NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 7: Bottom-up parsing

2020-01-28

Michael Engel

Includes material by Jan Christian Meyer and Rich Maclin (UNM)

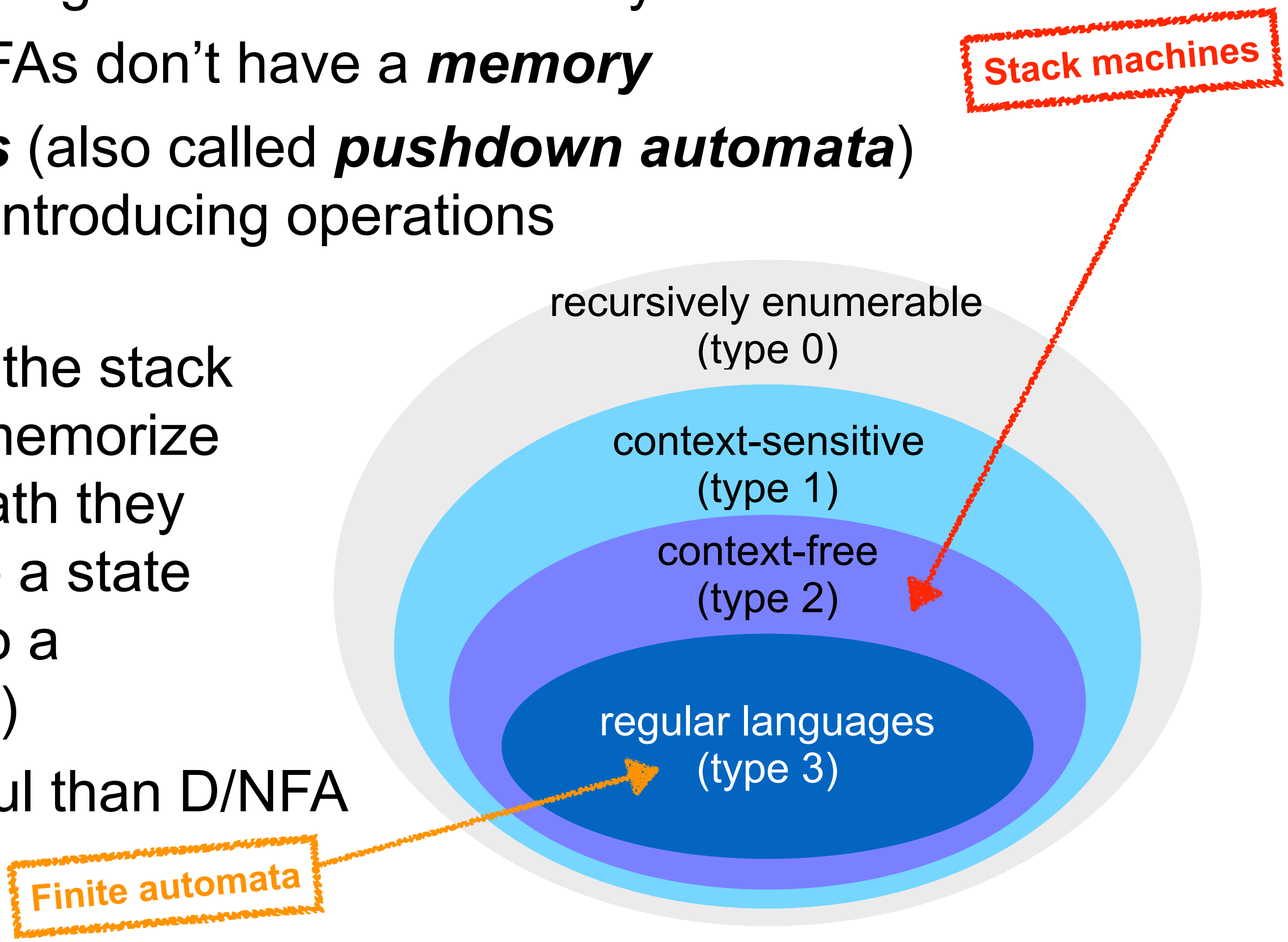
Overview

- Top-down parsing revisited
- Bottom-up parsing
    - Comparison to top-down parsing
    - Shift-reduce parsers
    - Conflict resolution

Types of languages and automata

Syntax analysis

- Context-free languages are a ***superset*** of regular languages
  - Regular languages can be detected by DFAs/NFAs
  - DFAs and NFAs don't have a ***memory***
- ***Stack machines*** (also called ***pushdown automata***) add memory by introducing operations ***push*** and ***pop***
  - They enable the stack machine to memorize (trace) the path they took to get to a state (and revert to a previous one)
  - More powerful than D/NFA



Top-down parsing and the stack

Syntax analysis

- We've seen LL(1) tables and manually built recursive descent parsers
- Another simple example:

$$A \rightarrow \mathbf{x}B \mid \mathbf{y}C$$
$$B \rightarrow \mathbf{x}B \mid \varepsilon$$
$$C \rightarrow \mathbf{y}C \mid \varepsilon$$

	x	y	EOF
A	$A\rightarrow \mathbf{x}B$	$A\rightarrow \mathbf{y}C$	
B	$B\rightarrow \mathbf{x}B$		$B\rightarrow\varepsilon$
C		$C\rightarrow \mathbf{y}C$	$C\rightarrow\varepsilon$

```
void parse_A() {
  switch (sym) {
    case 'x':
      add_tree(x,B);
      match(x);
      parse_B();
      break;
    case 'y':
      add_tree(y,C);
      match(y);
      parse_C();
      break;
    case EOF:
      error();
      break;
  }
  return;
}
```

```
void parse_B() {
  switch (sym):
    case 'x':
      add_tree(x,B);
      match(x);
      parse_B();
      break;
    case 'y':
      error(); break;
    case EOF:
      return;
  return;
}
```

```
void parse_C() {
  switch (sym):
    case 'x':
      error(); break;
    case 'y':
      add_tree(y,C);
      match(y);
      parse_C();
      break;
   case EOF:
      return;
  return;
}
```

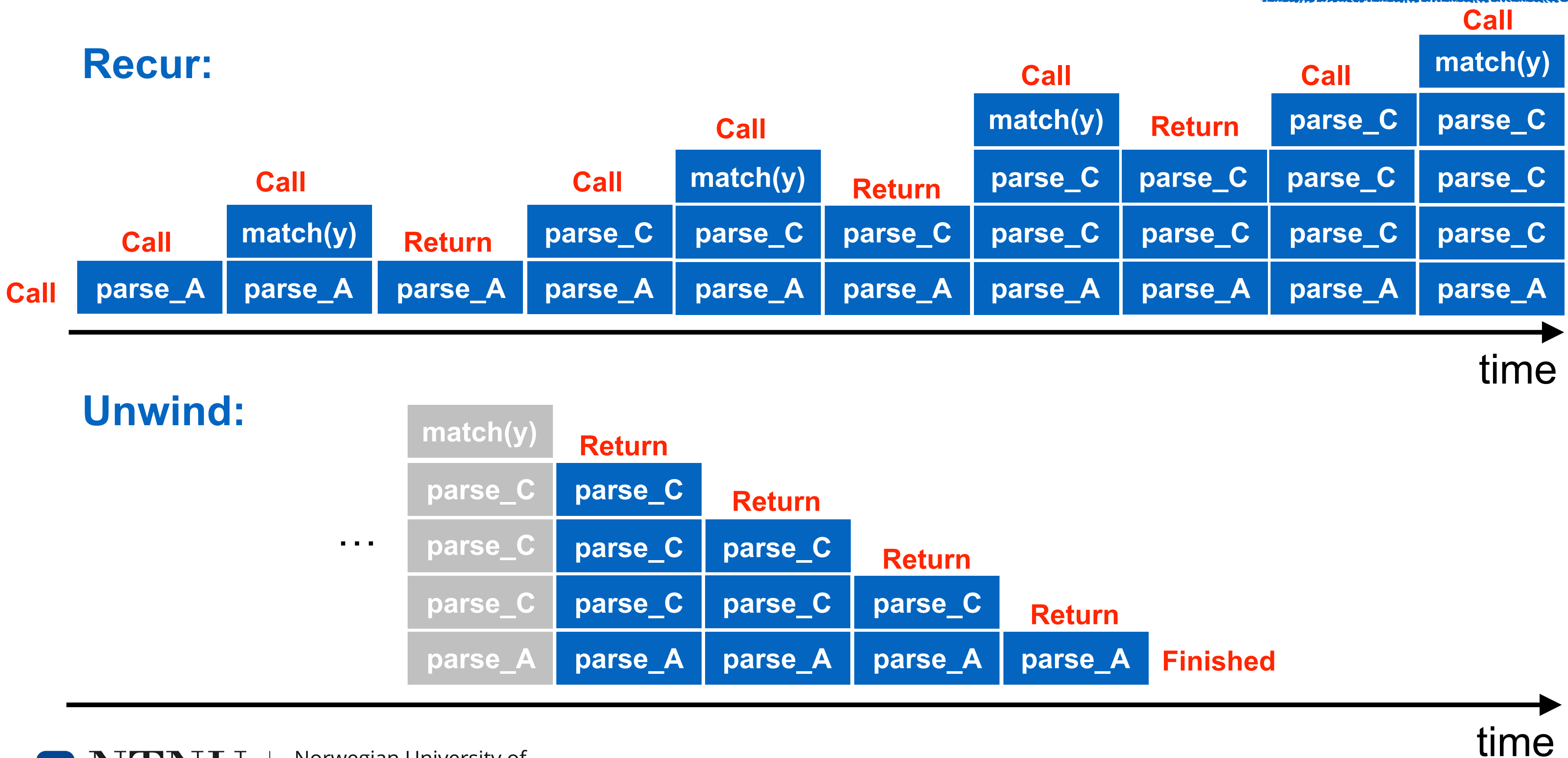
Tracing the recursive descent code

- Which derivation do we get when parsing "y y y"?

  $A \rightarrow yC \rightarrow yyC \rightarrow yyyC \rightarrow yyy$

- What is the related ***hierarchy of function calls***?

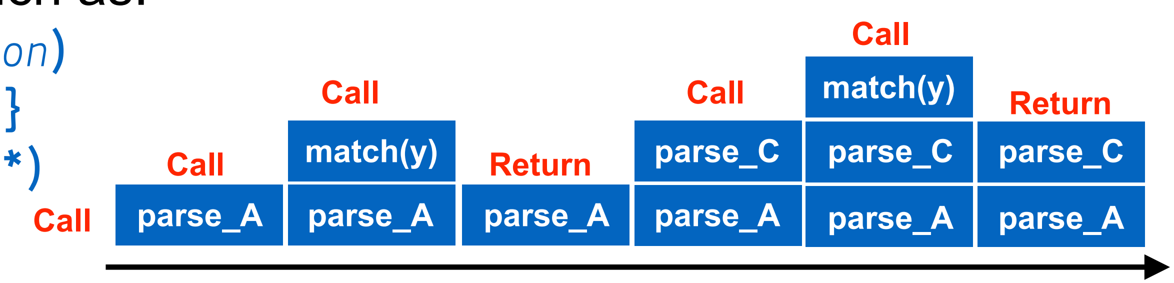
$$A \rightarrow xB \mid yC$$
$$B \rightarrow xB \mid \varepsilon$$
$$C \rightarrow yC \mid \varepsilon$$



Memory in recursive descent code

- Where is the memory hidden in our parser?
  - We do not explicitly store and retrieve state
- The programming language hides it:
  - When calling (returning) from a function, ***state is pushed onto (popped from)*** the computer's stack automatically
  - This state includes the ***return address*** of the call site
- We can also build LL(1) parsers using iterations
  - but then we have to implement our own stack…
- The stack is needed to match beginnings and ends of productions
- Any production of the form *A* → **x***B***y** where *B* can contain further instances of **x** and **y**, such as:

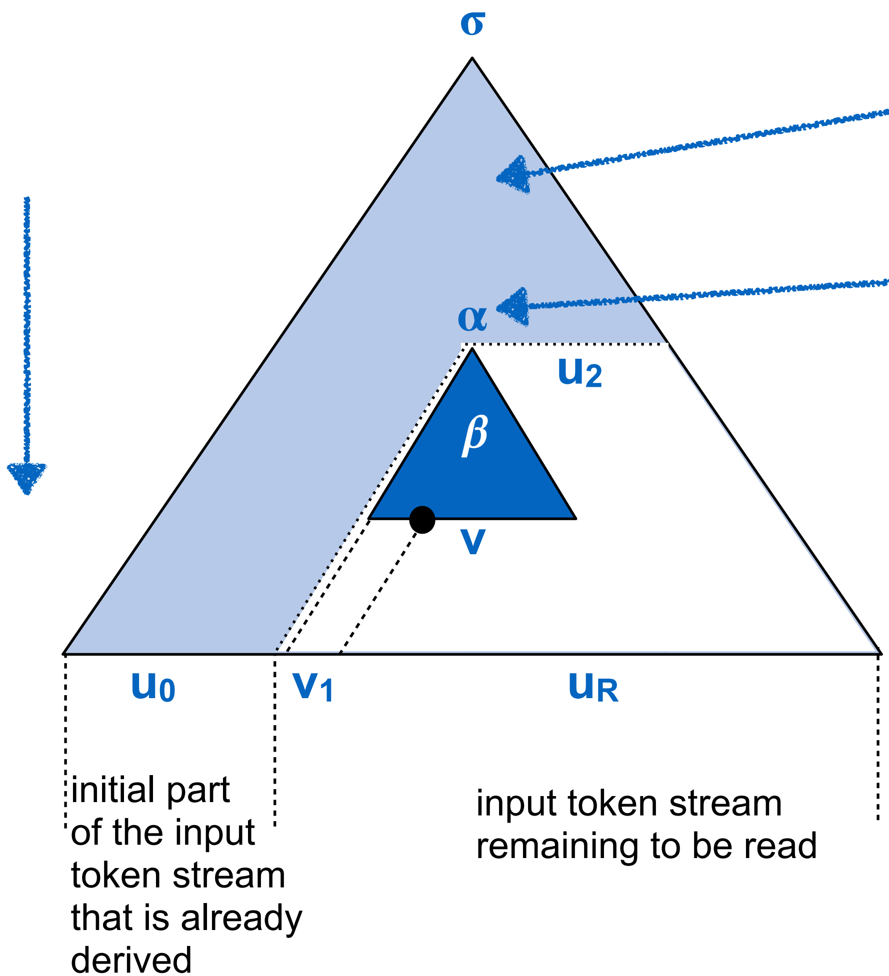
Expression → (*Expression*)
Statement → {*Statement*}
Comment → (* *Comment* *)



Top-down parsing and the syntax tree

Syntax analysis

LL(1) parsers generate a parse tree from top to bottom:



Part of the syntax tree that has already been derived

α: current NT symbol

At this point, the parser tries to find a derivation for **α**:

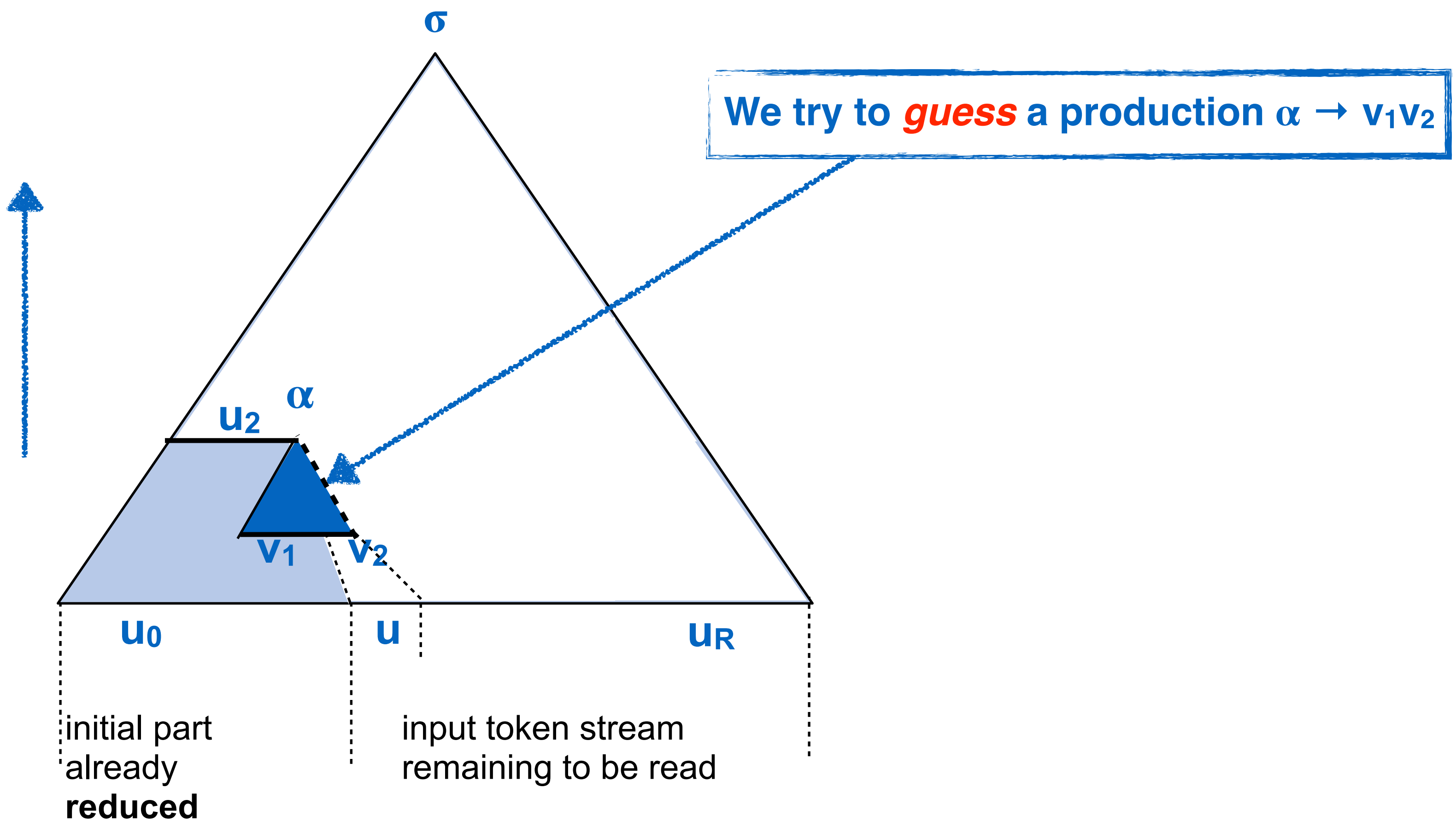
$u_0 \alpha u_2 \rightarrow u_0 v u_2$

u_R has to be derivable from u_2 to complete parsing (otherwise: syntax error)

Bottom-up parsing

Syntax analysis

Can we also construct the parse tree from bottom to top?



General idea of bottom-up parsing

- Bottom-up parsing starts from the input token stream (whereas top-down starts from the grammar start symbol)
- It ***reduces*** a string to the start symbol by ***inverting productions***
  - trying to find a production matching the ***right hand side***

E	→	*T*	+	*E*	|	*T*		
T	→	int	×	*T*	|	int	|	ε

⇒

E	←	*T*	+	*E*	|	*T*		
T	←	int	×	*T*	|	int	|	ε

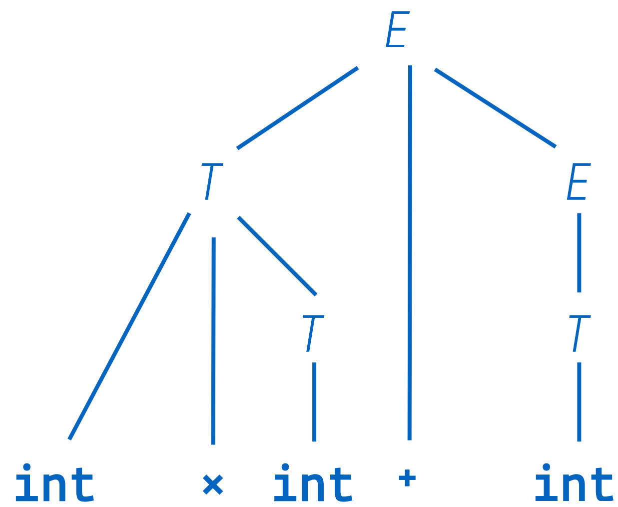
- Consider the input token stream **int * int + int**:
- Reading the productions in reverse (from ***bottom*** to ***top***) gives a ***rightmost derivation***

int × int + int	*T* → int
int × *T* + int	*T* → int × *T*
T + int	*T* → int
T + *T*	*E* → *T*
T + *E*	*E* → *T* + *E*
E	

The resulting parse tree

- A bottom-up parser traces a ***rightmost derivation in reverse***

```
int × int + int
int × T + int
T + int
T + T
T + E
E
```



A simple bottom-up parsing algo

Syntax analysis

- **Idea:** split input string (token stream) into two substrings
  - Right substring (a string of terminal symbols) has not been examined so far
  - Left substring has terminals and nonterminals (generated by **replacing** the right side of a production by the left side)

```
I = input string

repeat
    select a non-empty substring β of I
        where X→β is a production in the grammar
    if no such β exists, backtrack
    replace one β by X in I
until I == "S" /* start symbol */
   or all other possibilities exhausted /* error */
```

Bottom-up parsing steps

Syntax analysis

```
I = input string
repeat
    select a non-empty substring β of I
        where X→β is a production in the grammar
    if no such β exists, backtrack
    replace one β by X in I
until I == "S" /* start symbol */
   or all other possibilities exhausted /* error */
```

- Initially, all input is unexamined, written as:

$\uparrow x_1 x_2 x_3 \ldots x_n$

Two kinds of operations:

- **Shift**: move ↑ one place to the right

ABC↑xyz ➡ ABCx↑yz

- **Reduce**: Apply an inverse production at the right end of the left string
  - If $A \rightarrow xy$ is a production, then

Cbxy↑ijk ➡ CbA↑ijk

Example with reductions only

Syntax analysis

$E \rightarrow T + E \mid T$

$T \rightarrow$ `int` $\times\ T \mid$ `int` $\mid \varepsilon$

int × int ↑ + int	reduce $T \rightarrow$ int
int × T ↑ + int	reduce $T \rightarrow$ int × T
T + int ↑	reduce $T \rightarrow$ int
T + T ↑	reduce $E \rightarrow T$
T + E ↑	reduce $E \rightarrow T + E$

Example with shift-reduce parsing

$E \rightarrow T + E \mid T$
$T \rightarrow \texttt{int} \times T \mid \texttt{int} \mid \varepsilon$

↑ int × int + int	shift
int ↑ × int + int	shift
int × ↑ int + int	shift
int × int ↑ + int	reduce $T \rightarrow$ int
int × T ↑ + int	reduce $T \rightarrow$ int × T
T ↑ + int	shift
T + ↑ int	shift
T + int ↑	reduce $T \rightarrow$ int
T + T ↑	reduce $E \rightarrow T$
T + E ↑	reduce $E \rightarrow T + E$
E	(arrived at start symbol!)

Implementing the memory

Syntax analysis

$E \rightarrow T + E \mid T$
$T \rightarrow \mathtt{int} \times T \mid \mathtt{int} \mid \varepsilon$

Idea:

- Left substring can be implemented by a ***stack***
  - ***shift*** operating pushes a ***terminal*** symbol onto the stack
  - ***reduce*** pops zero or more symbols off the stack (the right-hand side of a production) and ***pushes a non-terminal symbol*** onto the stack (left-hand side of a production)

stack contents	input token stream	parser operation: stack operation(s)
[]	↑ int × int + int	shift: push [int]
[int]	int ↑ × int + int	shift: push [×]
[int, ×]	int × ↑ int + int	shift: push [int]
[int, ×, int]	int × int ↑ + int	reduce *T* → int: pop->int, push[*T*]
[int, ×, *T*]	int × int ↑ + int	reduce *T* → int × *T*: pop, push[*T*]
[*T*]	int × int ↑ + int	…

Conflicts in parsing

Problem:

- How do we decide when to shift or reduce?
  - Consider the step `int ↑ × int + int`
  - We could reduce using *T* → `int` giving *T* ↑ `× int + int`
  - A fatal mistake: **No way to reduce to the start symbol E**
- Generic shift-reduce strategy:
  - If there is a matching pattern (***handle***) *on* the stack, reduce
  - Otherwise, shift
- What if there is a choice (between two matching patterns)?
  - If it's legal to shift or reduce, there is a ***shift-reduce conflict***
  - If it is legal to reduce by two different productions, there is a ***reduce-reduce conflict***

Source of conflicts and example

Conflicts arise due to:

- Ambiguous grammars: always cause conflicts
- But beware, so do many non-ambiguous grammars

- Conflict example

Grammar

$$E \to E + E \mid E \times E \mid (E) \mid \texttt{int}$$

↑ int × int + int	shift
…	
$E \times E$ ↑ + int	reduce $E \to$ int $\times E$
E ↑ + int	shift
$E +$ ↑ int	shift
$E +$ int ↑	reduce $E \to$ int
$E + E$ ↑	reduce $E \to E + E$
E ↑	

Source of conflicts and example

↑ int × int + int	shift
…	
E × E ↑ + int	reduce E → int × E
E ↑ + int	shift
E + ↑ int	shift
E + int ↑	reduce E → int
E + E ↑	reduce E → E + E
E ↑	

E → E + E
| E × E
| (E)
| int

Another derivation is also possible:

↑ int × int + int	shift
…	
E × E ↑ + int	shift
E × E + ↑ int	shift
E × E + int ↑	reduce E → int
E × E + E ↑	reduce E → E + E
E × E ↑	reduce E → E × E
E ↑	

We can decide to either shift or reduce in this step

The choice whether to shift or reduce determines the associativity of + and ×!

Resolving conflicts: precedence

Syntax analysis

$E \rightarrow E + E$
$\mid E \times E$
$\mid (E)$
$\mid$ int

The choice whether to shift or reduce determines the associativity of + and ×

- We could rewrite the grammar to enforce precedence (as seen with top-down parsing)
- Alternative: provide ***precedence declarations***
  - these cause shift-reduce parsers to resolve conflicts in certain ways
  - Declaring "× has greater precedence than +" causes parser to reduce at $E \times E \uparrow + \texttt{int}$
  - More precisely, precedence declaration is used to resolve conflict between reducing a × and shifting a +

The term "precedence declaration" is misleading. These declarations do not define precedence; they define ***conflict resolutions***

What now?

Syntax analysis

- Our key ingredients for bottom-up parsing:
  - a stack to shift and reduce symbols on
  - an automaton that can use stacked history to backtrack its footsteps

- The LR(k) family of languages can all be parser using a shift-reduce parser like this
- The complexity of the grammars you can handle is related to how elaborate your automaton is
  - several variants: SLR, LALR, LR(1)
  - Let’s start with a simple one, LR(0), in the next lecture