Lecture 3: Static Analysis Principles – Abstract Syntax Trees (AST)

Passive Testing Techniques for Communication Protocols

Dr. Jorge López, PhD.
jorgelopezcoronado[at]gmail.com

National Research
Tomsk
State
University

February 15, 2016

ACKNOWLEDGMENTS

Some content was taken from: Saman Amarasinghe, and Martin Rinard. 6.035 Computer Language Engineering, Spring 2010. (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu. License: Creative Commons BY-NC-SA

OUTLINE

FROM CFG TO "TREES"

GRAMMAR AMBIGUITIES

ABSTRACT SYNTAX TREES

REMEMBER THE CFG DERIVATION?

Remember the CFG derivation?

Grammar

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Example derivation

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)
- $int\ op\ opar\ Expr\ cpar$ (4)
- $int\ op\ opar\ Expr\ op\ Expr\ cpar$ (2)
- $int\ op\ opar\ int\ op\ Expr\ cpar$ (3)
- $int\ op\ opar\ int\ op\ int\ cpar$ (3)
- 10 / (7 − 5)

REMEMBER THE CFG DERIVATION?

Grammar

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Example derivation

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)
- $int\ op\ opar\ Expr\ cpar$ (4)
- $int\ op\ opar\ Expr\ op\ Expr\ cpar$ (2)
- $int\ op\ opar\ int\ op\ Expr\ cpar$ (3)
- $int\ op\ opar\ int\ op\ int\ cpar$ (3)
- 10 / (7 − 5)

It has a natural tree-like structure!

PARSE TREES

Definition

PARSE TREES

Definition

- Formally, a directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure).

PARSE TREES

Definition

- Formally, a directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure).

Structure

- Terminal symbols are leafs
- Non-terminal symbols are intermediate vertices in the tree
- Initial symbol is the tree root (therefore a rooted tree)
- Production rules define arcs.

PARSE TREES

Definition

- Formally, a directed graph $T =< V, A >$, where V is a set of vertices (nodes), A is a set of ordered arcs formed by a pairs $(v_1, v_2) \in V \times V$, in which each two vertices are connected by a unique simple path (tree-like structure).

Structure

- Terminal symbols are leafs
- Non-terminal symbols are intermediate vertices in the tree
- Initial symbol is the tree root (therefore a rooted tree)
- Production rules define arcs.

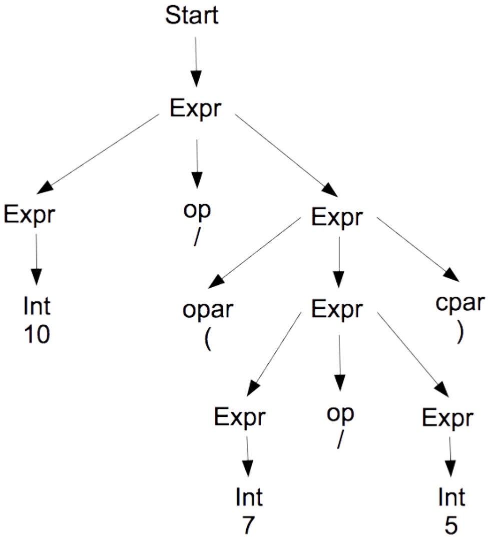
Let's take a look at one…

Parse Trees (cont.)

$10/(7-5)$ parse tree

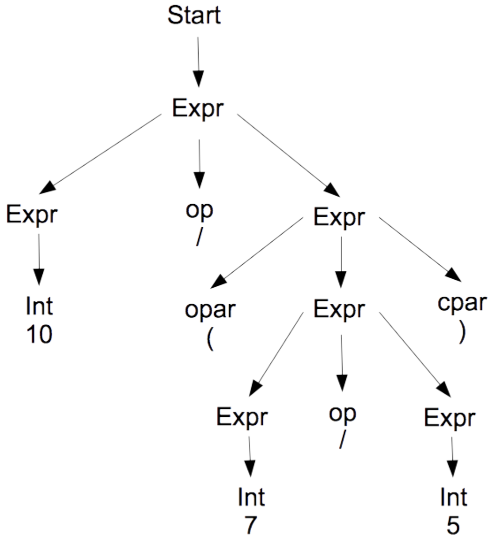
PARSE TREES (CONT.)

10/(7 – 5) parse tree



Parse Trees (cont.)

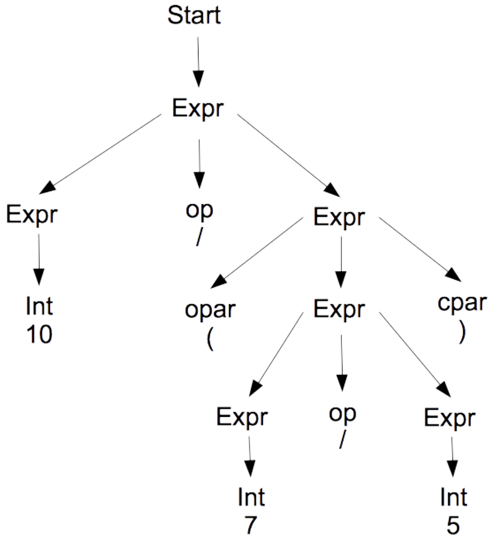
$10/(7-5)$ parse tree



1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Parse Trees (cont.)

$10/(7 - 5)$ parse tree

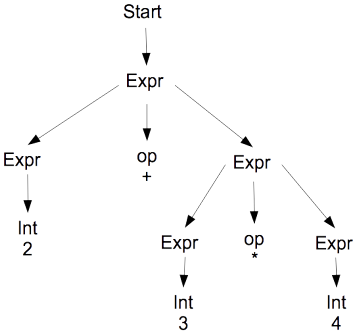


1. *Start* $\mapsto$ *Expr*
2. *Expr* $\mapsto$ *Expr op Expr*
3. *Expr* $\mapsto$ *int*
4. *Expr* $\mapsto$ *opar Expr cpar*

Please, derive the parse tree for $2 + 3 * 4$

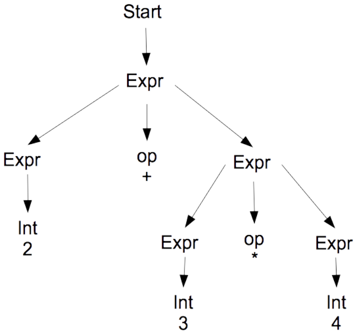
PARSE TREES (CONT. CONT.)

$2 + 3 * 4$ parse tree

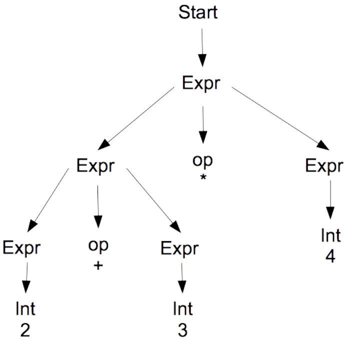


PARSE TREES (CONT. CONT.)

$2 + 3 * 4$ parse tree



$2 + 3 * 4$ parse tree!



Grammar Ambiguities
(and parse trees)

GRAMMAR AMBIGUITIES

Grammar Ambiguity…

GRAMMAR AMBIGUITIES

Grammar Ambiguity…

- Occurs when multiple derivations (and therefore parse trees) exist for a single string

GRAMMAR AMBIGUITIES

Grammar Ambiguity…

- Occurs when multiple derivations (and therefore parse trees) exist for a single string
- Ambiguity in the grammar reflects ambiguity in the language semantics

Grammar Ambiguities

Grammar Ambiguity…

- Occurs when multiple derivations (and therefore parse trees) exist for a single string
- Ambiguity in the grammar reflects ambiguity in the language semantics
  - For computer languages this is highly undesirable!

GRAMMAR AMBIGUITIES

Grammar Ambiguity…

- Occurs when multiple derivations (and therefore parse trees) exist for a single string
- Ambiguity in the grammar reflects ambiguity in the language semantics
  - For computer languages this is highly undesirable!
  - Non-determinism in computer code is something that humanity is not ready for…

GRAMMAR AMBIGUITIES

Grammar Ambiguity…

- Occurs when multiple derivations (and therefore parse trees) exist for a single string
- Ambiguity in the grammar reflects ambiguity in the language semantics
    - For computer languages this is highly undesirable!
    - Non-determinism in computer code is something that humanity is not ready for…

Solution?

GRAMMAR MODIFICATIONS (HACKING)

Recursiveness...

Grammar modifications (hacking)

Recursiveness…

- Left recursive production $P \mapsto P\ \alpha$, where $P \in T \wedge \alpha \in (T \cap U)^*$

GRAMMAR MODIFICATIONS (HACKING)

Recursiveness…

- Left recursive production $P \mapsto P\ \alpha$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Left recursive production ↦ Left recursive grammar

GRAMMAR MODIFICATIONS (HACKING)

Recursiveness…

- Left recursive production $P \mapsto P\ \alpha$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Left recursive production ↦ Left recursive grammar
- Right recursive production $P \mapsto \alpha\ P$, where $P \in T \wedge \alpha \in (T \cap U)^*$

Grammar modifications (hacking)

Recursiveness…

- Left recursive production $P \mapsto P\ \alpha$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Left recursive production ↦ Left recursive grammar
- Right recursive production $P \mapsto \alpha\ P$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Right recursive production ↦ Right recursive grammar

GRAMMAR MODIFICATIONS (HACKING)

Recursiveness…

- Left recursive production $P \mapsto P\ \alpha$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Left recursive production ↦ Left recursive grammar
- Right recursive production $P \mapsto \alpha\ P$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Right recursive production ↦ Right recursive grammar

Associativity

GRAMMAR MODIFICATIONS (HACKING)

Recursiveness…

- Left recursive production $P \mapsto P\ \alpha$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Left recursive production ↦ Left recursive grammar
- Right recursive production $P \mapsto \alpha\ P$, where $P \in T \wedge \alpha \in (T \cap U)^*$
- Right recursive production ↦ Right recursive grammar

Associativity

- Try to avoid left and right recursion for the same production rules, leave only one, for instance, only right recursiveness ↦ **right associativity**

Grammar modifications (hacking) (cont.)

GRAMMAR MODIFICATIONS (HACKING) (CONT.)

Original Grammar

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Grammar modifications (hacking) (cont.)

Original Grammar

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Hacked Grammar (associativity)

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Grammar modifications (hacking) (cont.)

Original Grammar

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Hacked Grammar (associativity)

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Unique $2 + 3 * 4$ parse tree

Grammar modifications (hacking) (cont.)

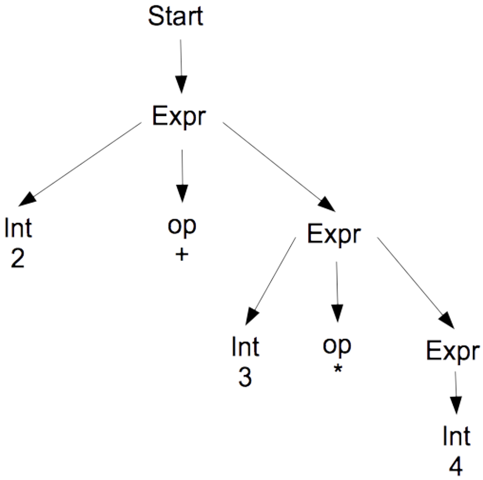
Original Grammar

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Hacked Grammar (associativity)

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Unique $2 + 3 * 4$ parse tree



GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful

GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful, sometimes…

GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful, sometimes…

- For the previous grammar, $2 * 3 + 4$ will give an incorrect parse tree given the naturally established **precedence**

Grammar modifications (hacking) (III)

Associativity can be useful, sometimes…

- For the previous grammar, $2 * 3 + 4$ will give an incorrect parse tree given the naturally established **precedence**
- If we hack the grammar to make it left associative, then we'd have problems with $2 + 3 * 4$, for instance…

GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful, sometimes…

- For the previous grammar, $2 * 3 + 4$ will give an incorrect parse tree given the naturally established **precedence**
- If we hack the grammar to make it left associative, then we'd have problems with $2 + 3 * 4$, for instance…

Precedence

GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful, sometimes…

- For the previous grammar, $2 * 3 + 4$ will give an incorrect parse tree given the naturally established **precedence**
- If we hack the grammar to make it left associative, then we'd have problems with $2 + 3 * 4$, for instance…

Precedence

- Group operators rules into precedence levels

GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful, sometimes…

- For the previous grammar, $2 * 3 + 4$ will give an incorrect parse tree given the naturally established **precedence**
- If we hack the grammar to make it left associative, then we'd have problems with $2 + 3 * 4$, for instance…

Precedence

- Group operators rules into precedence levels
- Non-terminal symbols generated separately for each precedence level

GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful, sometimes…

- For the previous grammar, $2 * 3 + 4$ will give an incorrect parse tree given the naturally established **precedence**
- If we hack the grammar to make it left associative, then we'd have problems with $2 + 3 * 4$, for instance…

Precedence

- Group operators rules into precedence levels
- Non-terminal symbols generated separately for each precedence level
- Within each non-terminal symbol associativity is possible

GRAMMAR MODIFICATIONS (HACKING) (III)

Associativity can be useful, sometimes...

- For the previous grammar, $2 * 3 + 4$ will give an incorrect parse tree given the naturally established **precedence**
- If we hack the grammar to make it left associative, then we'd have problems with $2 + 3 * 4$, for instance...

Precedence

- Group operators rules into precedence levels
- Non-terminal symbols generated separately for each precedence level
- Within each non-terminal symbol associativity is possible
- Higher precedence↦inner production rule (bind first)

GRAMMAR MODIFICATIONS (HACKING) (IV)

Grammar modifications (hacking) (IV)

Original Grammar

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

GRAMMAR MODIFICATIONS (HACKING) (IV)

Original Grammar

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Hacked Grammar (precedence)

- $mop = /|*$
- $aop = +|-$
- $int, opar, cpar$ same

1. $Start \mapsto Expr$

Grammar modifications (hacking) (IV)

Original Grammar

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Hacked Grammar (precedence)

- $mop = /|*$
- $aop = +|-$
- $int, opar, cpar$ same

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ aop\ Term$
3. $Expr \mapsto Term$

Grammar modifications (hacking) (IV)

Original Grammar

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Hacked Grammar (precedence)

- $mop = /|*$
- $aop = +|-$
- $int, opar, cpar$ same

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ aop\ Term$
3. $Expr \mapsto Term$
4. $Term \mapsto Term\ mop\ Num$
5. $Term \mapsto Num$

Grammar modifications (hacking) (IV)

Original Grammar

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto int\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Hacked Grammar (precedence)

- $mop = /|*$
- $aop = +|-$
- $int, opar, cpar$ same

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ aop\ Term$
3. $Expr \mapsto Term$
4. $Term \mapsto Term\ mop\ Num$
5. $Term \mapsto Num$
6. $Num \mapsto int$
7. $Num \mapsto opar\ Expr\ cpar$

EXERCISES – AMBIGUOUS GRAMMARS

Remove ambiguity from:

Exercises – Ambiguous Grammars

Remove ambiguity from:

- $S \mapsto SS|a|b$

Exercises – Ambiguous Grammars

Remove ambiguity from:

- $S \mapsto SS|a|b$
  - One way:

Exercises – Ambiguous Grammars

Remove ambiguity from:

- $S \mapsto SS|a|b$
    - One way:
        - $S \mapsto S2\ S|a|b$
        - $S2 \mapsto a|b$

Exercises – Ambiguous Grammars

Remove ambiguity from:

- $S \mapsto SS|a|b$
  - One way:
    - $S \mapsto S2\ S|a|b$
    - $S2 \mapsto a|b$
  - Another

Exercises – Ambiguous Grammars

Remove ambiguity from:

- $S \mapsto SS|a|b$
  - One way:
    - $S \mapsto S2\ S|a|b$
    - $S2 \mapsto a|b$
  - Another
    - $S \mapsto aS|a|b|bS$

EXERCISES – AMBIGUOUS GRAMMARS

Remove ambiguity from:

- $S \mapsto SS|a|b$
  - One way:
    - $S \mapsto S2\ S|a|b$
    - $S2 \mapsto a|b$
  - Another
    - $S \mapsto aS|a|b|bS$
- $S \mapsto SS|a|c$; *ca* has priority over *ac*

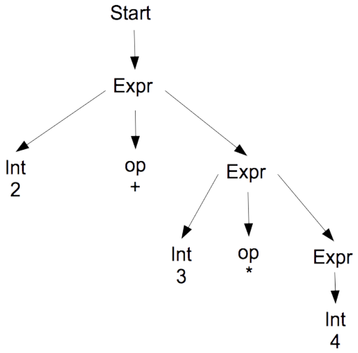
EXERCISES – AMBIGUOUS GRAMMARS

Remove ambiguity from:

- $S \mapsto SS|a|b$
  - One way:
    - $S \mapsto S2\ S|a|b$
    - $S2 \mapsto a|b$
  - Another
    - $S \mapsto aS|a|b|bS$
- $S \mapsto SS|a|c$; *ca* has priority over *ac*
  - **Homework**

CONCRETE PARSE TREES. . .

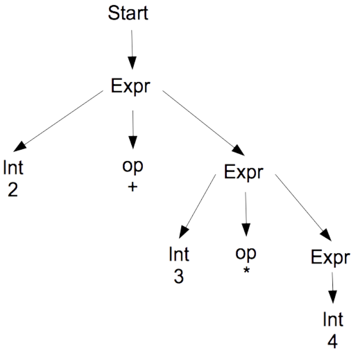
Old $2 + 3 * 4$ parse tree

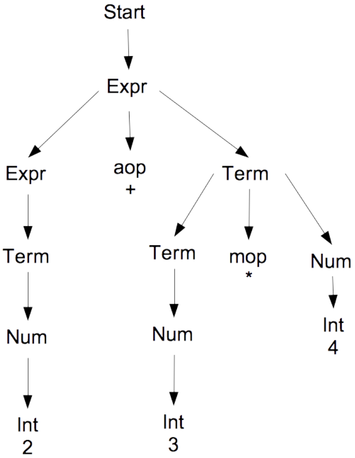


Concrete Parse trees. . .

New $2 + 3 * 4$ parse tree!

Old $2 + 3 * 4$ parse tree





Abstract Syntax Trees (AST)

The desired structure

PARSING...

A Parser (PDA)

PARSING. . .

A Parser (PDA)

- Can be generated automatically by a parser generator (takes as input a CFG and produces a parser)

Parsing. . .

A Parser (PDA)

- Can be generated automatically by a parser generator (takes as input a CFG and produces a parser)
- Takes a CFG derivation (a sentence / source code) and produces a "parse tree"

PARSING. . .

A Parser (PDA)

- Can be generated automatically by a parser generator (takes as input a CFG and produces a parser)
- Takes a CFG derivation (a sentence / source code) and produces a "parse tree"
- Its goal is to produce an AST, not a concrete parse tree

Parsing. . .

A Parser (PDA)

- Can be generated automatically by a parser generator (takes as input a CFG and produces a parser)
- Takes a CFG derivation (a sentence / source code) and produces a "parse tree"
- Its goal is to produce an AST, not a concrete parse tree

AST vs Concrete Parse Tree

- Concrete used to parse unambiguously

PARSING...

A Parser (PDA)

- Can be generated automatically by a parser generator (takes as input a CFG and produces a parser)
- Takes a CFG derivation (a sentence / source code) and produces a "parse tree"
- Its goal is to produce an AST, not a concrete parse tree

AST vs Concrete Parse Tree

- Concrete used to parse unambiguously
- Concrete too complex, we need simpler, we need an AST

ABSTRACT SYNTAX TREES

What it is

ABSTRACT SYNTAX TREES

What it is

- A concrete parse tree omits unnecessary terminal and non-terminal symbols

ABSTRACT SYNTAX TREES

What it is

- A concrete parse tree omits unnecessary terminal and non-terminal symbols
  - For example: () or intermediate $< Term >$ symbols

ABSTRACT SYNTAX TREES

What it is

- A concrete parse tree omits unnecessary terminal and non-terminal symbols
  - For example: () or intermediate $< Term >$ symbols

How to obtain it?

ABSTRACT SYNTAX TREES

What it is

- A concrete parse tree omits unnecessary terminal and non-terminal symbols
  - For example: () or intermediate $< \mathit{Term} >$ symbols

How to obtain it?

- Start with ambiguous grammar (probably)

ABSTRACT SYNTAX TREES

What it is

- A concrete parse tree omits unnecessary terminal and non-terminal symbols
  - For example: () or intermediate $< Term >$ symbols

How to obtain it?

- Start with ambiguous grammar (probably)
- Hack the grammar to obtain concrete parse tree

ABSTRACT SYNTAX TREES

What it is

- A concrete parse tree omits unnecessary terminal and non-terminal symbols
  - For example: () or intermediate $< Term >$ symbols

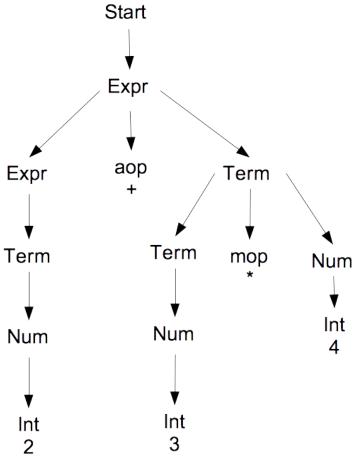
How to obtain it?

- Start with ambiguous grammar (probably)
- Hack the grammar to obtain concrete parse tree
- Remove undesired symbols and obtain the AST

Concrete Parse tree ↦ AST

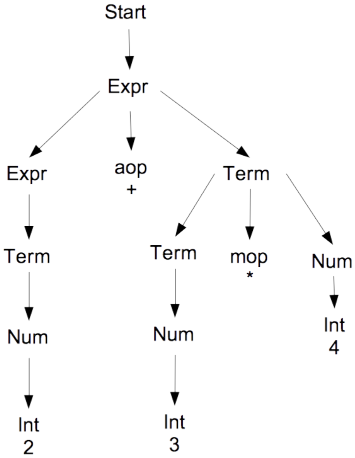
CONCRETE PARSE TREE ↦ AST

Concrete parse tree for
$2 + 3 * 4$

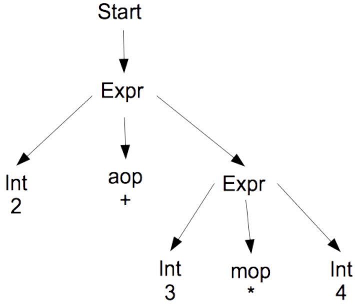


Concrete Parse tree ↦ AST

Concrete parse tree for $2 + 3 * 4$



AST for $2 + 3 * 4$



ASTs – Final Remarks

ASTs – Final Remarks

- Commonly, people show the AST of the generic CFG, representing a somehow ambiguous grammar

ASTS – FINAL REMARKS

- Commonly, people show the AST of the generic CFG, representing a somehow ambiguous grammar
- Many known parser generators, based on this approach, YACC, Bison, ANTLR, etc...

ASTs – Final Remarks

- Commonly, people show the AST of the generic CFG, representing a somehow ambiguous grammar
- Many known parser generators, based on this approach, YACC, Bison, ANTLR, etc...
- Recently ($\sim$2011...) some attention to regular expression derivatives to parse was introduced thanks to YACC is dead [Might and Darais, 2010], and all follow-up from it

ASTs – Final Remarks

- Commonly, people show the AST of the generic CFG, representing a somehow ambiguous grammar
- Many known parser generators, based on this approach, YACC, Bison, ANTLR, etc...
- Recently (∼2011. . .) some attention to regular expression derivatives to parse was introduced thanks to YACC is dead [Might and Darais, 2010], and all follow-up from it
- Until today, YACC is very alive. . .

ASTs – Final Remarks

- Commonly, people show the AST of the generic CFG, representing a somehow ambiguous grammar
- Many known parser generators, based on this approach, YACC, Bison, ANTLR, etc...
- Recently (∼2011. . .) some attention to regular expression derivatives to parse was introduced thanks to YACC is dead [Might and Darais, 2010], and all follow-up from it
- Until today, YACC is very alive. . .
- See you on Wednesday, we'll talk about some Static Analysis now that we know on which structure to perform it (please note that: this theory applies to much more than just Static Analysis)