Compilers Course
Lecture 4: Context Free Grammars

Example: attempt to define simple arithmetic expressions using named regular expressions:

num = [0-9]+
sum = expr "+" expr
expr = "(" sum ")" | num

Appears to generate "2", "(3+4)", "(3+(4+5))", etc.

However, expr is not a regular language because:

- If we have scanned N "(":s then we must also scan N ")":s, and finite automata cannot do that for arbitrarily large N:s .
- expr is a self-recursive definition, which implies that it is not a regular expression .

We need something stronger: Context-Free Grammars.

Context-Free Grammars (CFGs) :

A CFG G is a 4-tuple (V,T,P,S), where:

- V is a finite set of variable symbols very often the variables are instead called the non-terminals, and the set V is then called N .
- T is a finite set of terminal symbols V and T are disjoint .
- P is a finite set of productions $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$, i.e., α is a string of terminals and variables .
- S is the start symbol, $S \in V$.

Example:
V = { S }
T = { "(", ")" }
P = { $S \rightarrow (S)$, $S \rightarrow \varepsilon$}

This is typically written as
$S \rightarrow (S)$
$S \rightarrow \varepsilon$

or as
$S \rightarrow (S) \mid \varepsilon$

or as
S ::= (S) | ε

with the sets V and T being implicit.

Derivations

A grammar G defines a derivation relation, ⇒, as follows:

$\alpha A \gamma \Rightarrow \alpha \beta \gamma$
if $A \rightarrow \beta \in P$

where A is a variable, and α, β, and γ are strings over $V \cup T$.

The ⇒ relation is then extended to ⇒*:
$\alpha \Rightarrow^* \alpha$
$\alpha \Rightarrow^* \gamma \Rightarrow^*$ if $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$

That is, we can apply ⇒ zero or more times.

Finally, the language of a grammar G, L(G), is defined as:
$L(G) = \{ w \mid S \Rightarrow^* w \wedge w \in T^* \}$
That is, the set of strings of terminals w that can be generated by derivations starting from the start symbol S.

These grammars are called "context-free" because the context surrounding a variable, α and γ above, does not affect the applicability of a production for that variable.

If $S \Rightarrow^* w$ and $w \in T^*$, then w is called a sentence.
If $S \Rightarrow^* w$ and $w \in (V \cup T)^*$, then w is called a sentential form.

Example: G: $S \rightarrow (S) \mid \varepsilon$
$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow \ldots \Rightarrow ({}^i S)^i \Rightarrow ({}^i)^i$
$L(G) = \{ ({}^i)^i \mid i >= 0 \}$

Example: G is
(P0) stmt → id := expr
(P1) | if expr then stmt
(P2) | stmt ; stmt
(P3) expr → id
(P4) | num

Apparently V = { stmt, expr } and T = { id, num, :=, ;, if, then }.
We assume stmt is the start symbol since it is defined first.

Now:
<u>stmt</u> ⇒ (P1)

if <u>expr</u> then stmt ⇒ (P3)

if id then <u>stmt</u> ⇒ (P0)

if id then id := <u>expr</u> ⇒ (P4)

if id then id := num

This sequence is called a derivation. We have just proved that "if id then id := num" ϵ L(G). For clarity we underline the variable that is being rewritten, and indicate the production used after the ⇒ sign.

Left-most and Right-most derivations

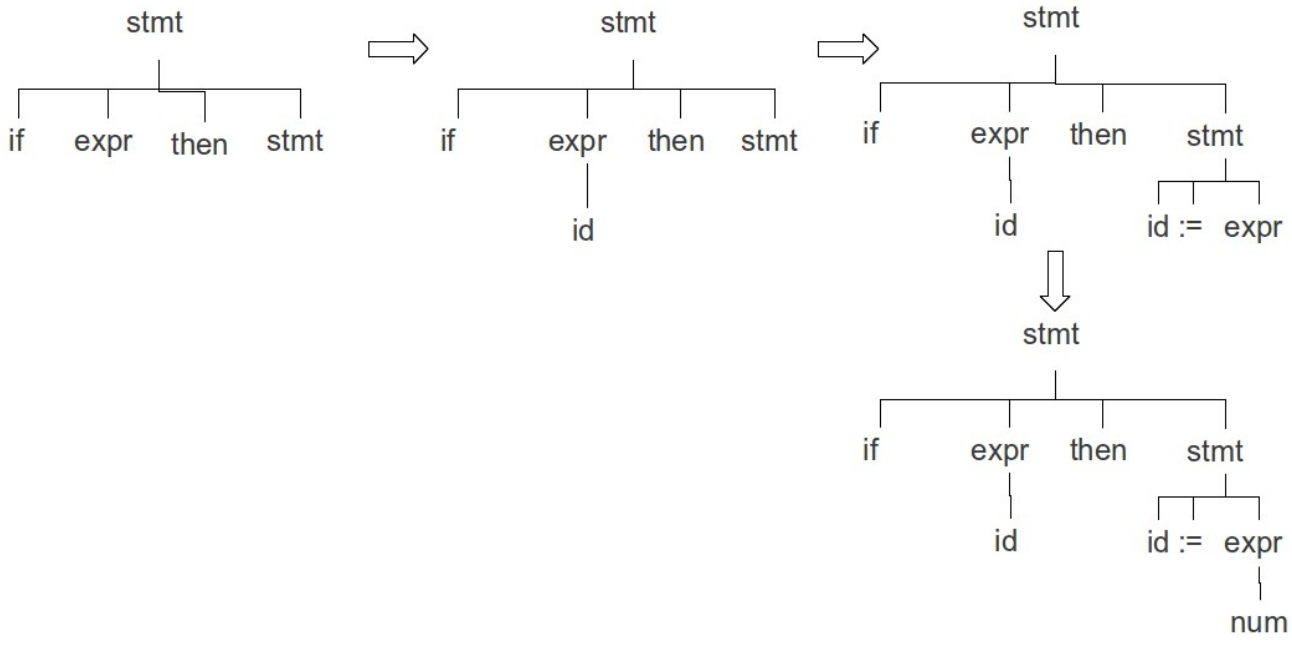
This was a left-most derivation, where in each step the left-most variable is rewritten: $\Rightarrow_{lm}$ There are also right-most derivations, where in each step the right-most variable is rewritten: $\Rightarrow_{rm}$ This doesn't change the generated language, but becomes relevant later when specific parsing algorithms are introduced.

Parsing and Parsers

To "parse" a string of terminals is to construct a derivation from the start symbol to that string. A "parser" is a device that parses strings, or rejects them if they cannot be parsed.

Parse Trees

A yes/no answer is insufficient. A compiler needs to see the structure of the token stream. Parse trees are those structures:



Normally these trees are written with the root at the top and subtrees and leaves further down.

In compilers, *all* further interpretation of the input program is based on its parse tree.

Ambiguity

Consider the grammar E → num | E + E | E * E, and the input "1+2*3".

Derivation 1:
E ⇒ E*E ⇒ (E+E)*E ⇒ (1+E)*E ⇒ (1+2)*E ⇒ (1+2)*3

(Parentheses shown for clarity, they are not actual terminals.)

Derivation 2:
E ⇒ E+E ⇒ 1+E ⇒ 1+(E*E) ⇒ 1+(2*E) ⇒ 1+(2*3)

But this results in two different parse trees, which when interpreted would evaluate to different results (9 vs 7).

Definition:
A grammar is *ambiguous* if its language includes strings for which there exist several different parse trees (derivations).

Example: Expressions

Consider the naive expression grammar

E → E + E | E * E | (E) | num | id

This grammar fails to express precedence (priority) and associativity.

Precedence

Conventionally "*" binds tighter than "+", so E1+E2*E3 should be parsed as E1+(E2*E3), not (E1+E2)*E3.

The way to ensure this is to rewrite the grammar so that E1+E2*E3 *cannot* be parsed as (E1+E2)*E3, which forces the other (correct) derivation.

To prevent the incorrect derivation we introduce new non-terminals, separate "+" expressions from "*" expressions, and allow "+" expressions to have "*" expressions as subexpressions but not vice-versa:

E → E+E | T
T → T*T | F
F → id | num | (E)

The left-hand side of "*" must be a T, but T cannot derive E+E, so E1+E2*E3 can only parse as E1+(E2*E3).

Splitting the productions into several precedence levels, with careful references between the levels, is a common "design pattern" for expressing precedence in expression grammars.

Associativity

The productions "E → E+E | T" do not express the associativity of the "+" operator, so E1+E2+E3 could parse as (E1+E2)+E3 or as E1+(E2+E3). Replace "+" by "-" or "/" and it is obvious that this ambiguity is a problem.

"E → E+E | T" generates sentential forms "T1+T2+...+Tn". We do not want to change that: what we *do* want is to control whether they parse as (((T1+T2)+T3)+...)+Tn or as T1+(T2+(T3+(...+Tn))).

If we want left associativity, then we must prevent derivations where T1+T2+T3 parse as T1+(T2+T3). That is, the right-hand side of a "+" expression must not itself be a "+" expression. Solution:

E → E+T | T

Now E ⇒ E+T ⇒ (E+T)+T ⇒ (T+T)+T, so "+" is left associative.

If we want right associativity, just change the order. For example, to make "E → E=E | T" right associative, write:

E → T=E | T

Then E ⇒ T=E ⇒ T=(T=E) ⇒ T=(T=T),
which is appropriate for e.g. C's assignment operator.

Standard Expression Grammar

Rewrites to express both precedence and associativity are usually required, leading to the following standard expression grammar:

E → E+T | T
T → T*F | F
F → id | num | (E)

More Examples

1. Optional constructs

S → if E then S else S
  | if E then S

At a higher level, we might want to say:

S → if E then S $(\text{else S})_{\text{optional}}$

This can be encoded using an auxiliary non-terminal S':

S → if E then S S'
S' → else S | ε

2. Sequences

Consider function calls: f(), f(x), f(x,y,z) How do we express the shape of the argument list?

E → id "(" Args ")"
Args → ε
  | ArgSeq
ArgSeq → E
  | ArgSeq "," E

The "Dangling Else" Ambiguity

Consider a simple statement grammar:

S → if E then S else S
  | if E then S
  | other

Now consider "if E1 then if E2 then S1 else S2".

This can be parsed as:
S ⇒ if E1 then S
  ⇒ if E1 then (if E2 then S1 else S2)

or as:
S ⇒ if E1 then S else S2
  ⇒ if E1 then (if E2 then S1) else S2

In the first case S2 is executed if E1 is true and E2 is false. In the second case S2 is executed if E1 is false. Clearly this ambiguity is unacceptable.

Traditionally programming languages prefer the first interpretation, which is that an "else" part is associated with the nearest previous incomplete "if" statement.

To achieve this we can rewrite the grammar and introduce non-terminals for "matched if:s" and "unmatched if:s", as follows:

S → MIF | UIF
MIF → if E then MIF else MIF
| other
UIF → if E then S
| if E then MIF else UIF

That is, the statement between "then" and "else" must always be a fully matched statement where each "if" has "else".

Now consider the second possible parse of the IF example above:
if E1 then (if E2 then S1) else S2
The inner (if E2 then S1) is clearly a UIF, so at some point in the derivation there would have to be a sentential form:
if E1 then UIF else S2
However, in the rewritten grammar only MIF can occur between "then" and "else", so this is impossible. Hence we have prevented this undesirable parse of nested IFs.

In practice the "dangling else" ambiguity is often tolerated because most parsing algorithms will produce the correct derivation.