Syntax Analysis

Martin Sulzmann

Syntax Analysis

Objective

Recognize individual tokens as sentences of a language (beyond regular languages).

Example 1 (OK)

Program text `x := x + 2` yields token stream

$$(\textit{ident}\ \text{“x”})\ \textit{assign}\ (\textit{ident}\ \text{“x”})\ \textit{plus}\ (\textit{const}\ 2)$$

Example 2 (Not OK)

Program text `x x := + 2` yields token stream

$$(\textit{ident}\ \text{“x”})\ (\textit{ident}\ \text{“x”})\ \textit{assign}\ \textit{plus}\ (\textit{const}\ 2)$$

We will discuss later if the 'OK' example shall be semantically valid.

Approach: Parsing with CFG

Context-Free Grammar (CFG)

Language described in terms of a context-free grammar:

$$\begin{array}{lcl} \mathit{Command} & \rightarrow & \mathit{ident}\ \mathit{assign}\ \mathit{Expression} \mid \ldots \\ \mathit{Expression} & \rightarrow & \mathit{const} \mid \mathit{ident} \mid \mathit{Expression}\ \mathit{plus}\ \mathit{Expression} \end{array}$$

ident, *const*, *plus* being tokens where attributes are omitted.

Note: CFG uses uppercase for non-terminal and lowercase for terminal symbols. Our parsing tool OCamlyacc just uses it the other way around!

Parsing yields Abstract Syntax Tree (AST)

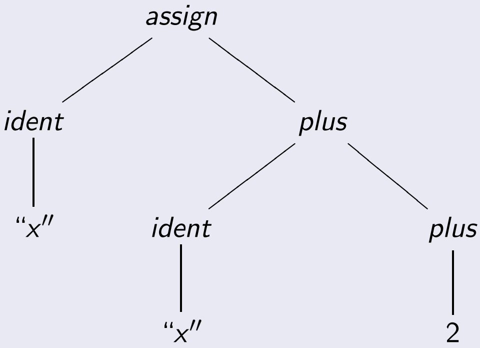
Check if stream of tokens is *valid* and produce AST.

Parsing with CFG Example

Token Stream

(*ident* “x”) *assign* (*ident* “x”) *plus* (*const* 2)

AST



Types of Parsing

Top-Down

- Recursive descent
- From root to leaves
- "ANTRL" style

Bottom-Up

- From leaves to root
- "yacc" style

Others

- Parser combinators
- Derivative-based parser
- ...

Error Handling

Types of Errors

lexical	misspelled keyword	`whle cond ...`
syntactic	unbalanced parentheses	`x:= x+1)`
semantic	incompatible types	`x:= x + True`
logical	infinite loop	`while True  ...`

Provide useful feedback to user. Mostly ignored here.

Context Free Grammar (CFG)

Definition

A CFG G is a 4-tuple (V_t, V_n, S, P) where

1. V_t is the set of *terminal* symbols (commonly the set of tokens).
2. V_n is the set of *non-terminal symbols*.
3. S is a distinguished non-terminal symbol call the *start* symbol.
4. P is a finite set of *productions* (aka rewrite rules) of the form: non-terminal $\rightarrow$ (non-terminal $\mid$ terminal)$*$.

Note:

- All sets are finite.
- Left-hand side (lhs) of production consists of a single non-terminal symbol.
- Right-hand side (rhs) consists of arbitrary combinations of terminal and non-terminal symbol described by a regular expression.

CFG Example and Notation

Example

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid ident
\end{array}
$$

Notation

- Vocabulary $V = V_t \cup V_n$
- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $\alpha, \beta, \gamma, \ldots \in V^*$
- $u, v, w, \ldots \in V_t^*$

CFG as a Rewrite System

Derivations by Rewriting

If $A \to \gamma \in P$ then $\alpha A \beta \Rightarrow \alpha \gamma \beta$ is a *one-step derivation* using $A \to \gamma$.

- We assume that $\Rightarrow^*$ denotes a derivation of ≥ 0 steps.
- In a *leftmost* derivation (denoted $\Rightarrow^*_L$) we replace the leftmost non-terminal in each step.
- In a *rightmost* derivation (denoted $\Rightarrow^*_R$) we replace the rightmost non-terminal in each step.
- If $S \to^* \beta$ then β is a *sentential form* of G.
- If $S \to^* w$ and $w \in V_t^*$ then w is a *sentence* of G.
- $L(G) = \{w \mid S \Rightarrow^* w, w \in V_t^*\}$ denotes the language described by G.

Example Derivation

CFG

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid ident \end{array}$$

Derivation $E \Rightarrow^* ident + (ident)$

$$\begin{array}{lclcl} E & \Rightarrow & E + T & \Rightarrow & T + T \\ & \Rightarrow & F + T & \Rightarrow & ident + T \\ & \Rightarrow & ident + F & \Rightarrow & ident + (E) \\ & \Rightarrow & ident + (T) & \Rightarrow & ident + (F) \\ & \Rightarrow & ident + (ident) & & \end{array}$$

Example Derivation (2)

CFG

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid \ldots \mid 9$$

Derivation $E \Rightarrow^* 3 + 7 * 2$

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E * E \Rightarrow 3 + 7 * E \Rightarrow 3 + 7 * 2$

Things to think about

- Multiple derivations for the same input word?
- How to represent derivations?

Parse Trees

Purpose

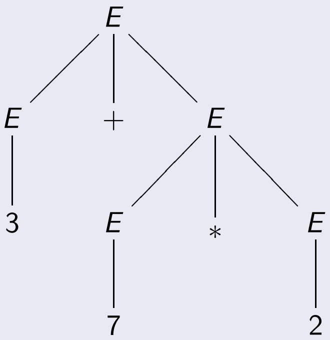
- Compact representation of a derivation $E \Rightarrow^* w$.
- Most often in terms of a tree-like structure (but other formats like bit-encodings are also possible).
- Different from AST!
  - Parse tree represents *concrete* input word.
  - AST is an *abstract* representation of input.

Parse Tree Example

CFG + Derivation

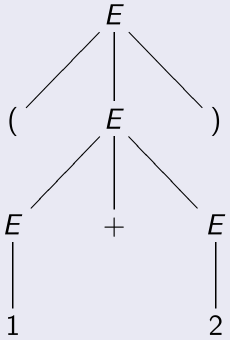
$$E \;\rightarrow\; E + E \mid E * E \mid (E) \mid 0 \mid \ldots \mid 9$$
$$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E * E \Rightarrow 3 + 7 * E \Rightarrow 3 + 7 * 2$$

Parse Tree

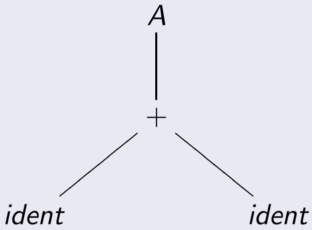


AST versus Parse Tree

Parse tree



AST



Parse tree: Maintain all source (concrete) syntax info.

AST: Abstract representation.

Ambiguity

Definition

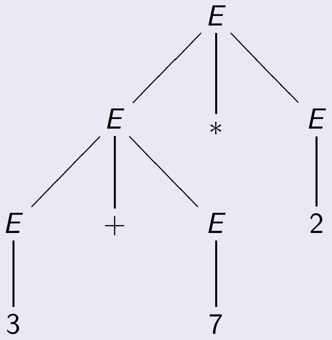
Multiple parse trees for the same grammar and input word.

Example

$$E \;\rightarrow\; E + E \mid E * E \mid (E) \mid 0 \mid \ldots \mid 9$$

$$\begin{aligned} E &\Rightarrow E * E \Rightarrow E + E * E \\ &\Rightarrow 3 + E * E \Rightarrow 3 + 7 * E \\ &\Rightarrow 3 + 7 * 2 \end{aligned}$$

Another parse tree:



Ambiguity Example

Ambiguous Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid ident$$

Equivalent Non-Ambiguous Grammar

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid ident \end{array} \qquad \begin{array}{lcl} E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & (E) \mid ident \end{array}$$

Fact

Some context-free languages are *inherently ambiguous*.

Ambiguity Example (2)

Exercise

Consider

$$\begin{array}{lcl} \mathit{Stmt} & \rightarrow & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmt} \\ & \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \\ & \mid & \texttt{other} \end{array}$$

Ambiguous? Does an equivalent unambiguous grammar exist?

Parsing Approaches: Top-Down versus Bottom-Up

Example

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid \ldots \mid 9$$

Top-Down ("ANTLR")

- Build left-most derivation (strictly reduce left-most non-terminal symbols).
- $E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E * E \Rightarrow 3 + 7 * E \Rightarrow 3 + 7 * 2$
- "Guess which rule to apply from left to right."

Bottom-Up ("yacc")

- Build right-most derivation.
- $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * 2 \Rightarrow E + 7 * 2 \Rightarrow 3 + 7 * 2$
- "Seek for matches of right-hand side, replace by left-hand side."

Top-Down Parsing

Example

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathit{ident}
\end{array}
$$

Approach

Build left-most derivation

- starting from start symbol,
- use target string to choose productions.

Challenge: Left Recursion

- $E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow \ldots$
- Several possible right-hand sides.
- Some lead to dead ends. Backtracking? Too inefficient.

Removal of Left Recursion

Example

Consider

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid ident \end{array}$$

Remove left recursion, e.g.

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \end{array}$$

Removal Left Recursion (2)

Direct Left Recursion Removal

Left recursion can be eliminated as follows:

- Grammar rules

$$A \rightarrow A\alpha_1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m$$

- are replaced by the equivalent rules

$$\begin{array}{lll} A & \rightarrow & \beta_1 A' \mid ... \mid \beta_m A' \\ A' & \rightarrow & \alpha_1 A' \mid ... \mid \alpha_n A' \mid \epsilon \end{array}$$

Indirect Left Recursion Removal

Consider

$$A \rightarrow BC$$
$$B \rightarrow A\beta$$

- Left recursion involves several rules.
- Can be eliminated (reduced to direct case) via unfolding.

Left Factoring

Issue

Remove common prefix of right hand sides.

Left Factoring

Left factoring of $A \to \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$ (where $\alpha \neq \epsilon$ is the longest common prefix) yields

$$\begin{array}{lcl} A & \to & \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m \\ A' & \to & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

Example

$$\begin{array}{lcl} S & \to & iEtS \mid iEtSeS \mid a \\ E & \to & b \end{array} \quad \text{transformed to} \quad \begin{array}{lcl} S & \to & iEtSS' \mid a \\ S' & \to & \epsilon \mid eS \\ E & \to & b \end{array}$$

Top-Down Parsing: Short Summary

Predictive Top-Down Parsing

- Eliminate left recursion.
- Apply left factoring.
- Deterministic. No backtracking.

Lookahead?

- How far to lookahead in the input string to (deterministically) apply a rule?
- Observe right-hand sides!

First (matching symbols)

Intuition

Which terminals can start a string matching the non-terminal?

Definition

$First(\alpha) = \{a \in V_t \cup \{\epsilon\} \mid \alpha \Rightarrow^* a\beta\}$

Inductive Definition

$$First(\alpha) \quad = \begin{cases} \{x\} & \text{if } \alpha = x\alpha',\ x \in V_t \\ First(x) & \text{if } \alpha = x\alpha',\ \epsilon \notin First(x),\ x \in V_n \\ First(x) \cup First(\alpha') & \text{if } \alpha = x\alpha',\ \epsilon \in First(x),\ x \in V_n \end{cases}$$

Sufficient to consider $First(X)$ where X is a non-terminal.

First Algorithm

Algorithm

Build $First(X)$ as follows:

1. If X is a terminal then $First(X) = \{X\}$.
2. If $X \rightarrow \epsilon$ then add ϵ to $First(X)$.
3. If $X \rightarrow Y_1 ... Y_k$:
   1. Put $First(Y_1) - \{\epsilon\}$ in $First(X)$.
   2. $\forall i.1 < i <= k$, if $\epsilon \in First(Y_1) \cap ... \cap First(Y_{i-1})$ (i.e. $Y_1 ... Y_{i-1} \Rightarrow^* \epsilon$) then put $First(Y_i) - \{\epsilon\}$ in $First(X)$.
   3. If $\epsilon \in First(Y_1) \cap ... \cap First(Y_k)$ then add ϵ to $First(X)$.

   Repeat until no more additions can be made.

First Example

Example

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \\
E' & \rightarrow & \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \\
T' & \rightarrow & \epsilon \\
F & \rightarrow & ident \\
F & \rightarrow & (E)
\end{array}
\qquad
\begin{array}{l}
First(E) = \{ident, (\} \\
First(E') = \{+, \epsilon\} \\
\\
First(T) = \{ident, (\} \\
First(T') = \{*, \epsilon\} \\
\\
First(F) = \{ident, (\} \\
\\
\end{array}
$$

Follow

Issue

- Which terminals can start a string matching *after* the nonterminal?
- Consider ϵ right hand sides.

Definition

$Follow(A) = \{w \in First(\beta) \mid S \Rightarrow^* wA\beta\}$

Algorithm

Build $Follow(A)$ as follows:

1. Add \$ to $Follow(S)$ where S is the start symbol.
2. If $A \to \alpha B \beta$:
   1. Put $First(\beta) - \{\epsilon\}$ in $Follow(B)$
   2. If $\beta = \epsilon$ (i.e. $A \to \alpha B$) or $\epsilon \in First(\beta)$ (i.e. $\beta \Rightarrow^* \epsilon$) then put $Follow(A)$ in $Follow(B)$.

Repeat until no more additions can be made (\$ $=$ EOF token).

First and Follow Example

Example

$$
\begin{array}{lcl}
S & \rightarrow & BAa \\
A & \rightarrow & \epsilon \mid BcA \\
B & \rightarrow & \epsilon \mid b
\end{array}
$$

	First	*Follow*
S	a, b, c	\$
A	ϵ, b, c	a
B	ϵ, b	a, b, c

LL(1) Grammar

definition

A grammar G is LL(1) iff for each set of productions $A \to \alpha_1 \mid ... \mid \alpha_n$:

1. $\mathit{First}(\alpha_1), ..., \mathit{First}(\alpha_n)$ are all pairwise disjoint.
2. If $\alpha_i \Rightarrow^* \epsilon$ then $\mathit{First}(\alpha_j) \cap \mathit{Follow}(A) = \emptyset$ forall $1 <= j <= n, i \neq j$.

If G is ϵ-free (there are no ϵ productions), first condition is sufficient.

Facts

1. No left-recursive grammar is LL(1).
2. No ambiguous grammar is LL(1).
3. Some languages have no LL(1) grammar.

LL(1) Grammar (cont'd)

Example

$S \rightarrow aS \mid a$ is not LL(1) because $First(aS) = First(a) = \{a\}$.
However, the following equivalent grammar is LL(1).

$$\begin{array}{lcl} S & \rightarrow & aS' \\ S' & \rightarrow & aS' \mid \epsilon \end{array}$$

Lemma

A grammar is LL(1) iff for each two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ we have that $Look(A \rightarrow \alpha) \cap Look(A \rightarrow \beta) = \emptyset$ where

$$Look(A \rightarrow \alpha) = \begin{cases} First(\alpha) \backslash \{\epsilon\} & \text{if } \epsilon \notin Follow(\alpha) \\ (First(\alpha) \backslash \{\epsilon\}) \cup Follow(A) & \text{otherwise} \end{cases}$$

Recursive Descent Parsing

Example

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \\
E' & \rightarrow & \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \\
T' & \rightarrow & \epsilon \\
F & \rightarrow & ident \\
F & \rightarrow & (E)
\end{array}
\qquad
\begin{array}{l}
First(E) = \{ident, (\} \\
First(E') = \{+, \epsilon\} \\
\\
First(T) = \{ident, (\} \\
First(T') = \{*, \epsilon\} \\
\\
First(F) = \{ident, (\} \\
\\
\end{array}
$$

Recursive Descent Parser

Grammar is LL(1). Introduce a function for each non-terminal symbol.

Recursive Descent Parser

Parser

Ignore the parse tree, so strictly speaking we only build a matcher.

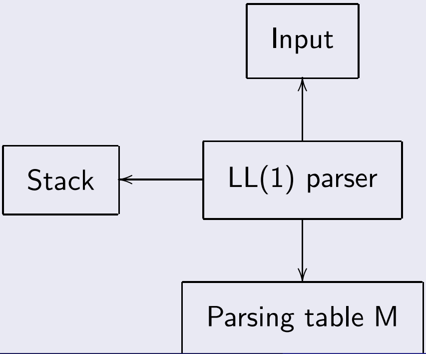
```
e()  { t() ; e'(); if token != EOF then HALT; }
e'() { if token == PLUS
        then get_next_token(); t(); e'(); }
t()  { f(); t'(); }
t'() { if token == TIMES
        then get_next_token(); f(); t'(); }
f () { if token == IDENT
        then get_next_token();
        else if token == OPENPAR
              then get_next_token(); e();
                   if token==CLOSEPAR
                    then get_next_token();
                    else HALT; }
```

Table Driven Parsing

Fact

A program making use of recursive function can be simulated by a non-recursive program (by making use of a stack).

Table Driven Approach



Stack elements are elements in our vo
Input consists of terminal symbols
Table encodes “LL(1) actions”

Parse Table Construction

Algorithm

Input: Grammar G

Output: Parsing table M

Method:

1. For each production $A \rightarrow \alpha$:
   1. For each $a \in \mathit{First}(\alpha)$ add $A \rightarrow \alpha$ to $M[A, a]$.
   2. If $\epsilon \in \mathit{First}(\alpha)$:
      1. For each $b \in \mathit{Follow}(A)$ add $A \rightarrow \alpha$ to $M[A, b]$.
      2. If $\$ \in \mathit{Follow}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.
2. Set each undefined entry of M to *error* (or leave empty).

If there are multiple entries then grammar is not LL(1).

Table Driven Parsing

Algorithm

```
Push $; Push start symbol
while Top of stack is not $ {
  X:= top of stack; a:= input symbol
  if X is a terminal
   then if X == a
         then pop X; advance input
         else error
   else if M[X,a] == X->Y1 ... Yk
         then pop X; push Yk, ..., Y1
         else error
}
```

Example

First and Follow Sets

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \\
E' & \rightarrow & \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \\
T' & \rightarrow & \epsilon \\
F & \rightarrow & ident \\
F & \rightarrow & (E)
\end{array}
$$

$$
\begin{array}{ll}
First(E) = \{ident, (\} & Follow(E) = \{\$,)\} \\
First(E') = \{+, \epsilon\} & Follow(E') = \{\$,)\} \\
 & \\
First(T) = \{ident, (\} & Follow(T) = \{\$, +,)\} \\
First(T') = \{*, \epsilon\} & Follow(T') = \{\$, +,)\} \\
 & \\
First(F) = \{ident, (\} & Follow(F) = \{\$, +, *,)\}
\end{array}
$$

Parse Table + Execution

	ident	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to ident$			$F \to (E)$		

Stack	Input	Rule
$\$E$	$id + (id)$	$E \to TE'$
$\$E'T$	$id + (id)$	$T \to FT'$
$\$E'T'F$	$id + (id)$	$F \to id$
$\$E'T'id$	$id + (id)$	
$\$E'T'$	$+(id)$	$T' \to \epsilon$
$\$E'$	$+(id)$	$E' \to +TE'$
$\$E'T+$	$+(id)$	
$\$E'T$	(id)	$T \to FT'$
$\$E'T'F$	(id)	$F \to (E)$
$\$E'T')E($	(id)	

Stack	Input	Rule
$\$E'T')E$	$id)$	$E \to TE'$
$\$E'T')E'T$	$id)$	$T \to FT'$
$\$E'T')E'T'F$	$id)$	$F \to id$
$\$E'T')E'T'id$	$id)$	
$\$E'T')E'T'$	$)$	$T' \to \epsilon$
$\$E'T')E'$	$)$	$E' \to \epsilon$
$\$E'T')$	$)$	
$\$E'T'$	\$	$T' \to \epsilon$
$\$E'$	\$	$E' \to \epsilon$
\$	\$	

Summary

- Predictive parsing (removal left recursion, left factoring).
- LL(1) grammars.
- Recursive descent parsing.
- Table driven parsing.
- How to construct parse tree? (later)
- Error recovery important (but no time!), please consult text book.
- ANTRL state-of-the art LL-style parser for Java.