Compiler verification for fun and profit

a week from now, 8:55am in Salle Polyvalente

Xavier Leroy Inria Paris–Rocquencourt

…Compilers are, however, vulnerable to miscompilation: bugs in the compiler that cause incorrect code to be generated from a correct source code, possibly invalidating the guarantees so painfully obtained by source-level formal verification. Recent experimental studies show that many widely-used production-quality compilers suffer from miscompilation. The formal verification of compilers and related code generators is a radical, mathematically-grounded answer to the miscompilation issue. By applying formal verification (typically, interactive theorem proving) to the compiler itself, it is possible to guarantee that the compiler preserves the semantics of the source programs it transforms, or at least preserves the properties of interest that were formally verified over the source programs…

Exercise: Unary Minus

1) Show that the grammar

$$A ::= -A$$
$$A ::= A - id$$
$$A ::= id$$

is ambiguous by finding a string that has two different parse trees. Show those parse trees.

2) Make two different unambiguous grammars for the same language:

a) One where prefix minus binds stronger than infix minus.

b) One where infix minus binds stronger than prefix minus.

3) Show the syntax trees using the new grammars for the string you used to prove the original grammar ambiguous.

4) Give a regular expression describing the same language.

Unary Minus Solution Sketch

1) An example of a string with two parse trees is

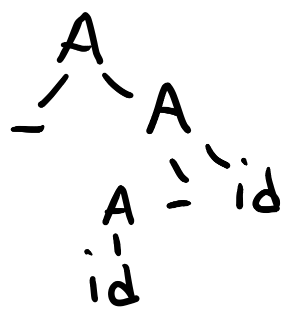
- id - id

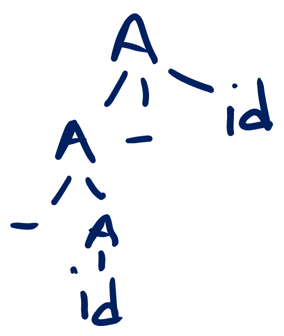
The two parse trees are generated by these imaginary parentheses (shown red): -(id-id) (-id)-id

and can generated by these derivations that give different parse trees

A => -A => - A - id => - id - id

A => A - id => - A - id => - id - id





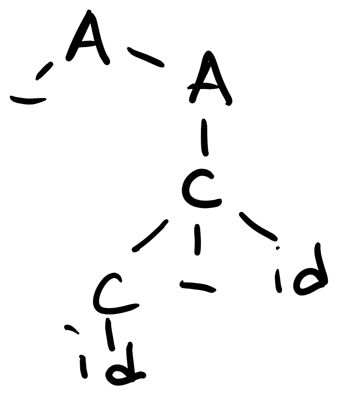
2) a) prefix minus binds stronger:

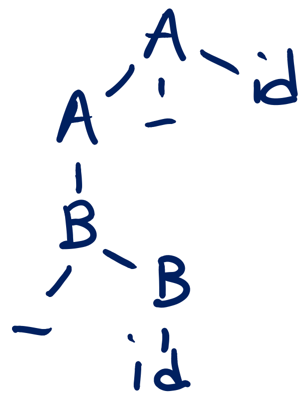
A ::= B | A - id B ::= -B | id

b) infix minus binds stronger

A ::= C | -A C ::= id | C - id

3) in two trees that used to be ambiguous instead of some A's we have B's in a) grammar or C's in b) grammar.





4) -*id(-id)*

Exercise: Basic Dangling Else

The dangling-else problem happens when the conditional statements are parsed using the following grammar.

S ::= id **:=** id

S ::= **if** id **then** S

S ::= **if** id **then** S **else** S

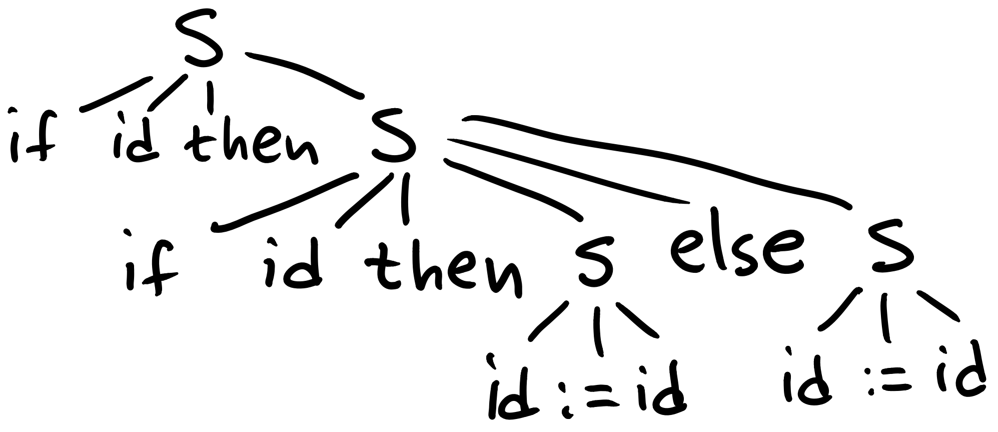
1) To prove that the grammar is ambiguous, find two parse trees that derive the same string

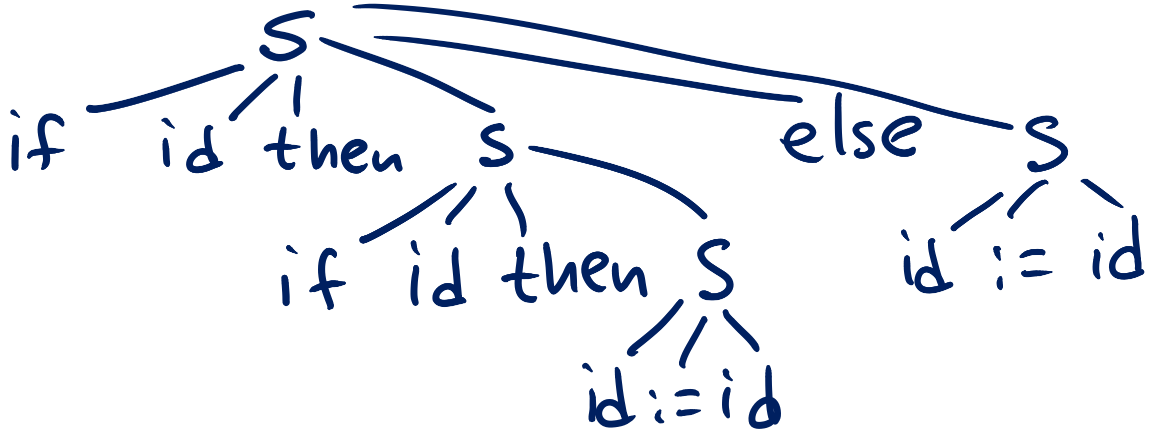
2) Find an unambiguous grammar that accepts the same strings and matches the else statement with the nearest unmatched if.

- This is a real issue in languages like C, Java
  - resolved by saying **else** binds to innermost **if**

Dangling Else: Ambiguous Input

```
1) if id then
       if id then
           id := id
       else
          id := id
```





Dangling Else: Solution

In class we have seen several wrong solutions, which are either still ambiguous, or remove some strings from the grammar.
The following is a correct solution:

S ::= id **:=** id
S ::= **if** id **then** S
S ::= **if** id **then** S_1 **else** S
S_1 ::= id **:=** id
S_1 ::= **if** id **then** S_1 **else** S_1

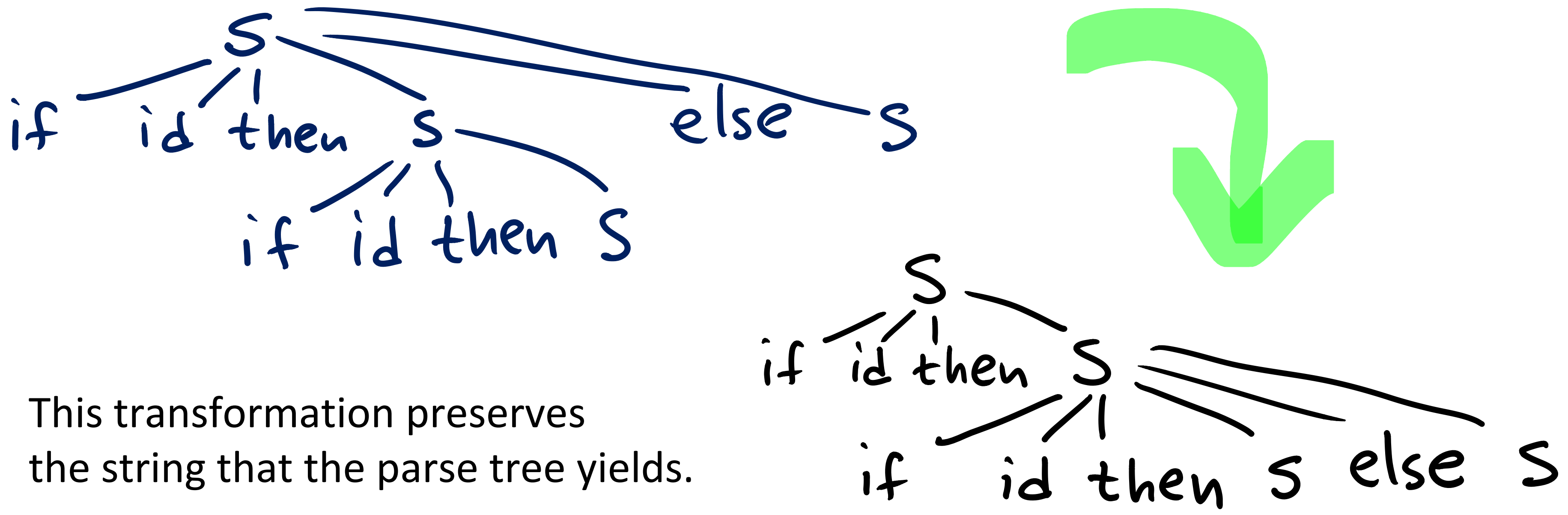
It can be parsed using recursive descent (LL(1)) parsers, so it is non-ambiguous. We will see later how to show this mechanically.

Dangling Else: Solution (Continued)

It is clear that every string of the new grammar can be derived by the old one: take a parse tree of the new grammar, and replace S_1 with S. We obtain a valid parse tree of the old grammar.

It is less obvious that the converse holds: for every parse tree in the old grammar, we can find a parse tree in the new grammar that yields the same string. We prove this as follows.

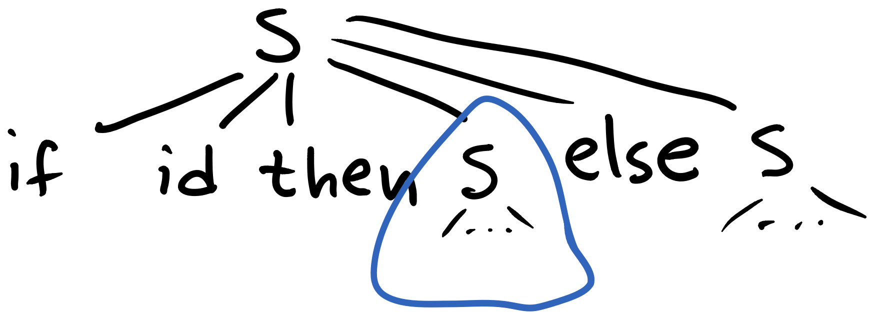
Consider a derivation in the old grammar. Apply the following transformation to all subtrees in the old tree as long long as possible:



This transformation preserves the string that the parse tree yields.

Dangling Else: Solution (Continued)

In the resulting tree, consider every subtree that appears like this:



And replace all occurrences of S inside such subtree with S_1

The crucial observation is that inside such tree each "then" has a corresponding else branch. This is clear for the top level, otherwise we would have moved the else shown in the picture one level down using the transformation on the previous slide. By applying this reasoning to the if then else inside, we also conclude (by induction) that all if nodes have else branches.

This means that we have a valid tree according to the new grammar.

Exercise: Dangling Else in Context

Suppose that in addition assignments and if statements we have statement sequencing:

```
S ::= S ; S
S ::= id := id
S ::= if id then S
S ::= if id then S else S
```

Find an unambiguous grammar that accepts the same conditional statements, matches the else statement with the nearest unmatched if, and treats the priority of ";" similarly to Java.

Sources of Ambiguity in this Example

- Ambiguity arises in this grammar here due to:
  - dangling **else**
  - binary rule for sequence (;) as for parentheses
  - priority between if-then-else and semicolon (;)

```
if p1
   if p2
     z  = x;
     u = z          // last assignment is not inside if
```

Wrong parse tree -> wrong generated code

How we Solved It

We identified a wrong tree and tried to refine the grammar to prevent it, by making a copy of the rules. Also, we changed some rules to disallow sequences inside if-then-else and make sequence rule non-ambiguous. The end result is something like this:

S::= ε |A S // a way to write S::=A*
A ::= id **:=** id
A ::= **if** id **then** A
A ::= **if** id **then** A' **else** A
A' ::= id **:=** id
A' ::= **if** id **then** A' **else** A'

(At some point we had a useless rule, so we deleted it.)

Note we cannot have multiple statements inside if branches. We therefore looked at what a grammar would need, to allow building ASTs with sequences inside if-then-else. It would add a case for blocks, like this:

A ::= **{** S **}**
A' ::= **{** S **}**

We could factor out some common definitions (e.g. define A in terms of A'), but that is not important for this problem.

Formalizing and Automating Recursive Descent: LL(1) Parsers

Task: Rewrite Grammar to make it suitable for recursive descent parser

- Assume the priorities of operators as in Java

```
expr ::= expr (+|-|*|/) expr
       | name | `(’ expr `)’
name ::= ident
```

k1

Grammar vs Recursive Descent Parser

```
expr ::= term termList
termList ::= + term termList
           |  - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
             | / factor factorList
             | ε
factor ::= name | ( expr )
name ::= ident
```

```
def expr = { term; termList }
def termList =
  if (token==PLUS) {
     skip(PLUS); term; termList
  } else if (token==MINUS)
     skip(MINUS); term; termList
  }
def term = { factor; factorList }
...
def factor =
  if (token==IDENT) name
  else if (token==OPAR) {
    skip(OPAR); expr; skip(CPAR)
  } else error("expected ident or )")
```

Note that the abstract trees we would create in this example do not strictly follow parse trees.

Rough General Idea

$$\boxed{\begin{array}{ll} A ::= & B_1 \ldots B_p \\ & \mid C_1 \ldots C_q \\ & \mid D_1 \ldots D_r \end{array}} \quad \Rightarrow$$

```
def A =
  if (token ∈ T1) {
    $B_1$ ... $B_p$
  else if (token ∈ T2) {
    $C_1$ ... $C_q$
  } else if (token ∈ T3) {
    $D_1$ ... $D_r$
  } else error("expected T1,T2,T3")
```

where:

T1 = **first**(B_1 ... B_p)

T2 = **first**(C_1 ... C_q)

T3 = **first**(D_1 ... D_r)

first(B_1 ... B_p) = $\{a \in \Sigma \mid B_1 \ldots B_p \Rightarrow \ldots \Rightarrow aw\}$

T1, T2, T3 should be **disjoint** sets of tokens.

Computing **first** in the example

```
expr ::= term termList
termList ::= + term termList
           |  - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
             | / factor factorList
             | ε
factor ::= name | ( expr )
name ::= ident
```

first(name) = {**ident**}

first(**(** expr **)**) = { **(** }

first(factor) = first(name)

U first(**(** expr **)**)

= {**ident**} U{ **(** }

= {**ident**, **(** }

first(***** factor factorList) = { ***** }

first(**/** factor factorList) = { **/** }

first(factorList) = { *****, **/** }

first(term) = first(factor) = {**ident**, **(** }

first(termList) = { **+** , **-** }

first(expr) = first(term) = {**ident**, **(** }

Algorithm for **first**

Given an arbitrary context-free grammar with a set of rules of the form X ::= $Y_1 \ldots Y_n$ compute first for each right-hand side and for each symbol.

How to handle

- alternatives for one non-terminal
- sequences of symbols
- nullable non-terminals
- recursion

Rules with Multiple Alternatives

$$\boxed{\begin{array}{l} A ::= \; B_1 \ldots B_p \\ \quad\;\;\, \mid C_1 \ldots C_q \\ \quad\;\;\, \mid D_1 \ldots D_r \end{array}} \Rightarrow \boxed{\begin{array}{l} \text{first}(A) = \; \text{first}(B_1 \ldots B_p) \\ \qquad\qquad \cup \; \text{first}(C_1 \ldots C_q) \\ \qquad\qquad \cup \; \text{first}(D_1 \ldots D_r) \end{array}}$$

Sequences

$\boxed{\text{first}(B_1 \ldots B_p) = \text{first}(B_1)}$ if not nullable(B_1)

$\boxed{\text{first}(B_1 \ldots B_p) = \text{first}(B_1) \cup \ldots \cup \text{first}(B_k)}$

if nullable(B_1), ..., nullable(B_{k-1}) and
not nullable(B_k) or $k=p$

Abstracting into Constraints

recursive grammar: constraints over finite sets: expr' is first(expr)

```
expr ::= term termList
termList ::= + term termList
           |  - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
             | / factor factorList
             | ε
factor ::= name | ( expr )
name ::= ident
```

```
expr' = term'
termList' = {+}
            U {-}

term' = factor'
factorList' = {*}
              U { / }

factor' = name' U { ( }
name' = { ident }
```

nullable: termList, factorList

For this nice grammar, there is no recursion in constraints. Solve by substitution.

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

$\Rightarrow$

S' = X' U Y'
X' = {b} U S'
Y' = Z' U X' U Y'
Z' = ∅ U {a}

terminals: **a,b**
non-terminals: S, X, Y, Z

reachable (from S): S, X, Y, Z
productive: S, X, Y, Z
nullable: Z,

First sets of terminals:
S', X', Y', Z' $\subseteq$ {a,b}

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

$\Rightarrow$

S' = X' U Y'
X' = {b} U S'
Y' = Z' U **X'**   U Y'
Z' = {a}

terminals: **a**,**b**
non-terminals: S, X, Y, Z

reachable (from S): S, X, Y, Z
productive: X, Z, S, Y
nullable: Z

These constraints are recursive.
How to solve them?

$$S', X', Y', Z' \subseteq \{a,b\}$$

How many candidate solutions

- in this case?
- for k tokens, n nonterminals?

Iterative Solution of **first** Constraints

	S'	X'	Y'	Z'
1.	{}	{}	{}	{}
2.	{}	{b}	{b}	{a}
3.	{b}	{b}	{a,b}	{a}
4.	{a,b}	{a,b}	{a,b}	{a}
5.	{a,b}	{a,b}	{a,b}	{a}

$$\boxed{\begin{array}{l} S' = X' \cup Y' \\ X' = \{b\} \cup S' \\ Y' = Z' \cup \mathbf{X'} \quad \cup Y' \\ Z' = \{a\} \end{array}}$$

- Start from all sets empty.
- Evaluate right-hand side and assign it to left-hand side.
- Repeat until it stabilizes.

Sets grow in each step

- initially they are empty, so they can only grow
- if sets grow, the RHS grows (U is monotonic), and so does LHS
- they cannot grow forever: in the worst case contain all tokens

Constraints for Computing Nullable

- Non-terminal is nullable if it can derive ε

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

$\Longrightarrow$

S' = X' | Y'
X' = 0 | (S' & Y')
Y' = (Z' & X' & 0) | (Y' & 0)
Z' = 1 | 0

S', X', Y', Z' ∈ {0,1}

0	- not nullable
1	- nullable
|	- disjunction
&	- conjunction

	S'	X'	Y'	Z'
1.	0	0	0	0
2.	0	0	0	1
3.	0	0	0	1

again monotonically growing

Computing first and nullable

- Given any grammar we can compute
  - for each non-terminal X whether nullable(X)
  - using this, the set first(X) for each non-terminal X
- General approach:
  - generate constraints over finite domains, following the structure of each rule
  - solve the constraints iteratively
    - start from least elements
    - keep evaluating RHS and re-assigning the value to LHS
    - stop when there is no more change

Summary: Algorithm for **nullable**

```
nullable = {}
changed = true
while (changed) {
  changed = false
  for each non-terminal X
    if ((X is not nullable) and
                (grammar contains rule    X ::= ε | ...    )
           or   (grammar contains rule    X ::= Y1 ... Yn | ...
                where {Y1,...,Yn} ⊆ nullable)
    then {
        nullable = nullable U {X}
        changed = true
     }
}
```

Summary: Algorithm for **first**

```
for each nonterminal X:  first(X)={}
for each terminal t:  first(t)={t}
repeat
  for each grammar rule X ::= Y(1) ... Y(k)
  for i = 1 to k
      if i=1 or {Y(1),...,Y(i-1)} ⊆ nullable then
        first(X) = first(X) U first(Y(i))
until none of first(...) changed in last iteration
```