Scanning, Parsing and Pretty-Printing

Koen van Ingen, s4058038
Joost Rijneveld, s4048911

Radboud University Nijmegen

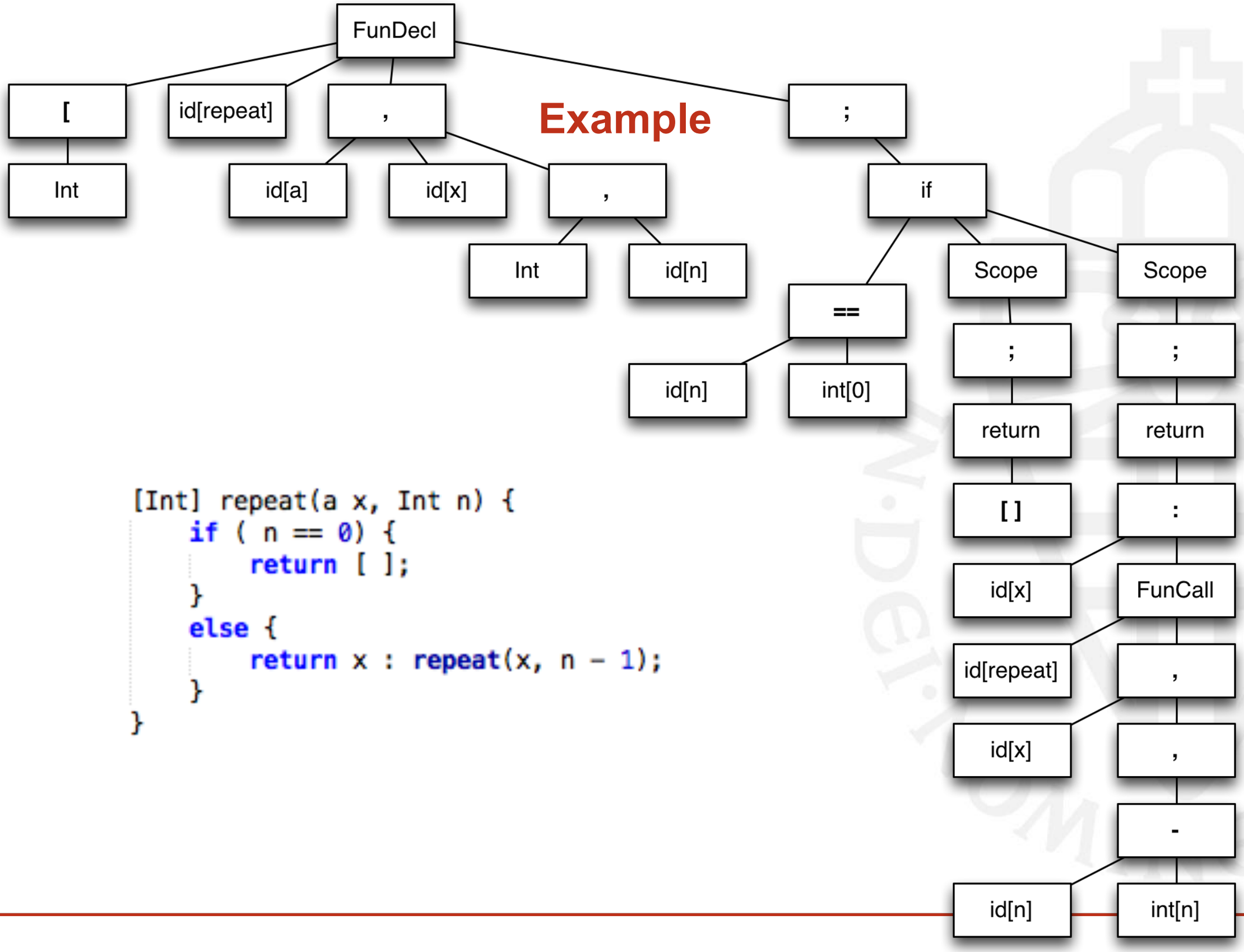
Grammar / strategy

- Top-down ‘Recursive Descent’ parser
- Tail recursion for left-associativity

```
Exp     = Op2 Exp
        |  Op1 Exp
        |  '( ' Exp ') '
        |  '[] '

Op2     = '||' | '&&'
        |  '+' | '-'
        |  '*' |  '/' | '%'
        |  ':'
Op1     = '-' | '!'


Fargs   = [ FArgs '.' ] Type id
```

```
Exp     = ExpOr
ExpOr   = ExpAnd    (‘||’ ExpAnd)*
        ...
ExpAdd  = ExpMult    (('+' | '-') ExpMult)*
ExpMult = ExpCon     (('*' | '/' | '%') ExpCon)*
ExpCon  = ExpUn     ':' ExpCon
        |  ExpUn
ExpUn   = ('-' | '!') ExpUn
        |  ExpBase
        ...

Fargs   = Type id [ ',' Fargs ]
```

Example




```
[Int] repeat(a x, Int n) {
    if ( n == 0) {
        return [ ];
    }
    else {
        return x : repeat(x, n - 1);
    }
}
```

Program details

- **Python**
- Imperative, but with functional magic
- Scanner: maintain a list of candidates until empty
- Parser: transform grammar into parsing functions

```
def tail_recursion(fn, ops, tokens):
    # fn is the next function for the recursive descent
    t = fn(tokens)
    while tokens and tokens[0].type in ops:
        tok = tokens.popleft()
        t_right = fn(tokens)
        t = Node(tok, t, t_right)
    return t

parse_exp_mult  = partial(tail_recursion, parse_exp_con, ['*', '/', '%'])
parse_exp_add   = partial(tail_recursion, parse_exp_mult, ['+', '-'])
parse_exp_cmp   = partial(tail_recursion, parse_exp_add, ['<', '<=', '>', '>='])
parse_exp_eq    = partial(tail_recursion, parse_exp_cmp, ['==','!='])
parse_exp_and   = partial(tail_recursion, parse_exp_eq, ['&&'])
```

Code example

```
def parse_stmt(tokens):
    tok = tokens.popleft()
    # ...
    elif tok.type == 'if':
        pop_token(tokens, '(')
        condition = parse_exp(tokens)
        pop_token(tokens, ')')
        if_stmt = parse_stmt(tokens)
        if tokens and tokens[0].type == 'else': # optional else
            pop_token(tokens, 'else')
            return Node(tok, condition, if_stmt, parse_stmt(tokens))
        return Node(tok, condition, if_stmt, None)
    # ...

def pop_token(tokens, literal):
    tok = tokens.popleft()
    if tok.type != literal:
        raise Exception("Line {}:{} Expected '{}', but got: {}".
                        format(tok.line, tok.col, literal, tok.type))
    return tok
```