\qquad\quad \mid\ \epsilon$

States: Progress indicators in productions.

Add transitions between states.

Result: Non-deterministic automaton.

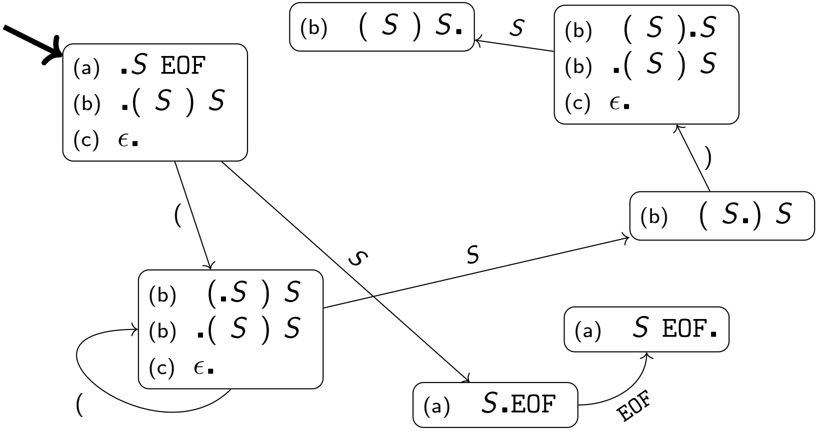


What is a LR(k) Parser?

(a) $E \rightarrow S$ EOF

(b) $S \rightarrow (\ S \ ) \ S$

(c) $\quad\quad \mid \epsilon$



Convert to deterministic automaton.

Issue: Shift-reduce conflicts.

What is a LR(k) Parser?

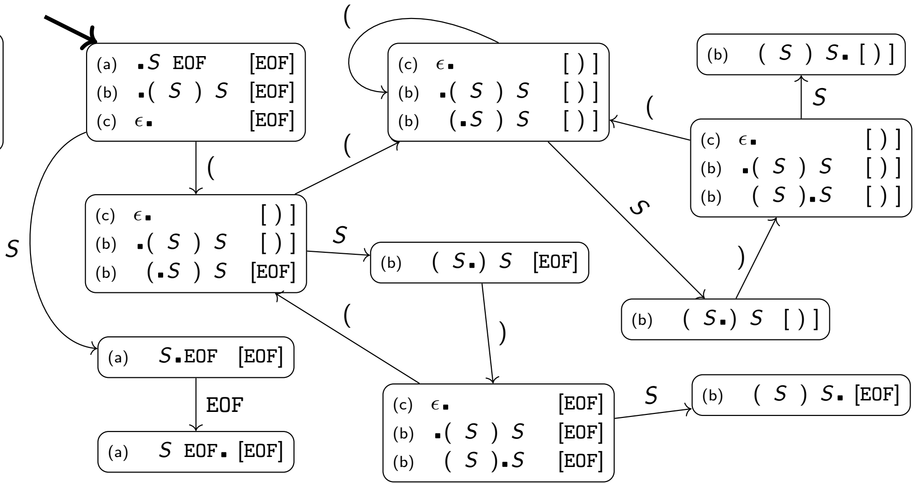
(a) $E \rightarrow S$ EOF

(b) $S \rightarrow (\; S \;) \; S$

(c) $\quad\quad\;\; | \; \epsilon$

Solution: Add production lookahead.

This example is LR(1).



Implementation Challenges

- Key issue:
  1. Naive LR(k) implementation & large language
  2. Leads to large automaton
  3. Which leads to code explosion
- Miking must be able to handle large languages
- Current LR(k) impl. generates 100k lines of MExpr code for large grammars
- Tricks exist to make the LR(k) parser more managable
  - But often depend on side-effects and coercing the type system

Implementation Challenges

Stack Implementation

Typically a single stack is used:

Type-safe and non-wrapping:

Type Unsafe:

```
let stack = [] in
...
let x: TypeX = ... in
let y: TypeY = ... in
let stack =
  cons (unsafeCoerce y) (
  cons (unsafeCoerce x)
  stack)) in
...
let vx: TypeX =
  unsafeCoerce (head stack) in
let stack = tail stack in
...
```

Type Wrapping:

```
type WrapType in
con WrapX: TypeX -> WrapType in
con WrapY: TypeY -> WrapType in
let stack: [WrapType] = [] in
...
let x: TypeX = ... in
let stack = cons (WrapX x)
                 stack in
...
let vx: TypeX =
  match head stack
  with WrapX x then x
  else error "oops"
in
```

Multi-Type Stack:

```
let stack =
  {typeX = [], typeY = [], ...}
in
...
let x: TypeX = ... in
let stack = {stack with typeX =
  cons x stack.typeX} in
...
let vx: TypeX =
  head stack.typeX in
let stack = {stack with typeX =
  tail stack.typeX} in
...
```

Implementation Challenges

Side-Effect Implementation

With side-effects:

Update and Fetch Elsewhere:

```
let pushX = lam x: TypeX.
  modref stackX (cons x (deref stackX))
...
recursive
let state00 = lam lh.
  ...
  switch lh
  case SomeToken x then
   pushX x;
   let tok = nextToken () in
   state01 tok
  case ... then
    ...
  end
```

Without side-effects:

Threading States, Modify in Function:

```
recursive
let state00 = lam lh. lam stack.
              lam lexstate.
  switch lh
  case SomeToken x then
    let stack = {stack with typeX =
                cons x stack.typeX} in
    switch nextToken lexstate
    case ResultOk (tok, lexstate) then
     state01 tok stack lexstate
    case ResultErr e then
     ResultErr e
    end
  case ... then
    ...
  end
```

Concluding Remarks

- The type-safety and functional nature allows the compiler to make more assumptions
- Issue: This places more code in generated functions
- Cannot apply the tricks
- **Next steps:** Investigate more sophisticated LR(k) transformations