SEMANTIC ANALYSIS

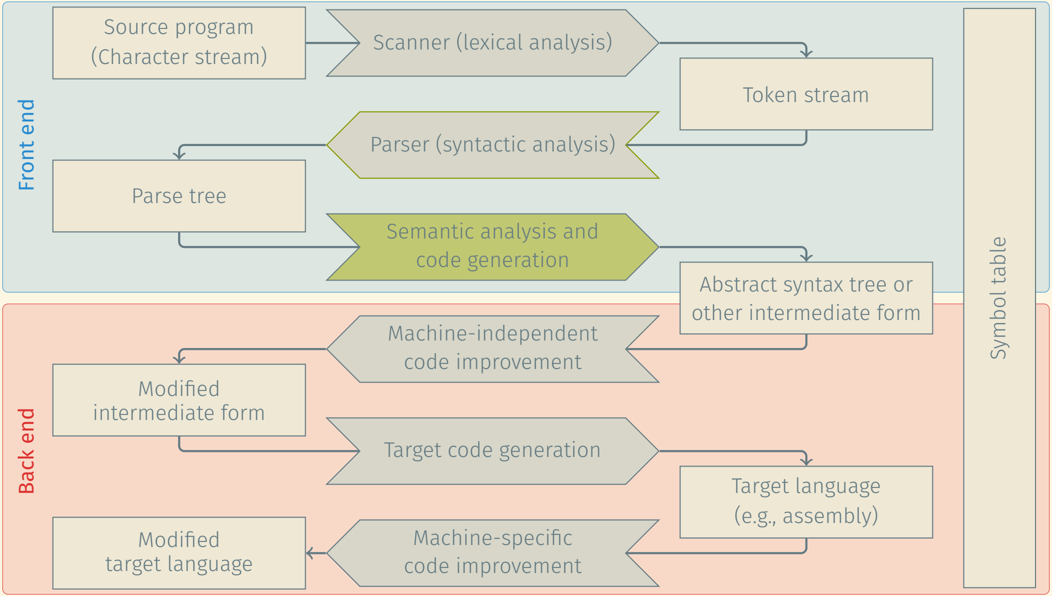
PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

PROGRAM TRANSLATION FLOW CHART



ROAD MAP

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

ROAD MAP

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

Syntax

- Describes form of a valid program
- Can be described by a context-free grammar

SYNTAX AND SEMANTICS

Syntax

- Describes form of a valid program
- Can be described by a context-free grammar

Semantics

- Describes meaning of a program
- Cannot be described by a context-free grammar

Syntax

- Describes form of a valid program
- Can be described by a context-free grammar

Semantics

- Describes meaning of a program
- Cannot be described by a context-free grammar

Some constraints that may appear syntactic are enforced by semantic analysis!

Example: Use of identifier only after its declaration

Semantic analysis

- Enforces semantic rules
- Builds intermediate representation (e.g., abstract syntax tree)
- Fills symbol table
- Passes results to intermediate code generator

SEMANTIC ANALYSIS

Semantic analysis

- Enforces semantic rules
- Builds intermediate representation (e.g., abstract syntax tree)
- Fills symbol table
- Passes results to intermediate code generator

Two approaches

- Interleaved with syntactic analysis
- As a separate phase

Semantic analysis

- Enforces semantic rules
- Builds intermediate representation (e.g., abstract syntax tree)
- Fills symbol table
- Passes results to intermediate code generator

Two approaches

- Interleaved with syntactic analysis
- As a separate phase

Formal mechanism

- Attribute grammars

ENFORCING SEMANTIC RULES

Static semantic rules

- Enforced by compiler at compile time
- **Example:** Do not use undeclared variable.

ENFORCING SEMANTIC RULES

Static semantic rules

- Enforced by compiler at compile time
- **Example:** Do not use undeclared variable.

Dynamic semantic rules

- Compiler generates code for enforcement at runtime.
- **Examples:** Division by zero, array index out of bounds
- Some compilers allow these checks to be disabled.

ROAD MAP

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

ROAD MAP

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

Attribute grammar

An augmented context-free grammar:

- Each symbol in a production has a number of attributes.
- Each production is augmented with semantic rules that
  - Copy attribute values between symbols,
  - Evaluate attribute values using semantic functions,
  - Enforce constraints on attribute values.

ATTRIBUTE GRAMMAR: EXAMPLE

$E \to E + T$

$E \to E - T$

$E \to T$

$T \to T * F$

$T \to T \,/\, F$

$T \to F$

$F \to -F$

$F \to (\,E\,)$

$F \to \text{const}$

ATTRIBUTE GRAMMAR: EXAMPLE

$E \to E + T$	$E_1 \to E_2 + T$	$\{\, E_1.val = \mathrm{add}(E_2.val, T.val) \,\}$
$E \to E - T$	$E_1 \to E_2 - T$	$\{\, E_1.val = \mathrm{sub}(E_2.val, T.val) \,\}$
$E \to T$	$E \to T$	$\{\, E.val = T.val \,\}$
$T \to T * F$	$T_1 \to T_2 * F$	$\{\, T_1.val = \mathrm{mul}(T_2.val, F.val) \,\}$
$T \to T \,/\, F$	$T_1 \to T_2 \,/\, F$	$\{\, T_1.val = \mathrm{div}(T_2.val, F.val) \,\}$
$T \to F$	$T \to F$	$\{\, T.val = F.val \,\}$
$F \to -F$	$F_1 \to -F_2$	$\{\, F_1.val = \mathrm{neg}(F_2.val) \,\}$
$F \to (E)$	$F \to (E)$	$\{\, F.val = E.val \,\}$
$F \to \mathrm{const}$	$F \to \mathrm{const}$	$\{\, F.val = \mathrm{const}.val \,\}$

Synthesized attributes

Attributes of LHS of production are computed from attributes of RHS of production.

Synthesized attributes

Attributes of LHS of production are computed from attributes of RHS of production.

Inherited attributes

Attributes flow from left to right:

- From LHS to RHS,
- From symbols on RHS to symbols later on the RHS.

SYNTHESIZED ATTRIBUTES: EXAMPLE

The language

$$\mathcal{L} = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n > 0\} = \{\mathrm{abc}, \mathrm{aabbcc}, \mathrm{aaabbbccc}, \ldots\}$$

is not context-free but can be defined using an attribute grammar:

The language

$$\mathcal{L} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n > 0\} = \{\mathtt{abc}, \mathtt{aabbcc}, \mathtt{aaabbbccc}, \ldots\}$$

is not context-free but can be defined using an attribute grammar:

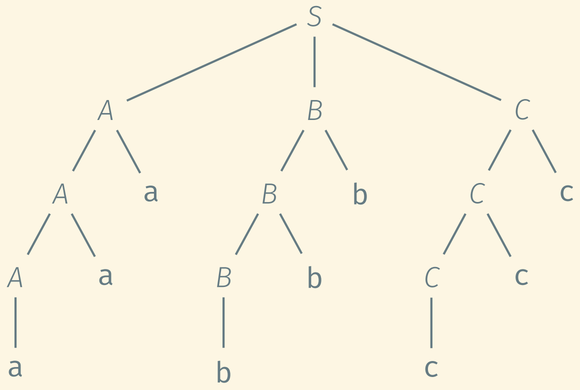
$$
\begin{array}{ll}
S \rightarrow A\;B\;C & \{\text{Condition: } A.count = B.count = C.count\} \\
A \rightarrow \mathtt{a} & \{A.count = 1\} \\
A_1 \rightarrow A_2\;\mathtt{a} & \{A_1.count = A_2.count + 1\} \\
B \rightarrow \mathtt{b} & \{B.count = 1\} \\
B_1 \rightarrow B_2\;\mathtt{b} & \{B_1.count = B_2.count + 1\} \\
C \rightarrow \mathtt{c} & \{C.count = 1\} \\
C_1 \rightarrow C_2\;\mathtt{c} & \{C_1.count = C_2.count + 1\}
\end{array}
$$

SYNTHESIZED ATTRIBUTES: PARSE TREE DECORATION (1)

Input: aaabbbccc

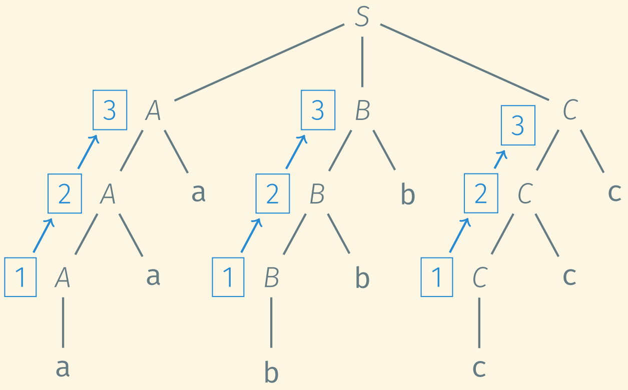
Input: aaabbbccc

Parse tree:



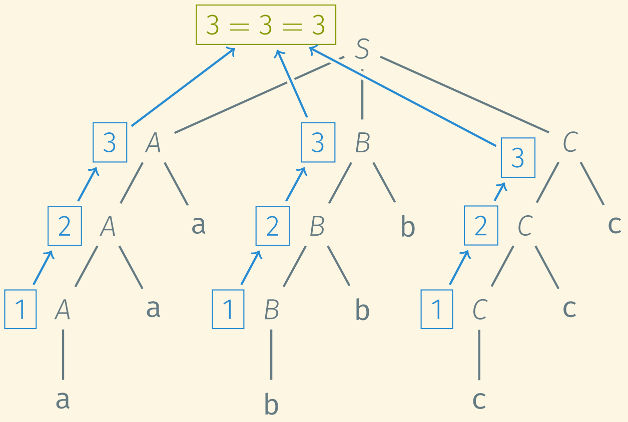
Input: aaabbbccc

Parse tree:



Input: aaabbbccc

Parse tree:

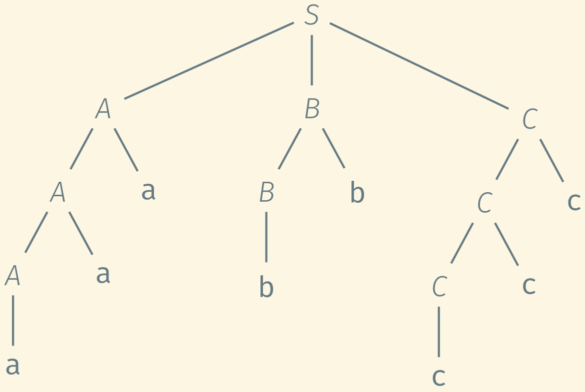


SYNTHESIZED ATTRIBUTES: PARSE TREE DECORATION (2)

Input: aaabbccc

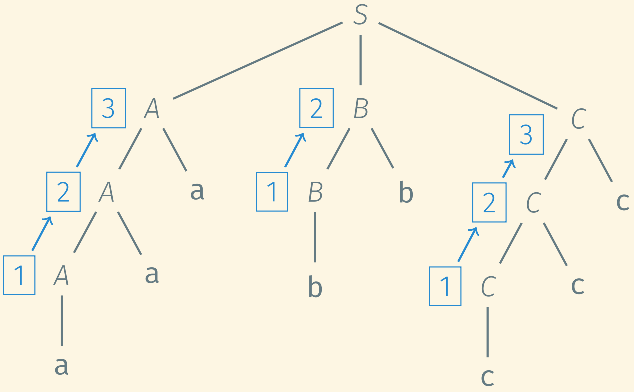
Input: aaabbccc

Parse tree:



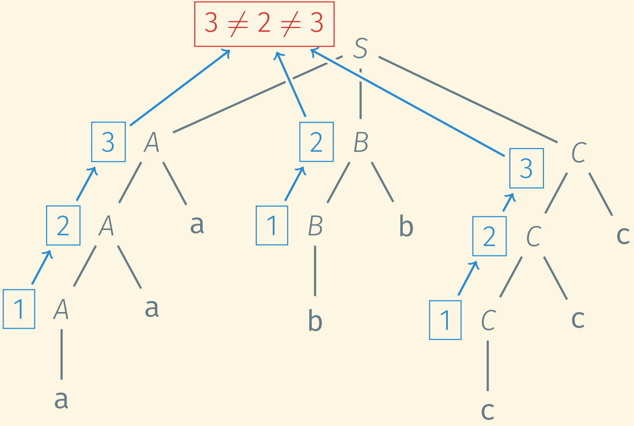
Input: aaabbccc

Parse tree:



Input: aaabbccc

Parse tree:



INHERITED ATTRIBUTES: EXAMPLE

Again, we consider the language

$$\mathcal{L} = \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n \mid n > 0\} = \{\mathsf{abc}, \mathsf{aabbcc}, \mathsf{aaabbbccc}, \ldots\}.$$

Again, we consider the language

$$\mathcal{L} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n > 0\} = \{\mathtt{abc}, \mathtt{aabbcc}, \mathtt{aaabbbccc}, \ldots\}.$$

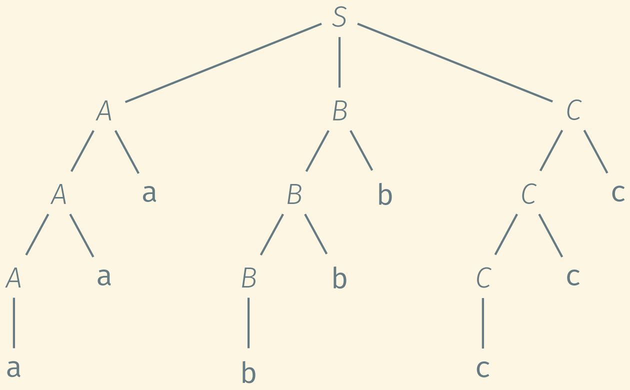
$$
\begin{aligned}
S &\to A\, B\, C && \{B.inhCount = A.count; C.inhCount = A.count\} \\
A &\to \mathtt{a} && \{A.count = 1\} \\
A_1 &\to A_2\, \mathtt{a} && \{A_1.count = A_2.count + 1\} \\
B &\to \mathtt{b} && \{Condition : B.inhCount = 1\} \\
B_1 &\to B_2\, \mathtt{b} && \{B_2.inhCount = B_1.inhCount - 1\} \\
C &\to \mathtt{c} && \{Condition : C.inhCount = 1\} \\
C_1 &\to C_2\, \mathtt{c} && \{C_2.inhCount = C_1.inhCount - 1\}
\end{aligned}
$$

INHERITED ATTRIBUTES: PARSE TREE DECORATION (1)

Input: aaabbbccc

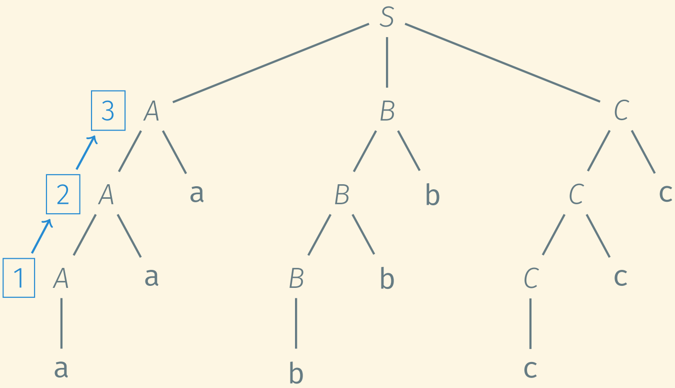
Input: aaabbbccc

Parse tree:



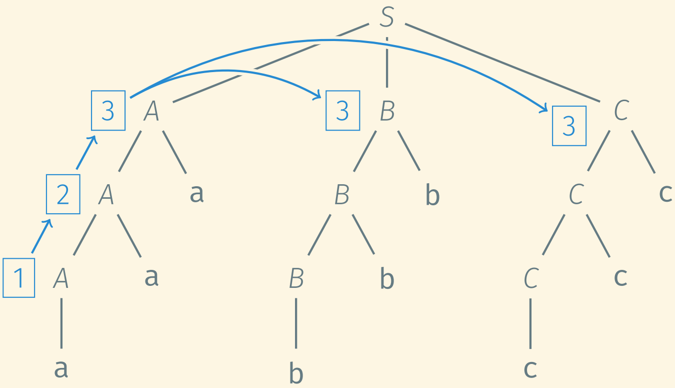
Input: aaabbbccc

Parse tree:



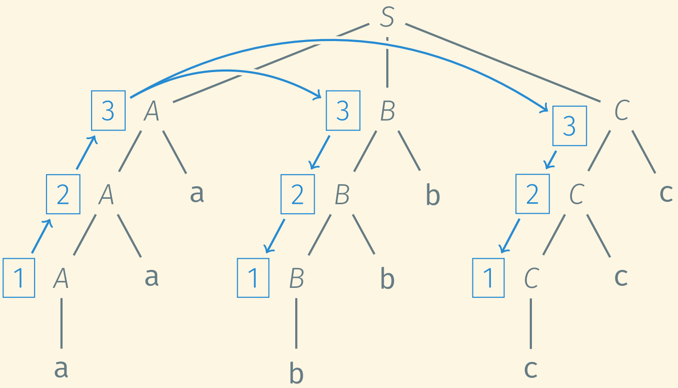
Input: aaabbbccc

Parse tree:



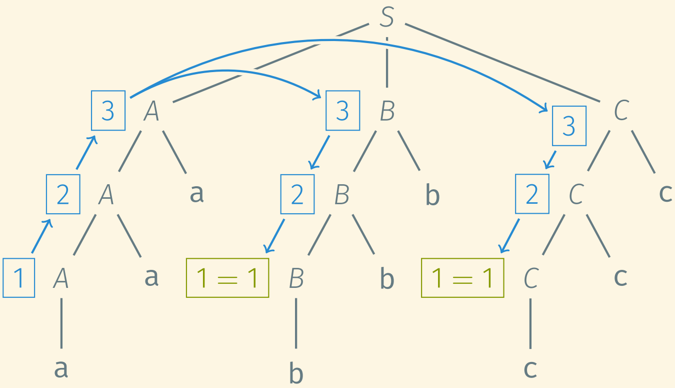
Input: aaabbbccc

Parse tree:



Input: aaabbbccc

Parse tree:



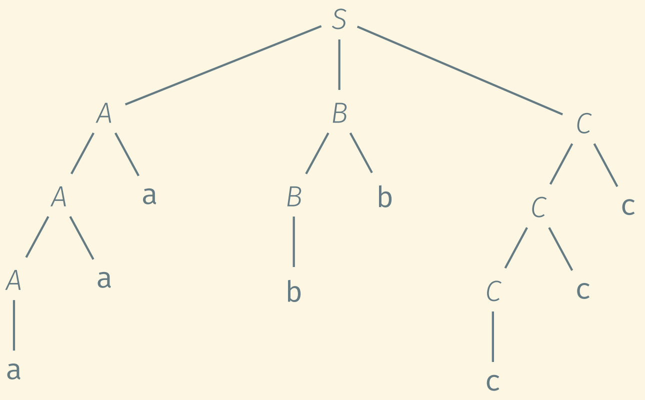
INHERITED ATTRIBUTES: PARSE TREE DECORATION (2)

Input: aaabbccc

INHERITED ATTRIBUTES: PARSE TREE DECORATION (2)

Input: aaabbccc

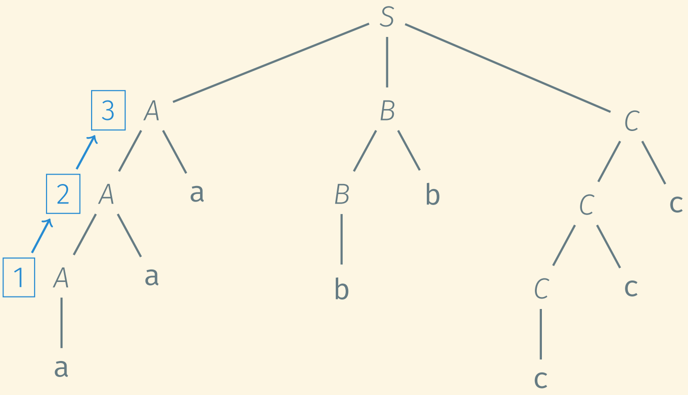
Parse tree:



INHERITED ATTRIBUTES: PARSE TREE DECORATION (2)

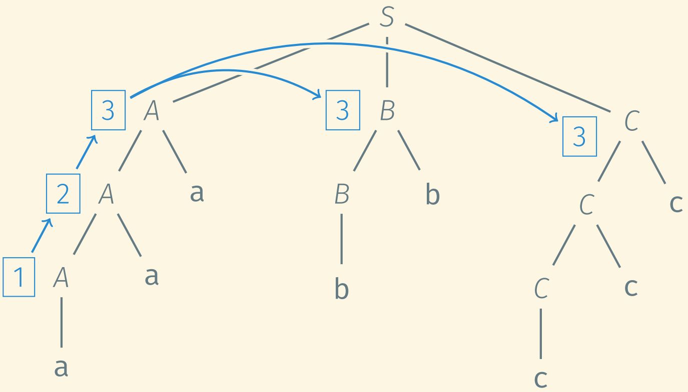
Input: aaabbccc

Parse tree:



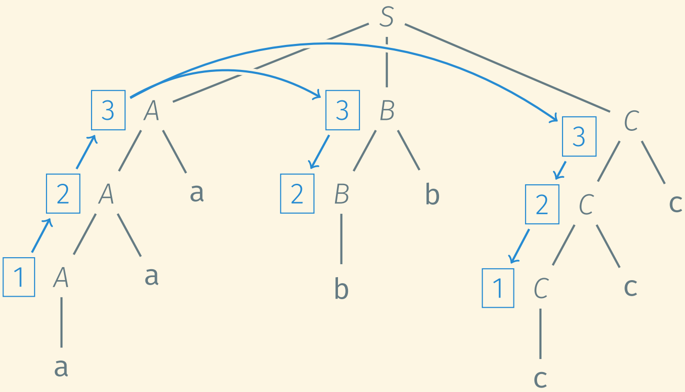
Input: aaabbccc

Parse tree:



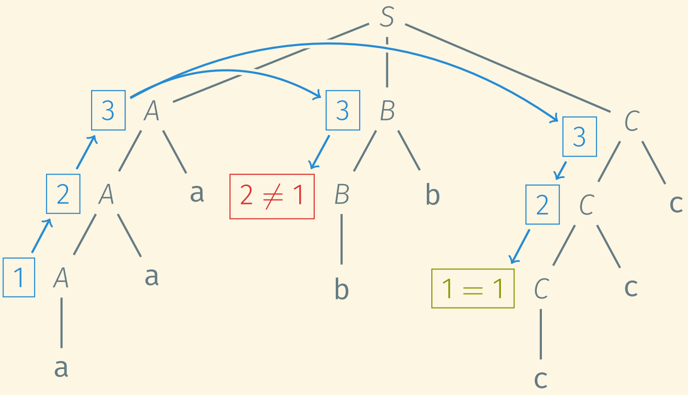
Input: aaabbccc

Parse tree:



Input: aaabbccc

Parse tree:



Annotation or decoration of the parse tree:

- Process of evaluating attributes

ATTRIBUTE FLOW

Annotation or decoration of the parse tree:

- Process of evaluating attributes

Synthesized attributes:

- Attributes of LHS of each production are computed from attributes of symbols on the RHS of the production.
- Attributes flow bottom-up in the parse tree.

ATTRIBUTE FLOW

Annotation or decoration of the parse tree:

- Process of evaluating attributes

Synthesized attributes:

- Attributes of LHS of each production are computed from attributes of symbols on the RHS of the production.
- Attributes flow bottom-up in the parse tree.

Inherited attributes:

- Attributes for symbols in the RHS of each production are computed from attributes of symbols to their left in the production.
- Attributes flow top-down in the parse tree.

S-attributed grammar

- All attributes are synthesized.
- Attributes flow bottom-up.

S-ATTRIBUTED AND L-ATTRIBUTED GRAMMARS

S-attributed grammar

- All attributes are synthesized.
- Attributes flow bottom-up.

L-attributed grammar

For each production $X \to Y_1 Y_2 \ldots Y_k$,

- $X.syn$ depends on $X.inh$ and all attributes of $Y_1, Y_2, \ldots, Y_k$.
- For all $1 \leq i \leq k$, $Y_i.inh$ depends on $X.inh$ and all attributes of $Y_1, Y_2, \ldots, Y_{i-1}$.

S-attributed grammar

- All attributes are synthesized.
- Attributes flow bottom-up.

L-attributed grammar

For each production $X \rightarrow Y_1 Y_2 \ldots Y_k$,

- $X.syn$ depends on $X.inh$ and all attributes of $Y_1, Y_2, \ldots, Y_k$.
- For all $1 \leq i \leq k$, $Y_i.inh$ depends on $X.inh$ and all attributes of $Y_1, Y_2, \ldots, Y_{i-1}$.

S-attributed grammars are a special case of L-attributed grammars.

LL(1) PARSING, LEFT-ASSOCIATIVITY, AND L-ATTRIBUTED GRAMMARS (1)

A simple grammar for arithmetic expressions using addition and subtraction:

$E \rightarrow T$

$E \rightarrow EAT$

$T \rightarrow n$

$A \rightarrow +$

$A \rightarrow -$

LL(1) PARSING, LEFT-ASSOCIATIVITY, AND L-ATTRIBUTED GRAMMARS (1)

A simple grammar for arithmetic expressions using addition and subtraction:

$E \rightarrow T$

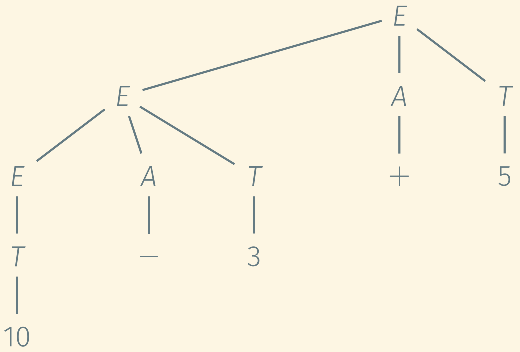
$E \rightarrow EAT$

$T \rightarrow n$

$A \rightarrow +$

$A \rightarrow -$

$10 - 3 + 5$



A simple grammar for arithmetic expressions using addition and subtraction:

$E \to T$

$E \to EAT$

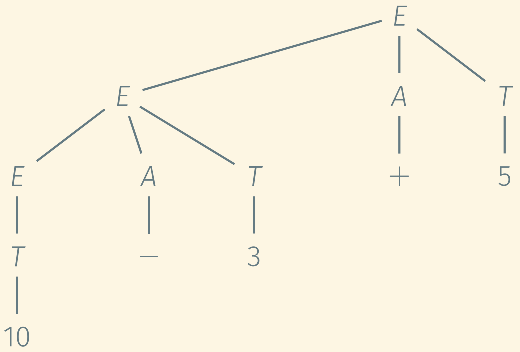
$T \to n$

$A \to +$

$A \to -$

$10 - 3 + 5 = (10 - 3) + 5 = 12$

$10 - 3 + 5 \neq 10 - (3 + 5) = 2$



A simple grammar for arithmetic expressions using addition and subtraction:

$E \rightarrow T$

$E \rightarrow EAT$

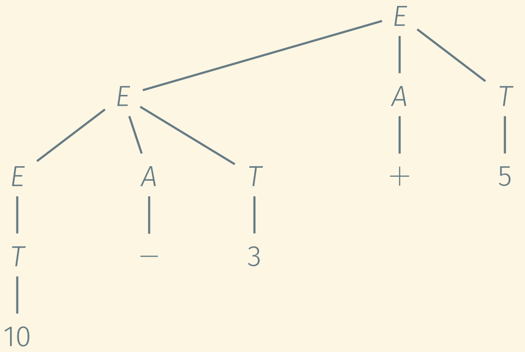
$T \rightarrow n$

$A \rightarrow +$

$A \rightarrow -$

$10 - 3 + 5 = (10 - 3) + 5 = 12$

$10 - 3 + 5 \neq 10 - (3 + 5) = 2$



This grammar captures left-associativity.

A simple grammar for arithmetic expressions using addition and subtraction:

$E \rightarrow T$

$E \rightarrow EAT$

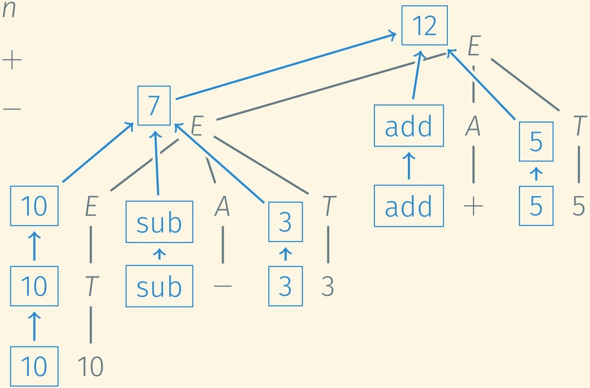
$T \rightarrow n$

$A \rightarrow +$

$A \rightarrow -$

$10 - 3 + 5 = (10 - 3) + 5 = 12$

$10 - 3 + 5 \neq 10 - (3 + 5) = 2$



This grammar captures left-associativity.

A simple grammar for arithmetic expressions using addition and subtraction:

$E \to T$

$E \to EAT$

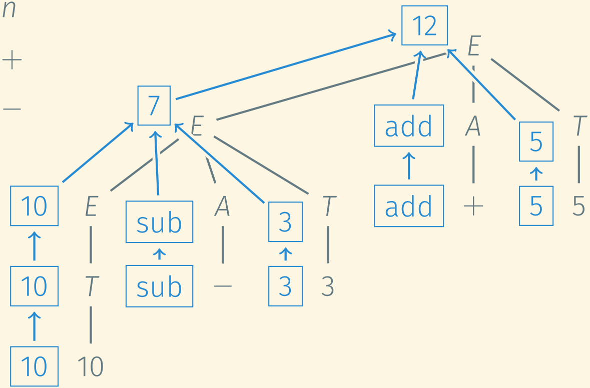
$T \to n$

$A \to +$

$A \to -$

$10 - 3 + 5 = (10 - 3) + 5 = 12$

$10 - 3 + 5 \neq 10 - (3 + 5) = 2$



Rule R	PREDICT(R)
$E \to T$	$\{n\}$
$E \to EAT$	$\{n\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$

This grammar captures left-associativity.

A simple grammar for arithmetic expressions using addition and subtraction:

$E \to T$

$E \to EAT$

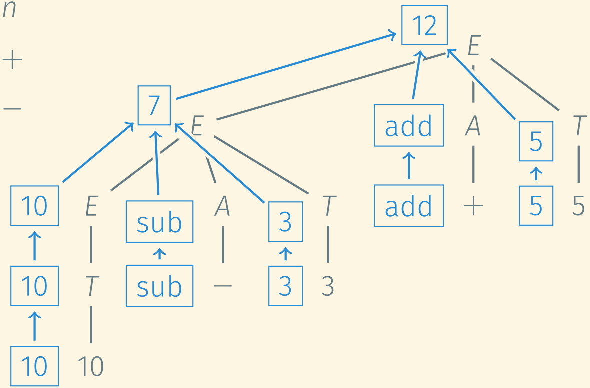
$T \to n$

$A \to +$

$A \to -$

$10 - 3 + 5 = (10 - 3) + 5 = 12$

$10 - 3 + 5 \neq 10 - (3 + 5) = 2$



Rule R	PREDICT(R)
$E \to T$	$\{n\}$
$E \to EAT$	$\{n\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$

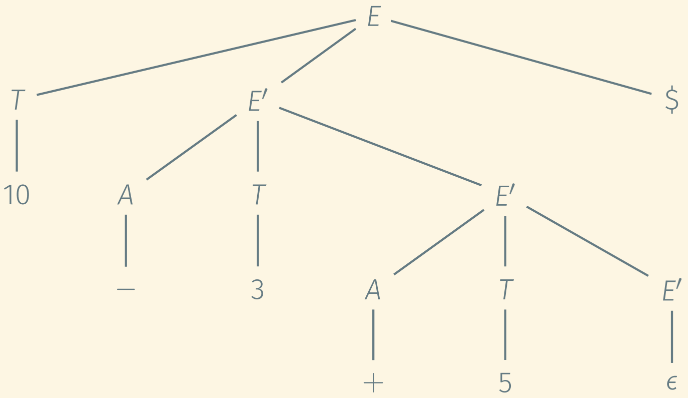
This grammar captures left-associativity. It is not LL(1)!

An LL(1) grammar for the same language:

	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$

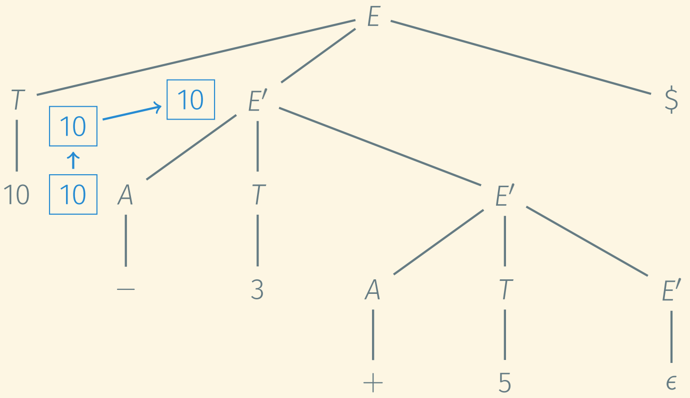
An LL(1) grammar for the same language:

	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$



An LL(1) grammar for the same language:

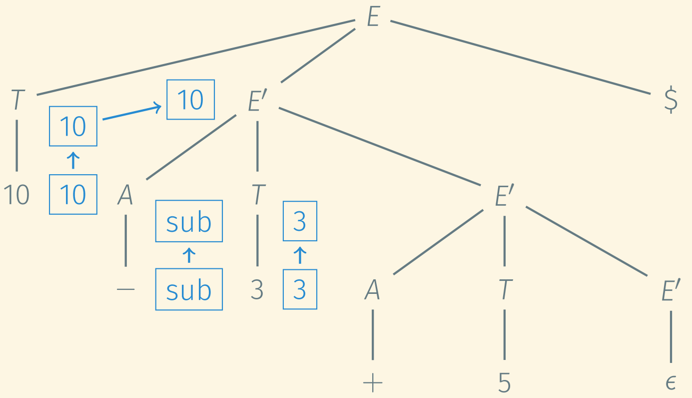
	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$



LL(1) PARSING, LEFT-ASSOCIATIVITY, AND L-ATTRIBUTED GRAMMARS (2)

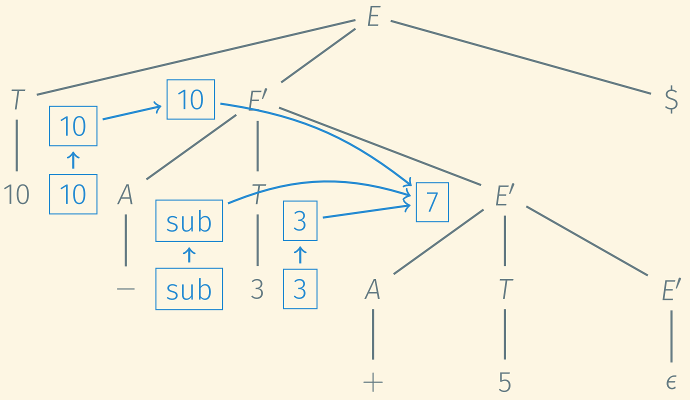
An LL(1) grammar for the same language:

	PREDICT
$E \to TE'\$$	$\{n\}$
$E' \to \epsilon$	$\{\$\}$
$E' \to ATE'$	$\{+, -\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$



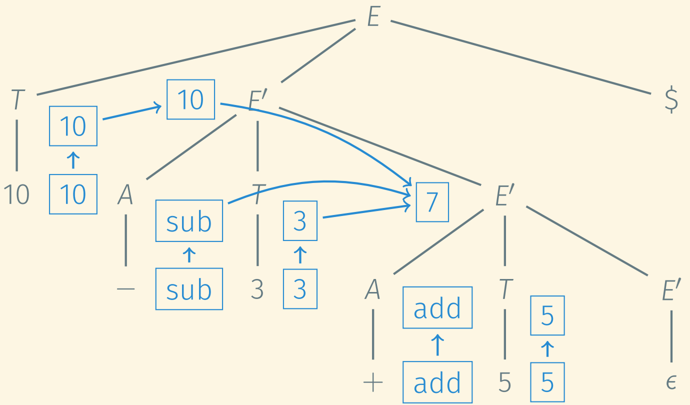
An LL(1) grammar for the same language:

	PREDICT
$E \to TE'\$$	$\{n\}$
$E' \to \epsilon$	$\{\$\}$
$E' \to ATE'$	$\{+, -\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$



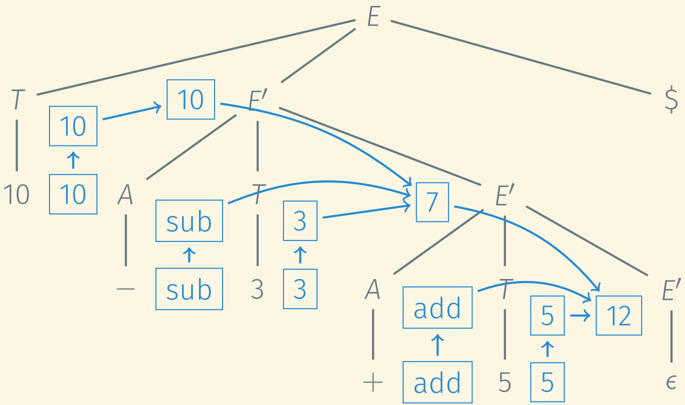
An LL(1) grammar for the same language:

	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$



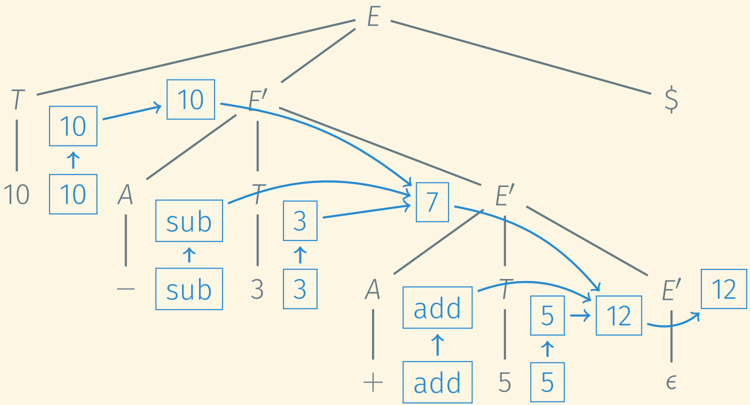
An LL(1) grammar for the same language:

	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$



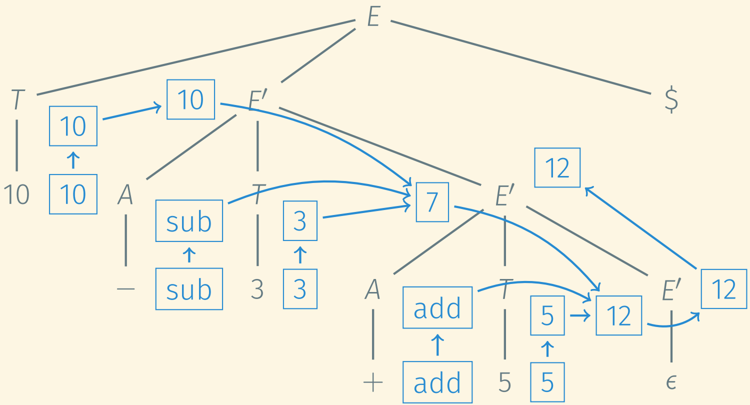
An LL(1) grammar for the same language:

	PREDICT
$E \to TE'\$$	$\{n\}$
$E' \to \epsilon$	$\{\$\}$
$E' \to ATE'$	$\{+, -\}$
$T \to n$	$\{n\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$



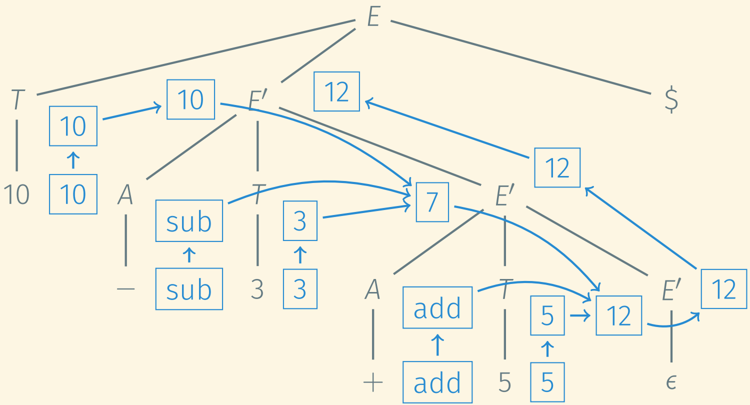
An LL(1) grammar for the same language:

	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$



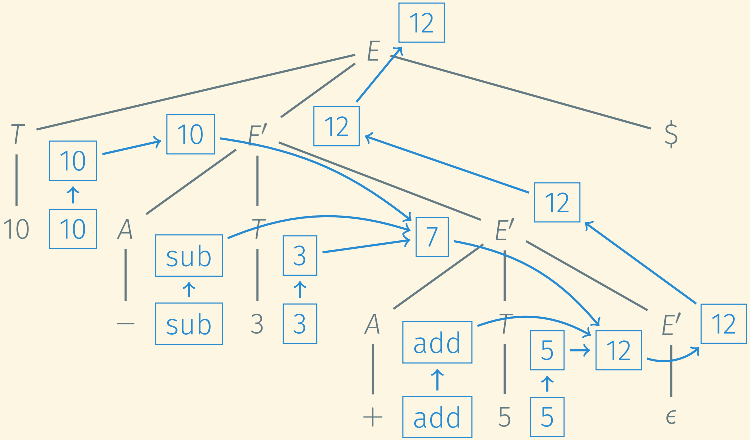
An LL(1) grammar for the same language:

	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$



An LL(1) grammar for the same language:

	PREDICT
$E \rightarrow TE'\$$	$\{n\}$
$E' \rightarrow \epsilon$	$\{\$\}$
$E' \rightarrow ATE'$	$\{+, -\}$
$T \rightarrow n$	$\{n\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$



An LL(1) grammar for the same language:

$$
\begin{array}{ll}
E \to TE'\$ & \{E'.in = T.val; E.val = E'.val\} \\
E' \to \epsilon & \{E'.val = E'.in\} \\
E'_1 \to ATE'_2 & \{E'_2.in = A.fun(E'_1.in, T.val); E'_1.val = E'_2.val\} \\
T \to n & \{T.val = n.val\} \\
A \to + & \{T.fun = \text{add}\} \\
A \to - & \{T.fun = \text{sub}\}
\end{array}
$$

ROAD MAP

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

ROAD MAP

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

ACTION ROUTINES

Action routines are instructions for ad-hoc translation interleaved with parsing.

Parser generators (e.g., `bison` or `yacc`) allow programmers to specify action routines in the grammar.

Action routines can appear anywhere in a rule (as long as the grammar is LL(1)).

ACTION ROUTINES: EXAMPLE

Example:

$$E' \rightarrow AT \; \{\$3.in = \$1.fun(\$0.in, \$2.val)\} \; E' \; \{\$0.val = \$3.val\}$$

Example:

$$E' \rightarrow AT\ \{\$3.in = \$1.fun(\$0.in, \$2.val)\}\ E'\ \{\$0.val = \$3.val\}$$

Corresponding parse function in recursive descent parser:

```
def parseEE(node0):
  node1 = ParseTreeNode()
  node2 = ParseTreeNode()
  node3 = ParseTreeNode()
  parseA(node1)
  parseT(node2)
  node3.op = node1.fun(node0.in, node2.val)
  parseEE(node3)
  node0.val = node3.val
```

ROAD MAP

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

- Syntax, semantics, and semantic analysis
- Attribute grammars
- Action routines
- Abstract syntax trees

Problem with parse trees:

- They represent the full derivation of the program using grammar rules.
- Some grammar variables are there only to aid in parsing (e.g., to eliminate left-recursion or common prefixes).
- Code generator is easier to implement if the output of the parser is as compact as possible.

Abstract syntax tree (AST)

A compressed parse tree that represents the program structure rather than the parsing process.

ABSTRACT SYNTAX TREE: EXAMPLE (1)

$$
\begin{aligned}
Fun &\rightarrow \textbf{fun}\ id\ Stmts\ \textbf{.} \\
Stmts &\rightarrow \epsilon \\
Stmts &\rightarrow Stmt\ Stmts \\
Stmt &\rightarrow \ldots
\end{aligned}
$$

ABSTRACT SYNTAX TREE: EXAMPLE (1)

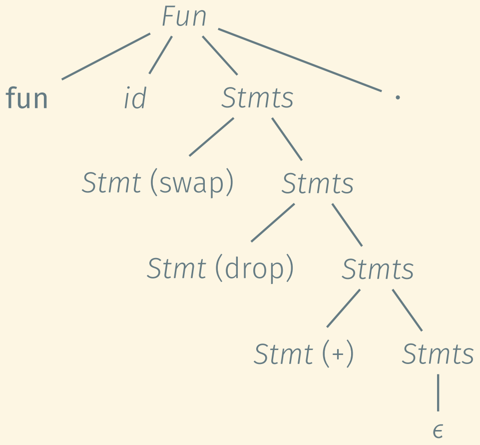
$$
\begin{aligned}
\mathit{Fun} &\rightarrow \textbf{fun}\ \mathit{id}\ \mathit{Stmts}\ \texttt{.} \\
\mathit{Stmts} &\rightarrow \epsilon \\
\mathit{Stmts} &\rightarrow \mathit{Stmt}\ \mathit{Stmts} \\
\mathit{Stmt} &\rightarrow \ldots
\end{aligned}
$$

```
fun foo
  swap drop +
.
```

ABSTRACT SYNTAX TREE: EXAMPLE (1)

$$
\begin{aligned}
\mathit{Fun} &\rightarrow \textbf{fun}\ \mathit{id}\ \mathit{Stmts}\ \texttt{.} \\
\mathit{Stmts} &\rightarrow \epsilon \\
\mathit{Stmts} &\rightarrow \mathit{Stmt}\ \mathit{Stmts} \\
\mathit{Stmt} &\rightarrow \ldots
\end{aligned}
$$

```
fun foo
  swap drop +
.
```



ABSTRACT SYNTAX TREE: EXAMPLE (1)

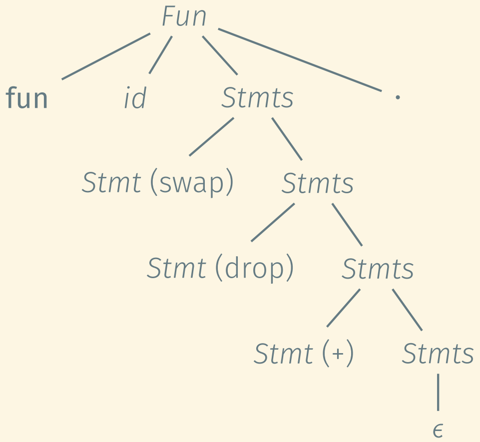
$$Fun \to \textbf{fun}\ id\ Stmts\ \texttt{.}$$

$$Stmts \to \epsilon$$

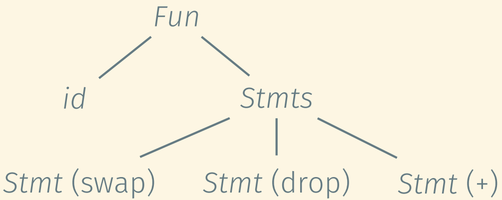
$$Stmts \to Stmt\ Stmts$$

$$Stmt \to \ldots$$

```
fun foo
  swap drop +
.
```



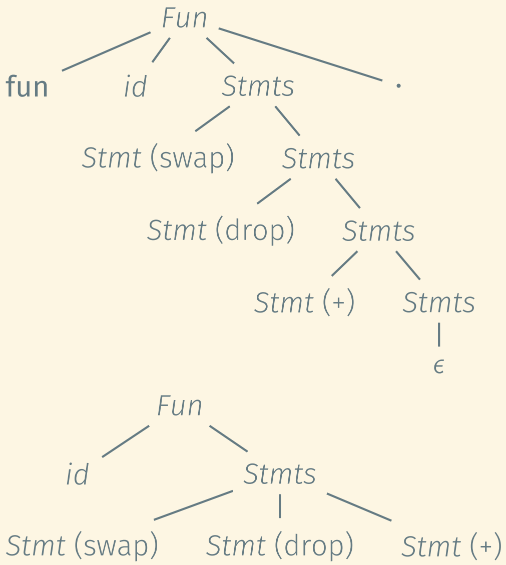
AST:



ABSTRACT SYNTAX TREE: EXAMPLE (2)

```
def parseFun(node0):
  node1 = ParseTreeNode()
  node2 = ParseTreeNode()
  matchFunKW()
  parseId(node1)
  parseStatements(node2)
  matchEndKW()

def parseStatements(node0):
  if next token is .:
    node0.statements = []
  else:
    node1 = ParseTreeNode()
    node2 = ParseTreeNode()
    parseStatement(node1)
    parseStatement(node2)
    node0.statements = \
      [node1.statement] + \
      node2.statements
```



SUMMARY

- Semantic analysis augments the parsing process to represent the meaning of the program.
- The output is often an annotated abstract syntax tree (AST).
- Attribute grammars and action routines are used to construct the AST.