CONTEXT-FREE GRAMMARS

Allows recursion (needs stack, so PDA): Nonterminals defined in terms of themselves. Rule: Only single nonterminal, no terminals on LHS of productions. Supports structure like nested (()). G=(S,N,T,P): (Start, NonTerm, Term, Prods). e.g. in BNF:

expr $\longrightarrow$ `id` | `number` | - *expr* | (*expr*)
| *expr op expr*
op $\longrightarrow$ + | - | * | /

(note paren-matching trick). More powerful than RE: easy to do concat. and alternate, thus Kleene closure, but can also define things in terms of themselves, which RE's can't. BNF sometimes augmented: e.g.

id-list $\longrightarrow$ *id (, id)**

(similarly Kleene + and note | is not needed.) Parser doesn't distinguish one terminal symbol (e.g. `id`) from another, but semantics has to, so scanner saves spellings.

DERIVATIONS

Starting with grammar's start symbol and rewriting non-terminals using the rules lets you generate (derive) syntactically valid strings of terminals and non-terminals, called *sentential forms*, which are the *yield* of the derivation. Here's a *rightmost* or *canonical* derivation.

expr $\Longrightarrow$ *expr op expr*
$\Longrightarrow$ *expr op* `id`
$\Longrightarrow$ *expr* `+ id`
$\Longrightarrow$ *expr op expr* `+ id`
$\Longrightarrow$ *expr op* `id + id`
$\Longrightarrow$ *expr* `* id + id`
$\Longrightarrow$ `id * id + id`
(slope) (x) (intercept).

DERIVATION TYPES

Leftmost Derivation: replace leftmost non-terminal at each step.

Rightmost Derivation: replace rightmost non-terminal at each step.

Ambiguous Grammar: has multiple left- or rightmost derivations for a single sentential form.

CFG ADVANTAGES

Precise syntactic specification of programming language.
Easy to understand, avoids *ad hoc* definition.
Easy to maintain and add new features.
Can automatically construct efficient parser.
Parser construction reveals ambiguity or other infelicities.
Imparts structure to language.
Supports syntax-directed translation (parser in charge).
Strictly more powerful than REs (strict language inclusion).
- Need to "look ahead": potential $O(n^3)$ problem. Keep linear by 'good' (parser-aware) language design.

PARSE TREES

Represent a derivation as a parse tree, with root the start symbol, internal nodes are nonterminals, leaves are terminals, children of node T are symbols on RHS of some production for T in grammar. Generated terminal string has ≥ 1 parse tree, every tree represents a sting gen'd by grammar. The simple grammar

expr $\longrightarrow$ `id` | `number` | - *expr* | (*expr*)
| *expr op expr*
op $\longrightarrow$ + | - | * | /

allows two possible parse trees for `slope * x + intercept`. It is thus an *ambiguous* grammar. There are infinitely many grammars for any CFL, and ambiguous grammars are much less useful in CS than unambiguous ones. Also to be avoided are grammars with *useless* symbols (nonterminals that can't generate any string of terminals (e.g. $A \longrightarrow B$.)) or terminals that never appear in any yield. Grammars may be put into *canonical form*, which simplifies applications and analysis.

AN UNAMBIGUOUS GRAMMAR

expr $\longrightarrow$ *term* | *expr add-op term*
term $\longrightarrow$ *factor* | *term mult-op factor*
factor $\longrightarrow$ `id` | `number` | `-` `factor` | `(` `expr` `)`
add-op $\longrightarrow$ `+` | `-`
mult-op $\longrightarrow$ `*` | `/`

This unambiguous grammar also captures arithmetic precedence in the way the productions use each other, and it captures the usual left-associativity by building sub-exprs to the left of the operator. Note precedence is not property of CFG, it's property of semantics we choose to apply to the strings. If grammar reflects what we want, easier for compiler!

PARSER TYPES

Top-Down (LL) Parsers
— Left to right, Leftmost derivation.
— Starts at S, root of tree, and 'grows' it down.
— Predicts next state with N-lookahead. —
Often use execution stack for PDA.

Bottom-up (LR Parsers)
— Left to Right, Rightmost derivation.
— Starts at the leaves and combines low-level bits into higher.
— Starts with input string, ends with S (ideally).
— Starts in state valid for legal first tokens.
— Changes state to encode possibilities as input is consumed.
— Use explicit stacks PDA, storing state and sentential forms.

FOR TOP DOWN PARSING:

No *left recursion* or *common prefixes*. A grammar is left recursive if there exists A in NonTerms such that $A \rightarrow A\Sigma$ for some string Σ.

– Transform the grammar to remove left recursion:

$A \rightarrow A\Sigma \mid \mu$
$\Rightarrow$
$A \rightarrow \mu B$
$B \rightarrow \Sigma B \mid \epsilon$, where B is new nonterminal.

ELIMINATE COMMON PREFIXES

$$
\begin{aligned}
&A \rightarrow B\delta \\
&\quad \rightarrow B\mu \\
&\Rightarrow \\
&A \rightarrow BBtail \\
&Btail \rightarrow \delta \mid \mu
\end{aligned}
$$

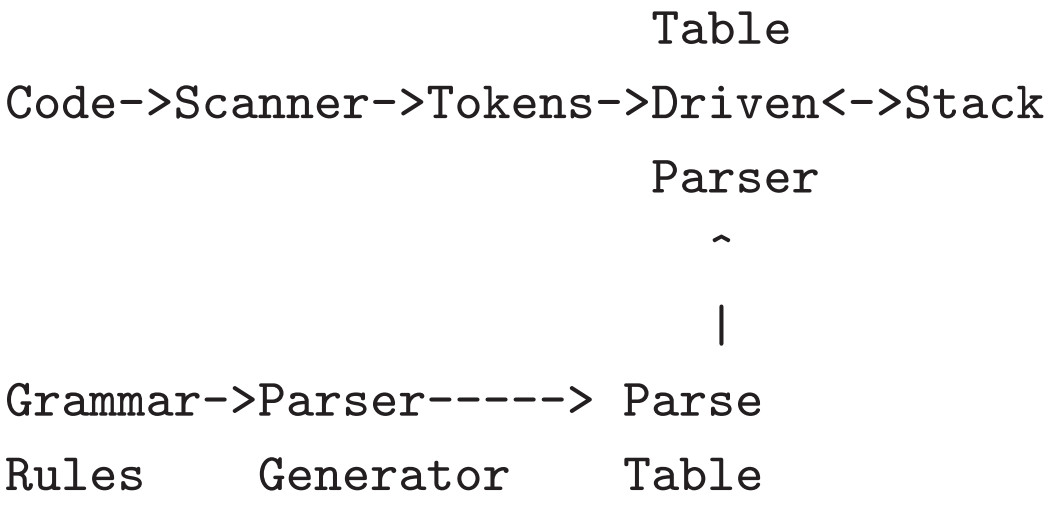
PARSER CONSTRUCTION

*** Recursive descent parsing
– Top-down parsing algorithm
– Built on procedure calls (may be recursive)
– Write procedure for each non-terminal, turning each production into clause
– Insert call to procedure `A()` for non-terminal A and to `match(x)` for terminal x
– Start by invoking procedure for start symbol S

PREDICTIVE (TABLE-DRIVEN) PARSING

** Actions
– Match a terminal
– Predict a production
– Announce a syntax error
* Push as yet unseen portions of productions onto a stack
* Use
– FIRST (A)
– FOLLOW(A)

PREDICTIVE (TABLE-DRIVEN) PARSING



THE FIRST SET

** FIRST(a) is the set of terminal symbols that begin strings derived from a
** To build FIRST(X):
– If X is a terminal, FIRST(X) is X
– If X $\rightarrow \epsilon$, then $\epsilon \in$ FIRST(X)
– If X $\rightarrow$ Y1 Y2 Yk then put FIRST(Y1) in FIRST(X)
– If X is a non-terminal and X $\rightarrow$ Y1 Y2 Yk, then a $\in$ FIRST(X) if a $\in$ FIRST(Yi) and $\epsilon \in$ FIRST(Yj), for all $1 < j < i$.

THE FOLLOW SET

** For a non-terminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A in some sentential form.
** To build FOLLOW(B) for all B –
– Starting with goal, place eof in FOLLOW(¡goal¿)
– If A $\rightarrow$ aBb, then put FIRST(b)-ϵ in FOLLOW(B).
– If A $\rightarrow$ aB, then put FOLLOW(A) in FOLLOW(B) – If A $\rightarrow$ aBb and $\epsilon \in$ FIRST(b), then put FOLLOW(A) in FOLLOW(B).

USING FIRST AND FOLLOW

** For each production A $\rightarrow$ a and lookahead token
– Expand A using the production if token $\in$ FIRST(a)
– If $\epsilon \in$ FIRST(a), expand A using the production if token $\in$ FOLLOW(A)
– All other tokens return error
* If there are multiple choices, the grammar is not LL(1) (predictive).

LL(1) GRAMMARS

A Grammar G is LL(1) if and only if, for all non-terminals A, each distinct pair of productions A $\rightarrow$ a and A $\rightarrow$ b satisfy the condition FIRST(a) $\cap$ FIRST(b) = ϕ, i.e.,
For each set of productions A $\rightarrow$ a1 | a2 mid ... | an,
— FIRST(a1), FIRST(a2), , FIRST(an) are all pairwise disjoint
— If ai $\rightarrow \epsilon$ for any i, then FIRST(aj) $\cap$ FOLLOW(A) = ϕ, for all j $/neq$ i.

THE COMPLEXITY OF LL(1) PARSING

Expect linear since no back-up (look-ahead).

** Instructions inside main loop bounded by constant (function of symbols on RHS)
** How many times does the main loop execute?
– Number of iterations is the number of nodes in the parse tree, which is $\leq$ N*P (N is the number of tokens in the input, P is the max. number of productions on a RHS)
– P is a constant, therefore running time is O(N).

EXAMPLE NON-LL GRAMMAR CONSTRUCT

```
stmt --> if condition then-clause else-clause
| other_stmt

then-clause --> then stmt
else-clause --> else stmt | e
```

`if C1 then if C2 then S1 else S2` — the `else` can be paired with either `then`. Neither LL nor LR. Rather famous problem.

FIX? AN UNNATURAL B-UP (LR) GRAMMAR

```
stmt --> balanced-stmt | unbalanced-stmt
balanced-stmt --> if condition then balanced-stmt
else balanced-stmt | other-stmt

unbalanced_stmt --> if condition then stmt
| if condition then balanced-stmt
else unbalanced-stmt
```

OR: Use special disambiguating rules, like "use production that occurs first in case of conflict".

OR: Use a better syntax!