

- A *grammar* specifies the legal syntax of a language. The kind of grammar most commonly used in computer language processing is a *context-free grammar*.
- A grammar specifies a set of
 - *terminal symbols*: tokens from alphabet of strings;
 - *non-terminal symbols*: phrase names or parts of language;
 - *productions*: each production specifies how a nonterminal symbol may be replaced by a string of terminal or non-terminal symbols;
 - *start symbol*: one of the non-terminals.

Grammar is a four-tuple $G = (T, N, S, P)$ where:

- **T** is the set of *terminal symbols* or *words* of the language.
- **N** is a set of *nonterminal symbols* or *phrase names* that are used in specifying the grammar. We say $V = TN$ is the *vocabulary* of the grammar.
- **S** is a distinguished element of N called the *start symbol*.
- **P** is a set of *productions*, $P \subseteq V^*NV^* \rightarrow V^*$. We write productions in the form $a \rightarrow b$ where a is a string of symbols from V containing at least one nonterminal and b is any string of symbols from V

- **Regular**
 - $A \rightarrow x, A \rightarrow xB,$
 - A, B are N, x is T^*
- **Context-Free**
 - $A \rightarrow a$
 - A is N, a is V^*
- **Context-Sensitive**
 - $a \rightarrow b$
 - a is V^*NV^*, b is V^*

Sample CFGrammar

productions:

$E \rightarrow T$
 $E \rightarrow E + E$
 $E \rightarrow E - E$
 $T \rightarrow \text{var}$
 $T \rightarrow \text{num}$
 $T \rightarrow T * T$
 $T \rightarrow (E)$

terminals:

$\{\text{var}, \text{num}, +, -, (,), *\}$

non-terminals:

$\{E, T\}$

start symbol:

$\{E\}$

productions are recursive and mutually recursive

- derivation
 - start with starting sentence;
 - repeat until no non-terminals remain:
 - chose a non-terminal symbol
 - chose a production to use
 - replace the non-terminal
- leftmost: always chose left-most non-terminal to expand next
- rightmost: always chose right-most non-terminal to expand next

- example: $3+x*(y+7)$
- 3, x, y, 7 – semantic values
- left-hand side is a single N => derivation forms a *parse tree*
- structure of parse tree important!
- parse tree -> abstract syntax tree (AST)

- ambiguity:
 - same sentence with two different parse trees (will have different meaning!)
 - x-y-z
- transform grammar
 - operand precedence, new non-terminals, ... more later
 - e.g.
 - $E \rightarrow E + T \mid E - T \mid T$
 - $T \rightarrow T * F \mid F$
 - $F \rightarrow \text{num} \mid \text{var} \mid (E)$
 - '*' associates to left

- parser must read EOF marker, so augment grammar with
 - $S \rightarrow E \$$
 - $E \rightarrow E + T \mid E - T \mid T$
 - $T \rightarrow T * F \mid F$
 - $F \rightarrow \text{num} \mid \text{var} \mid (E)$

- Parsing:

- takes token as input, and performs syntactical, also semantic analysis

- top-down:

- root->leaves in tree, left->right in rules, abstract to concrete, performs guessing, handwritten or automatically gen.

- bottom-up:

- leaves to root in tree, right->left, concrete to abstract, performs matching, automatically gen.

LL Parsing

- Left-to-right, Leftmost derivation
- top-down, predicative, LL(k) – k tokens lookahead, recursive descent – turn each grammar production into a clause of a recursive function
- start at top – from start rule, and try to expand start symbol by choosing appropriate rule based on input token... continue for all Ns.

```

S -> if E then S else S      L -> end
S -> begin S L              L -> ; S L
S -> print E                E -> num = num

datatype token=IF | THEN | ELSE ...
val tok = ref (getToken())
fun advance() = tok:=getToken()
fun eat(t) = if (!tok=t) then advance() else error()

fun S() = case !tok
  of IF => (eat(IF); E(); eat(THEN); S(); eat(ELSE); S())
  | BEGIN => (eat(BEGIN); S(); L())
  | PRINT => (eat(PRINT); E())
and L() = case !tok
  of END => (eat(END))
  | SEMI => (eat(SEMI); S(); L())
and E() = (eat(NUM); eat(EQ); eat(NUM))

```

```

S -> A C $                  B -> b B
A -> a B C d                | eps
  | B Q                      Q -> q
C -> c                       | eps
  | eps

parseA() =
  if peek() = a then
    (eat(a); parseB(); parseC(); eat(d))
  else
    (parseB(); parseQ();)

? if another rule
A -> a Q a
conflict: cannot predict with only 1 token lookahead

```

- necessary: first terminal symbol of each subexpression to provide enough info to choose production rule.
- FIRST – given a string a from V^* , $FIRST(a)$ is set of all terminals that can begin any string derived from a .
 - $E \rightarrow E + T \mid E - T \mid T$
 - $T \rightarrow T * F \mid F$
 - $F \rightarrow \text{num} \mid \text{var} \mid (E)$
- $FIRST(T*F) = \{\text{num}, \text{var}, (\}$
- $FIRST(AB) = \{a, b, \alpha\}$
- if $X \rightarrow a1$ and $X \rightarrow a2$, and x is in $FIRST(a1)$ and $FIRST(a2) \rightarrow$ conflict
- nullable(X) – true if X can derive the empty string
- FOLLOW(X) – set of terminals that can immediately follow X

FIRST and FOLLOW – empty, nullable – false

for each terminal t
 $FIRST[t] \leftarrow \{t\}$

repeat

- for each production** $X \rightarrow Y_1 \dots Y_k$
 - if** $Y_1 \dots Y_k$ are all nullable (or $k=0$)
 - then** nullable[X]=true
- for each i** from 1 to k , each j , from $i+1$ to k
 - if** $Y_1 \dots Y_{i-1}$ are all nullable (or $i=1$)
 - then** $FIRST[X]$ is $FIRST[X] \cup FIRST[Y_i]$
 - if** $Y_{i+1} \dots Y_k$ are all nullable (or $i=k$)
 - then** FOLLOW[Y_i] is FOLLOW[Y_i] \cup FOLLOW[X]
 - if** $Y_{i+1} \dots Y_{j-1}$ are all nullable (or $i+1=j$)
 - then** FOLLOW[Y_i] is FOLLOW[Y_i] \cup FIRST[Y_j]

until FIRST, FOLLOW, and nullable stop changing

- pre-order, in-order, post-order tree traversals
- idea of top-down vs. bottom-up parsers
- LL(1) top-down, 1-token look-ahead
- we start in starting state, and based on input token need to determine which production rule to choose for the next node in the tree
 - FIRST sets

- $\text{id} + (\text{id} + \text{id})$
- input $\rightarrow E \$$
- $E \rightarrow T R$
- $T \rightarrow \text{id} \mid (E)$
- $R \rightarrow + E \mid \text{eps}$

First Set

- $FIRST(X)$ – for $X = \{\text{terminals, nonterminals, all production alternatives, and alternative tails}\}$;
 $FIRST(\text{eps}) = \{\text{eps}\}$
- init:
 - for $T = \{\text{token}\}$
 - for N , and non-eps alternatives and tails = $\{\}$
 - for eps alternatives and tails = $\{\text{eps}\}$
- for each $N \rightarrow a$, $FIRST(N)$ contains $FIRST(a)$
- for each alternative or tail of alternative a of form Ab , $FIRST(a)$ contains $FIRST(A)$ excluding eps
- if $FIRST(A)$ contains eps, $FIRST(a)$ contains $FIRST(b)$ including eps.

Input	{ id, (}
E \$	{id, (}
\$	{ \$ }
E	{id, (}
T R	{id, (}
R	{+,eps}

T	{id, (}
id	{id}
(E)	{ (}
E)	{id,) }
)	{) }
R	{+, eps}
+ E	{+}
E	{id, (}
eps	{eps}

Follow sets

- if N is nullable, and we see a token t , how do we know which rule to choose?
- need information on Follow sets.
- init all $FOLLOW(N)$ to $\{\}$
- for each $M \rightarrow aNb$, $FOLLOW(N)$ contains $FIRST(b)$ without eps
- if $FIRST(b)$ includes eps, $FOLLOW(N)$ includes $FOLLOW(M)$

non-term	FollowSet
Input	{ }
E	{ \$,) }
T	{ +, \$,) }
R	{ \$,) }

Predictive table

- For each N enter production N->a for all T in FIRST(a).
- If a nullable, enter N->a for all T in FOLLOW (N)

	id	+	()	\$
Input	E		E		
E	TR		TR		
T	id		(E)		
R		+E		eps	eps

- can be modeled with a push-down automaton
- for each input
 - pop N from stack
 - check prediction table
 - if T => match
 - else – push entry from pred. table on stack

Z->d	Y->	X->Y	
X->X Y Z	Y->c	X->a	
	nullable	FIRST	FOLLOW
X	yes	a, c	a, c, d
Y	yes	c	a, c, d
Z	no	a, c, d	(from FOLLOW[X])
Predicate parsing table			
	a	c	d
X	X->a	X->Y	X->Y
	X->Y		
Y	Y->	Y->	Y->
		Y->c	
Z	Z->XYZ	Z->XYZ	Z->d
			Z->XYZ

Conflicts

- cannot determine which rule to take on a character
 - multiple entries in prediction tables – grammar not LL(1) – can have additional lookahead
- FIRST/FIRST conflict – FIRST sets all alternatives for N must be disjunct
- FIRST/FOLLOW conflict – for each N with a nullable alternative – FOLLOW(N) and FIRST(N) must be disjunct
- No multiple nullable alternatives

- left factoring

$T \rightarrow id \mid id [E]$ $T \rightarrow id \text{ After_id}$
 $\text{After_id} \rightarrow [E] \mid \text{eps}$

- substitution

$T \rightarrow id \mid \text{Indexed_id} \mid (E)$
 $\text{Indexed_id} \rightarrow id [E]$

$T \rightarrow id \mid id [E] \mid (E)$
 $T \rightarrow id \text{ After_id} \mid (E)$

- left-recursion removal

- $E \rightarrow E+T$
- $E \rightarrow T$
- $\text{FIRST}(E+T)$ contains $\text{FIRST}(E)$ contains $\text{FIRST}(T)$???
- make it into right recursive production!

- $N \rightarrow Na \mid b \rightarrow$

b, ba, baa, baaa, baaaaaaaaaaaa,

$N \rightarrow bN'$
 $N' \rightarrow aN' \mid \text{eps}$

$E \rightarrow E+T \mid T$ $E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \text{eps}$

Errors in LL(1)

- we want to continue parsing
- delete, replace, insert?
- change prediction rule to match input, or input to match prediction rule?
- must not permit corrupt parse trees to be created!
- skip input until it's safe to continue parsing
 - when is that?

- direct recursion (the obvious)
- indirect: N starts with A, which starts with B, which starts with N (use substitution)
- hidden: N is aN, but a is nullable