Grammars and context-free languages

Grammars

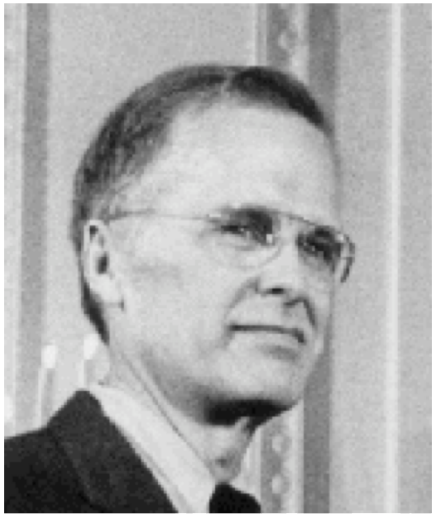
A grammar is (N, Σ, P, S)

- N is a set of non-terminal symbols
- Σ is an alphabet of terminal symbols (tokens)
- P is a set of productions
- S ∈ N is the start symbol

Productions are rewrite rules: Starting with S, produce all strings in the language (with terminal symbols only).

Backus-Naur Form

Or is it Backus Normal Form?



John Backus, 1977 Turing Award winner and developer of FORTRAN language and compiler (25,000 lines of machine code). Co-developer of BNF. Died 2007 in Ashland, Oregon.

Peter Naur, 2005 Turing Award winner, editor of *Algol 60 report*. Recommended Backus' grammar notation with minor revisions.

Example: parenthesized lists (S-expressions)

⟨Sexp⟩ ::= (⟨List⟩)
⟨Sexp⟩ ::= atom
⟨List⟩ ::= ⟨Sexp⟩ ⟨List⟩
⟨List⟩ ::=

Grammars and Languages

$$L(G)=\{ x \in \Sigma^* \mid S \Rightarrow^+ x\}$$

In other words:

- Each string in L(G) contains terminal symbols only
- Each string is derived by repeated application of rewrite rules

Hierarchy of grammars and languages

Type 0: No restrictions

Type 1: Context sensitive

$xAy \rightarrow xzy$

Type 2: Context free

$A \rightarrow \gamma$

Where γ is any string from $(N \cup \Sigma)*$

Type 3: Right linear

$A \rightarrow xB$

where $x \in \Sigma$ and $B \in N$, or $B = \lambda$

Types 0 and 1 are too general — can't be efficiently parsed

Type 3, right linear, is same as regular expressions (see why?)

Type 2, context free, is used for programming languages

Parsing (Recognition)

- Apply productions "backwards," replacing right-hand-side by left-hand-side.
- Beginning with a string of terminal symbols, produce S.
- History of rewrites is "derivation," encoded in parse tree.

The most general parsers are searches with back-tracking.

example: Bison GLR parser uses breadth-first search.

Practical parsing for programming languages should* be "left-to-right" in one pass.

*Opinion: GLR parsing is useful in reverse engineering and dealing with messy data. If your language requires GLR parsing, it's broken.

Parsing

The problem: Given $x \in \Sigma^*$, and a grammar $G = (N, \Sigma, P, S)$, find the derivation of x from S

Derivation is a sequence of productions to produce x

Also:
If $x \notin L(G)$, x must be rejected (but: report error and continue)

There could be more than one derivation — in which case we say G is ambiguous

Recursive descent

The method of choice for small, hand-built parsers and for very vast production compilers with excellent error handling

Design the grammar so that we can always predict the right production by looking at the next token. This is "LL(1)".

The code is simple and follows the grammar. The grammar may be convoluted because one token isn't much to go by.

(Let's create a grammar for nested lists on the board)

S-expression parser excerpt

Sexp ::= ‘(‘ List ‘)’

Sexp ::= atom

```
void Sexp() {
   if (token == ATOM) {          ## Peek at the next token
        match(ATOM);             ## Consume it (after checking)
   } else if (token == LPAREN) {
        match(LPAREN);  List();    match(RPAREN);
   } else {
        printf("\n***Unexpected token %s"
        " while trying to recognize Sexp***\n",
        token_name(token));
        punt();
} }
```

Recursive descent / LL1 parsing requires predictions

If a non-terminal could be expanded by multiple predictions, we must predict which one to try by peeking at the input

LL(1) means we peek at just the next token. LL(k) means we can peek at up to k tokens (some fixed number). Parser generators like Antlr produce LL(k) parsers for arbitrary k.

Assume a lookahead of 1 (peek at next token). How can we predict?

Predicting a production

Given

a non-terminal N that we are trying to parse

multiple productions N ::= α

(where α is a string of terminals and non-terminals)

a single "lookahead" symbol x that should start N

We want to predict a production if

x can be the ***first*** token in α, or

α is "nullable" (can expand to zero symbols)
and x can ***follow*** N

First and Follow

First(α): The terminal symbols (tokens) that can appear at the beginning of α

Follow(N): The terminal symbols that can appear immediately after N

Note that "First" is about α, the right hand side of the production; "Follow" is about N, the non-terminal symbol.

Before going on, try to devise algorithms for computing First and Follow wherever needed in a grammar.

Computing First(α)

Remember α is a sequence of zero or more non-terminal and terminal symbols. First(α) should be a set of terminal symbols (tokens) and a flag to indicate whether α is "nullable"

Cases:

α is a terminal symbol t: First(α) = { t }

α is empty: First(α) is { } and α is nullable

α is βγδ: First(α) ⊇ First(β)

and if β is nullable, then First(α) ⊇ First(βδ)

This is actually a simple iterative algorithm: Keep map from suffixes of productions to follow sets and "nullable" flag, iterate rules until nothing changes.

Computing Follow(N)

Recall N is a non-terminal symbol (not a single production)

$ is in Follow(S), where S is “start symbol” and $ is “EOF”

If A ::= αNβ, then Follow(N) ⊇ First(β)

If A ::= αNβ and β is nullable, then Follow(N) ⊇ Follow(A)

Also solvable by an iterative fixed-point algorithm: Just apply the rules until nothing changes.

How do we know this algorithm will halt?

Left recursion

Suppose our grammar only has integers and +, -, *, /

We would like to write

Expr ::= Expr + Term | Expr – Term | Term

Term ::= Term * integer | Term / integer | integer

but we can't, because it would be an infinite loop. We could write

Expr ::= Term + Expr | Term – Expr | Term

Term ::= integer * Term | integer / Term | integer

but we shouldn't. Why?

Expressing priority

Expr ::= Term + Expr | Term – Expr | Term

Term ::= integer * Term | integer / Term | integer

accepts the same language (sequences of symbols) as

Expr ::= Expr + Term | Expr – Term | Term

Term ::= Term * integer | Term / integer | integer

but it builds the wrong trees! What can we do?

Refactoring left recursion

We wanted

Expr ::= Expr + Term

Expr ::= Term

But we didn't want left recursion. So we refactor

Expr ::= Term ETail

Etail ::= + Term Etail

Etail ::= λ

(not as bad as it looks)

Refactored left recursion in recursive descent

```
def _expr(stream: TokenStream) -> expr.Expr:
    """ expr ::= term { ('+'|'-') term } """

    left = _term(stream)
    while stream.peek().value in ["+", "-"]:
        op = stream.take()
        right = _term(stream)
        if op.value == "+":
            left = expr.Plus(left, right)
        elif op.value == "-":
            left = expr.Minus(left, right)
        else:
            raise InputError(f"What's that op? {op}")
    return left
```

From a small hand-written parser used for a project in CIS 211.

In practice

For very small languages,

- Write recursive descent parser by hand, refactoring left recursion as loops (like example prior slide)
- Calculate first and follow sets by hand as needed

For larger languages,

- Tools like Antlr and JavaCC accept (refactored) grammars and compute first and follow sets for you
- They mostly work well, although sometimes the grammar is difficult to refactor, and it's easy to mess up

Both hand-written and generated LL(1) parsers are in wide use.