Parsing

CS565

Purdue University

January 20, 2010

Lexing

```
print("Hello!")
```

$\rightarrow$

IDENTIFIER(*print*)
LEFT_PAREN
STRING_LITERAL(*Hello*!)
RIGHT_PAREN

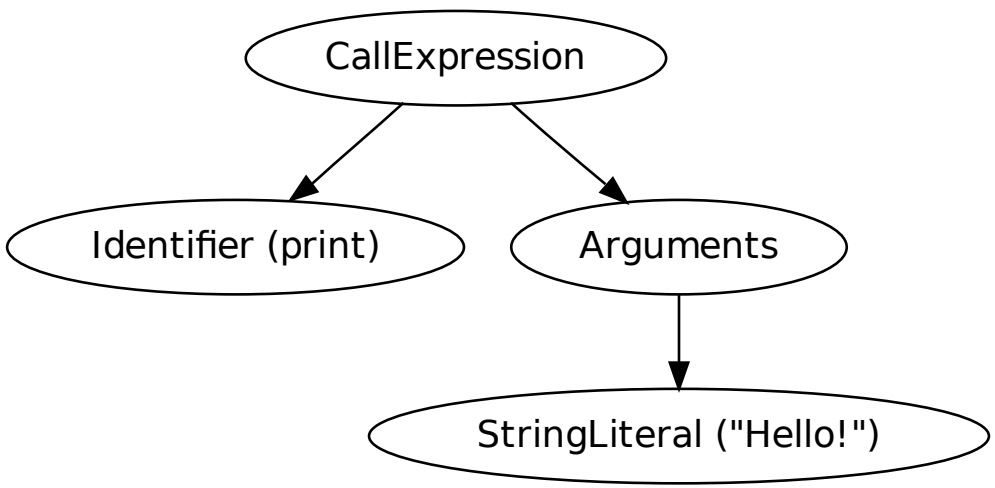
Parsing

```
print("Hello!")
```

$\rightarrow$

IDENTIFIER(*print*)
LEFT_PAREN
STRING_LITERAL(*Hello*!)
RIGHT_PAREN

$\rightarrow$



Grammars

- Language syntax defined by a grammar
- Grammar does *not* define semantics
- Often separate lexical grammar (defining tokens) from syntactic grammar

BNF

Grammars are usually expressed in BNF
(Backus N{ormal,aur} Form):

```
num :
    NUMERIC_LITERAL

mul :
    num
  | mul '*' num
  | mul '/' num

add :
    mul
  | add '+' mul
  | add '-' mul
```

Extensions to BNF

Many extensions to BNF exist. The only extensions in the ECMAScript spec are

- Optional symbols
- No-line-terminator symbols
- Lookahead

Parsers

- Many types of parsers exist
- Top-down: LL(x), recursive-descent, etc.
- Bottom-up: LALR, SLR, etc.
- But this isn't the compilers course ...

Recursive-Descent Parsing

- Every nonterminal becomes a function
- This function takes a token stream (e.g. as an array and index) and returns a parsed tree
- (Store the tree however you want)

```
AssignmentStatement :
    IDENTIFIER ':=' Expression

function parseAssignmentStatement(toks[], tokindex) {
    if (toks[tokindex] is not an IDENTIFIER) return FAIL
    ident := toks[tokindex]
    tokindex := tokindex + 1
    if (toks[tokindex] is not ':=') return FAIL
    tokindex := tokindex + 1
    exp := parseExpression(toks, ref tokindex)
    if (exp is not FAIL) return AssignmentStatement(ident, exp)
    return FAIL
}
```

Recursive Descent: Problems

Recursive descent is easy to write, but has problems:

- Multiple productions, one prefix of the other

```
Try : 'try' '{' Stmt '}' 'catch' '(' ID ')' '{' Stmt '}'
    | 'try' '{' Stmt '}' 'catch' '(' ID ')' '{' Stmt '}'
      'finally' '{' Stmt '}'
```

- Order is important: First will be preferred if ambiguous
- Left recursion

Left Recursion

Left recursion is extremely common in grammars

```
mul : mul '*' num
    | num
```

But a direct translation doesn't work

```
function parseMul(toks[], tokindex) {
    left := parseMul(toks, ref tokindex) # whoops!
    ...
}
```

Left Recursion — The Trick

How "the book" wants you to learn it:

```
mul : mul '*' num
    | num


mul' : (nothing)
     | '*' num mul'

mul : num mul'
```

Left Recursion — The Real Trick

```
mul : mul '*' num
    | num

function parseMul(toks[], tokindex) {
    left := parseNum(toks, ref tokindex)
    if (left is FAIL) return FAIL

    loop {
        if (toks[tokindex] is not '*') break
        tokindex := tokindex + 1

        right := parseNum(toks, ref tokindex)
        left := Mul(left, right)
    }

    return left
}
```

CS565 Canonical Parser

Demo and code

JavaScript

- The JavaScript (ECMAScript) grammar is in the ECMAScript spec
- Both strewn throughout section 10 and onward,
- and compactly in annex A
- Search for "Program :" to start off

Vagaries of JavaScript

JavaScript is a fairly easy language to parse but it's not the easiest

Semicolons

JavaScript has automatic semicolon insertion. See ECMA-262 7.9.
My recommendation:

- Create a function to parse "semicolons":
  1. Check if there is in fact a semicolon
  2. Check if EOF or right brace
  3. Check if, whatever there was, there was a line terminator first
- Don't worry too much about semicolons, start by only accepting ';' and improve it later

CallExpression and NewExpression

CallExpression and NewExpression are written strangely to avoid an ambiguity. Needs to be rewritten to avoid another ambiguity with recursive-descent:

```
LeftHandSideExpression :
    CallExpression
    NewExpression

CallExpression :
    MemberExpression
    CallExpression Arguments
    CallExpression [ Expression ]
    CallExpression . IdentifierName
```

Try-catch-finally

What, you don't remember my mentioning this just a few slides ago?

```
TryStatement :
    'try' Block Catch Finally
    'try' Block Catch
    'try' Block Finally
```

Conditional Keywords

- 'get' and 'set' are supposed to be conditional keywords
- Conditional keywords are a pain
- Lexer treats them as full keywords

Optional Symbols

When parsing an optional symbol returns nothing, put the node "None" in your parse tree.

Lexer vagaries

- Technically, "-10" should be a single numeric literal. Instead, it's a unary '-' followed by the numeric literal 10
- ECMAScript is written to support HTML comments. Ha, no.

Next Week – JavaScript!

Next week we'll talk about how JavaScript works, then later how to interpret it.