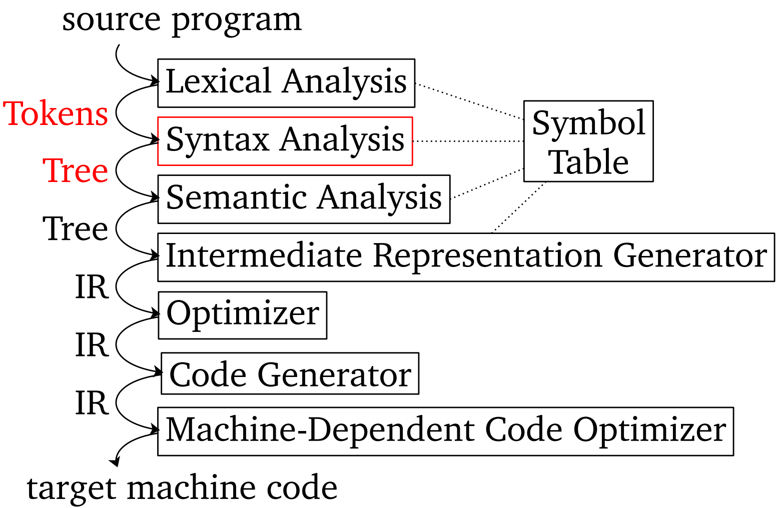
3. Top-Down Syntax Analysis

Eva Rose Kristoffer Rose

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)
http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-3.pdf

September 18, 2014

Second compilation phase



Example

```
position = initial + rate * 60
```

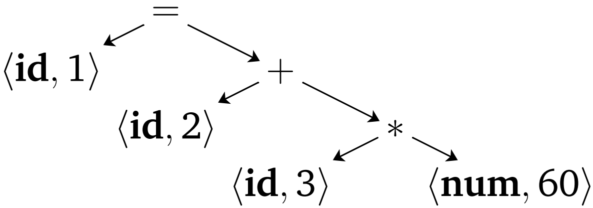
scanned into list of *tokens*:

$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle \mathbf{num}, 60\rangle$

1	`position`
2	`initial`
3	`rate`

Example

parsed into *syntax tree*:



according to precedence rules...

Example

Some tree-structuring syntax rules:

- block indentation rules (Python)
- block delimiters like {...} (Java, C)
- grouping rules like (...) (most languages)
- built-in algebraic precedence rules (most languages)
- statements vs expressions ...

1. Context
2. **Syntax Definitions**
3. Parsers
4. Context-Free Grammars
5. Top-Down Parsing
   - FIRST and FOLLOW
   - Predictive Parsing
6. HACS *as a* Parser Generator

How do we specify language syntax?

- Context-free grammar
- special notation (BNF)
- Set of rules (productions)

Example:

```
if (x=2) print("yep"); else print("nope");
```

Corresponds to a rule:

$$\mathit{stm} \rightarrow \textbf{if}\ (\mathit{expr})\ \mathit{stm}\ \textbf{else}\ \mathit{stm}$$

How do we specify language syntax?

- Context-free grammar
- special notation (BNF)
- Set of rules (productions)

Example:

```
if (x=2) print("yep"); else print("nope");
```

Corresponds to a rule:

$$stm \;\rightarrow\; \textbf{if}\ (expr)\ stm\ \textbf{else}\ stm$$

Production rules

$$\mathit{stm} \quad \rightarrow \quad \underbrace{\textbf{if}\ (\mathit{expr})\ \mathit{stm}\ \textbf{else}\ \mathit{stm}}$$

production head

production body

"can have the form"

Production rules

$$\mathit{stm} \rightarrow \mathbf{if}\ (\mathit{expr})\ \mathit{stm}\ \mathbf{else}\ \mathit{stm}$$

Nonterminals need more rules to define them.

Terminals are well defined, no more rules define them.

Components of context-free grammar.

1. Set of terminal symbols.
2. Set of nonterminal symbols.
3. Set of productions.
    - The head is non-terminal.
    - The body is a sequence of terminals and non-terminals.
4. Designation of one nonterminal as the starting symbol.

Example

$$
\begin{aligned}
\mathit{list} &\rightarrow \mathit{list} + \mathit{digit} \\
\mathit{list} &\rightarrow \mathit{list} - \mathit{digit} \\
\mathit{list} &\rightarrow \mathit{digit} \\
\mathit{digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

1. What are the terminals here?
2. What are the nonterminals?
3. What does the grammar generate?

Derivations

Derivation algorithm:

- given the grammar (production rules),
- begin with the start symbols,
- repeatedly replace nonterminals (head) with their bodies,
- the generated set of terminals defines the language of that grammar.

Example: How do we *derive* the string $9 - 5 + 7$?

Derivations

$$\begin{aligned} \mathit{list} &\to \mathit{list} + \mathit{digit} \mid \mathit{list} - \mathit{digit} \mid \mathit{digit} \\ \mathit{digit} &\to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

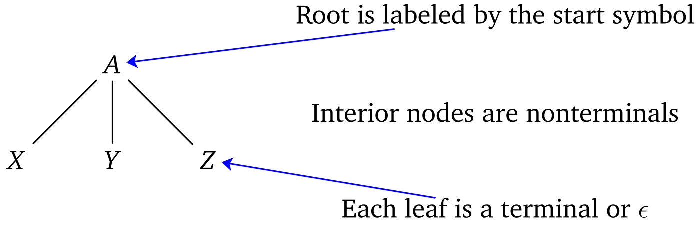
Deriving the string $9 - 5 + 7$:

$$\begin{aligned} \mathit{list} &\Rightarrow \mathit{list} + \mathit{digit} \Rightarrow \mathit{list} - \mathit{digit} + \mathit{digit} \Rightarrow \mathit{digit} - \mathit{digit} + \mathit{digit} \\ &\Rightarrow 9 - \mathit{digit} + \mathit{digit} \Rightarrow 9 - 5 + \mathit{digit} \Rightarrow 9 - 5 + 7 \end{aligned}$$

Parsing

Given a string of terminals, figure out how to picture a tree from the start symbol of the grammar where all the terminals are at the leaves.

$A \to XYZ$



The process of finding such a “grammar” tree is called parsing. The “grammar” tree is called a parse tree.

Exercise

$$list \rightarrow list + digit \mid list - digit \mid digit$$
$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Derive a parse tree for the string $9 - 5 + 7$.

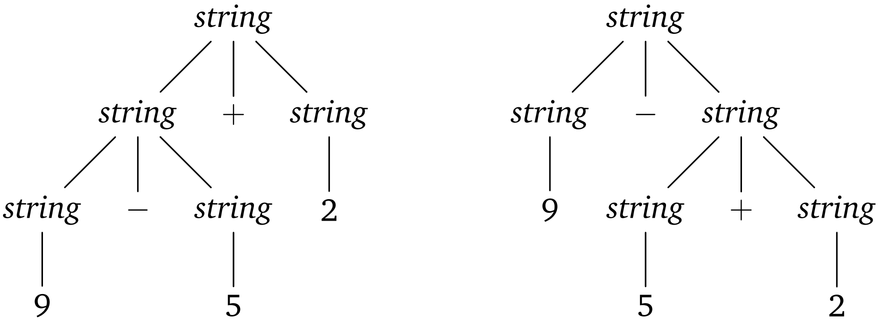
Example

$$string \to string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

What about a parse tree for the string $9 - 5 + 7$ with this grammar?

Ambiguity

A grammar is ambiguous if it has more than one parse tree generating the same string of terminals.



Two parse trees for $9 - 5 + 2$. Which is right?

Examples

Are any of the following grammars ambiguous?

1. $S \rightarrow +\ S\ S \ \mid\ -\ S\ S \ \mid\ a$
2. $S \rightarrow S\ (\ S\ )\ S \ \mid\ \epsilon$

Associativity of operators

How will you evaluate $9-5-2$?

- If 5 goes with the '–' on the left: $(9-5)-2$ we say the operator is left associative.
- If 5 goes with the '–' on the right: $9-(5-2)$ we say the operator is right associative.

Associativity of operators

How do we express associativity in production rules?

- Left associative $(9-5)-2$:

$$term \to term - digit \mid digit$$
$$digit \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- Right associative $9-(5-2)$:

$$term \to digit - term \mid digit$$
$$digit \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Precedence of operators

What if operators are different, e.g., $9 - 5 * 2$?

- If '$*$' takes operands before '$-$', it is said to have higher precedence.
- Another example: logical operators...

Precedence of operators

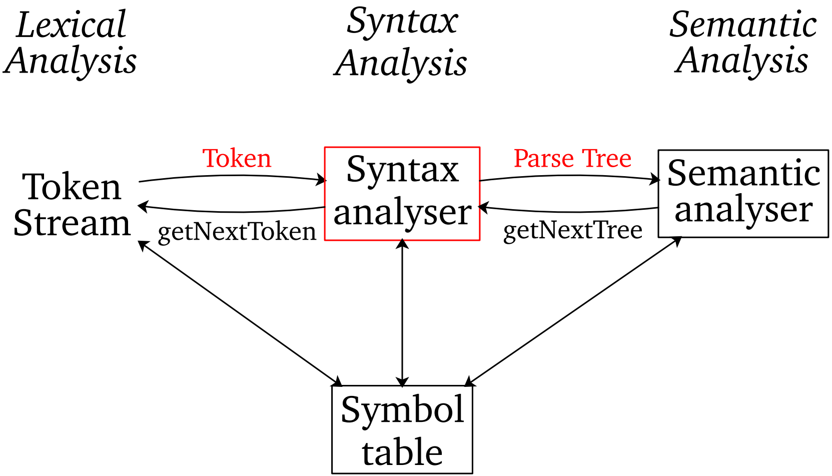
How to present precedence in productions?

$$expr \to expr + term \mid expr - term \mid term$$

$$term \to term * factor \mid term / factor \mid factor$$

$$factor \to \mathbf{digit} \mid (expr)$$

The example shows both operator precedence ('$*$', '$/$' over '$+$', '$-$') and left associativity.


Lexical Analysis
Syntax Analysis
Semantic Analysis
Token
Parse Tree
Token Stream
Syntax analyser
Semantic analyser
getNextToken
getNextTree
Symbol table

Parsing

The parsing task:

- We have: set of grammar productions.
- We have: string of terminals for the grammar.
- We need: find a parse tree that generates the string.

Example: parsing

Given these grammar productions:

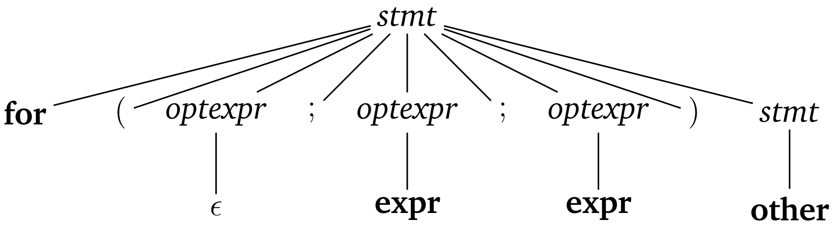
$$
\begin{aligned}
\mathit{stmt} &\rightarrow \mathbf{expr} \\
&\mid \mathbf{if}\,(\mathbf{expr})\,\mathit{stmt} \\
&\mid \mathbf{for}\,(\mathit{optexpr}\,;\mathit{optexpr}\,;\mathit{optexpr})\,\mathit{stmt} \\
&\mid \mathbf{other} \\
\mathit{optexpr} &\rightarrow \mathbf{expr} \mid \epsilon
\end{aligned}
$$

and this string:

$$\mathbf{for}\,(\epsilon\,;\mathbf{expr}\,;\mathbf{expr})\,\mathbf{other}$$

Example: parsing

How do we generate this parse tree in an operational manner?



Construction of a parse tree

Three general kinds of parsers:

Universal from any grammar, but too inefficient.

Bottom-up identifying the string symbols as terminals, constructing the tree from leaves to root.

Top-down beginning with the start symbol as the root, constructing the tree from root to leaves.

Definition

Given an input string top-down parsing is defined as follows:

- Start naming the root with the starting (nonterminal) symbol.
- Repeat whilst scanning the input sting, one symbol at a time:
  1. At node N labeled with nonterminal A: *select a production for A* and construct children at N with the symbols in the production body.
  2. Find the next unexpanded node/nonterminal, typically the leftmost unexpanded nonterminal in the tree

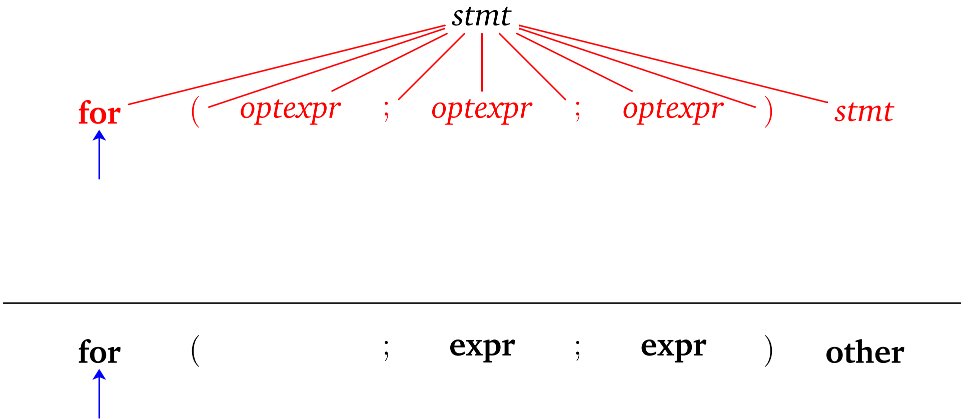
"*select a production for A*" is guided by the current input symbol.

Example

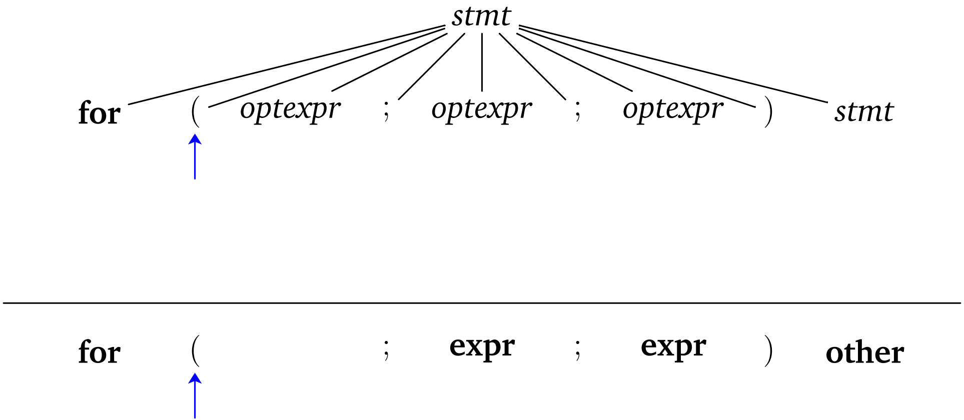
stmt

for (; **expr** ; **expr**) **other**

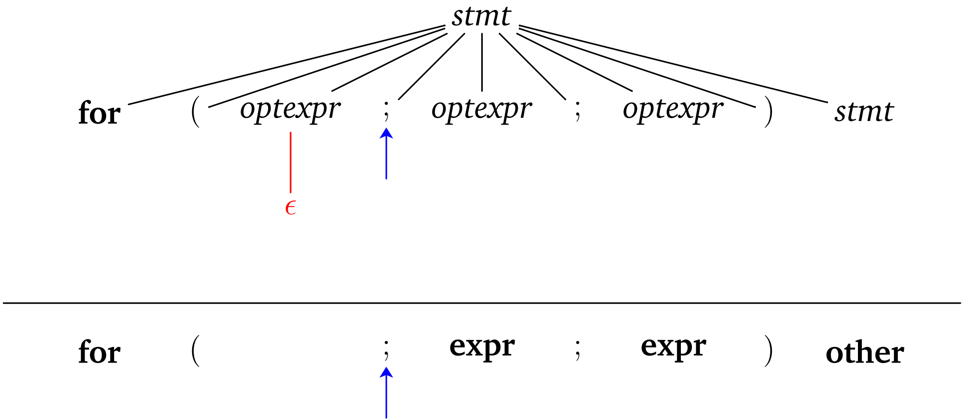
Example



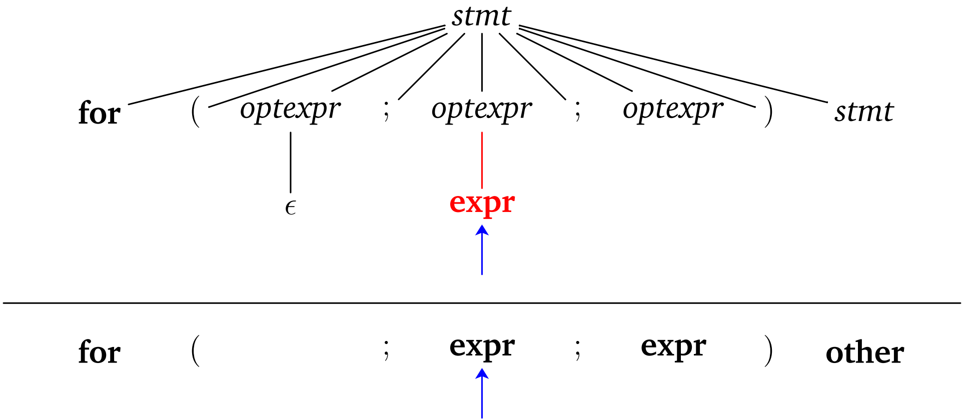
Example



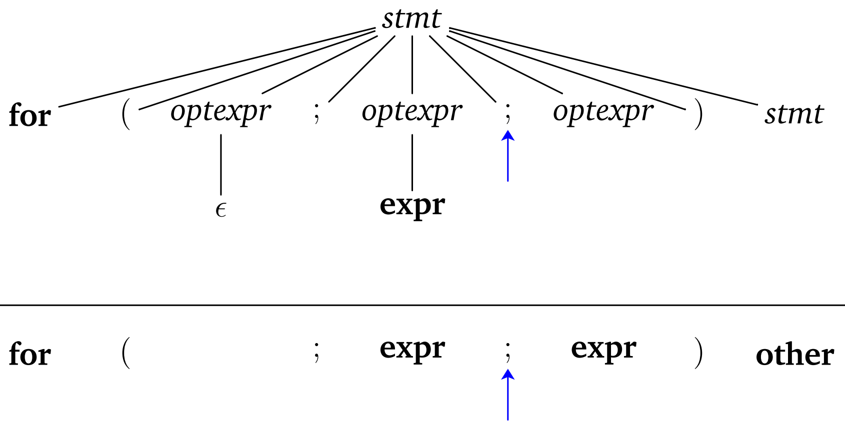
Example



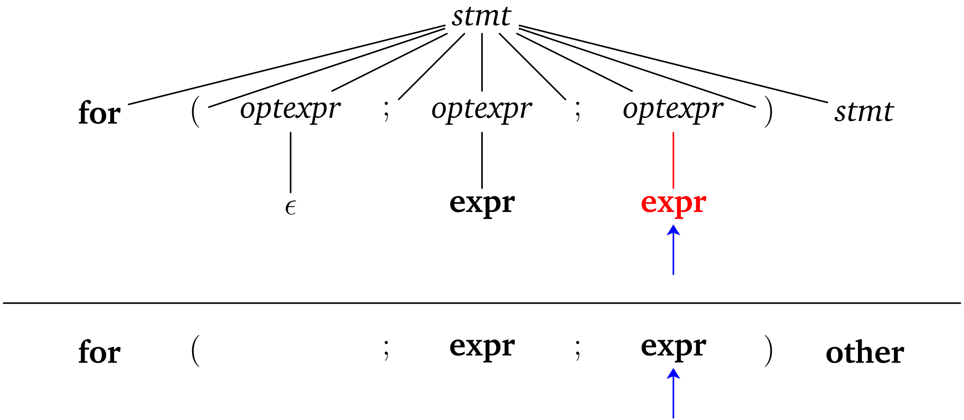
Example


stmt
for
(
optexpr
;
optexpr
;
optexpr
)
stmt
ϵ
expr
for
(
;
expr
;
expr
)
other

Example

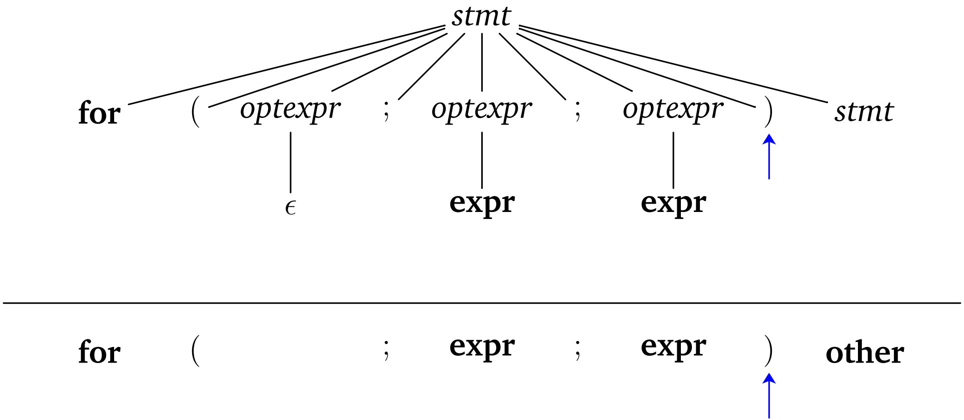

stmt
for
(
optexpr
;
optexpr
;
optexpr
)
stmt
ϵ
expr
for
(
;
expr
;
expr
)
other

Example

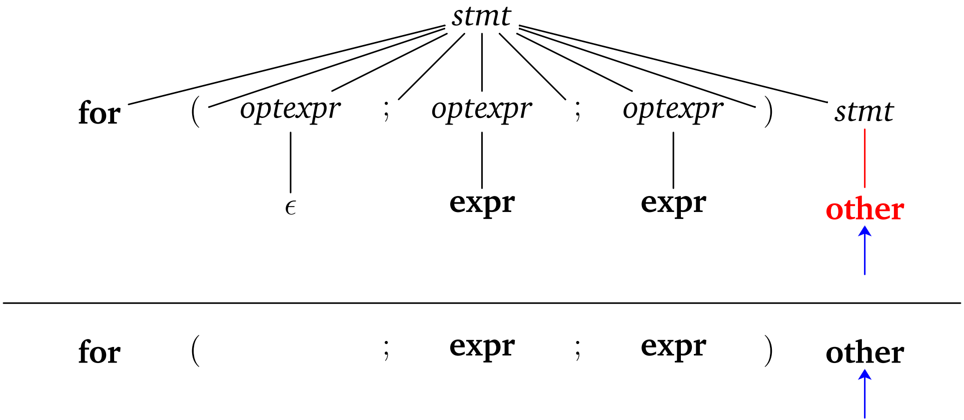


stmt
for (optexpr ; optexpr ; optexpr) stmt
ϵ
expr
expr
for (; expr ; expr) other

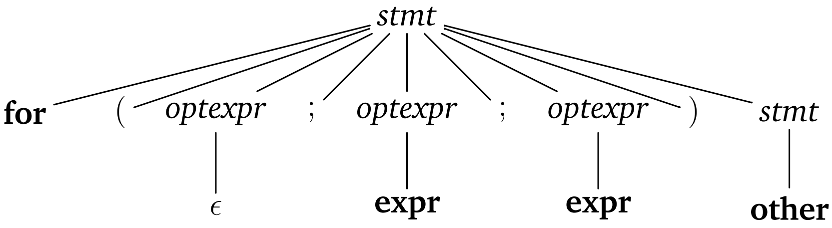
Example


stmt
for
(
optexpr
;
optexpr
;
optexpr
)
stmt
ϵ
expr
expr
for (; expr ; expr) other

Example


stmt
for
(
optexpr
;
optexpr
;
optexpr
)
stmt
ϵ
expr
expr
other
for (; expr ; expr) other

Example



for (; **expr** ; **expr**) **other**

Parsing

Sometimes choosing the right production involves trial and error, whence backtracking!

Parsing

Predictive parsing: NO backtracking!

- Part of a category called recursive-descent parsing are top-down methods, based on recursive procedures.
- The lookahead symbol unambiguously determines the flow of control.

Parsing

Designing a predictive parser.

- A procedure is designed for every nonterminal/production.
- Examination of the lookahead symbol chooses a production.
- No conflict between two bodies with the same head may occur.
- The procedure mimics the body of the chosen production:
  - nonterminals are procedure calls,
  - terminals are matched and lookahead is advanced.

Left recursion: a problem!

Left recursive grammar:

$$expr \rightarrow expr + term \mid term$$

Eliminating left recursion:

$$\begin{aligned} expr &\rightarrow term\ factor \\ factor &\rightarrow +\ term\ factor \mid \epsilon \end{aligned}$$

Both generating: $term,\ term{+}term,\ term{+}term{+}term,\ \ldots$

Context Free Grammars, revisited

Start symbol, Nonterminals, Terminals, Productions

$$
\begin{aligned}
\mathit{expression} &\rightarrow \mathit{expression} + \mathit{term}\\
\mathit{expression} &\rightarrow \mathit{expression} - \mathit{term}\\
\mathit{expression} &\rightarrow \mathit{term}\\
\mathit{term} &\rightarrow \mathit{term} * \mathit{factor}\\
\mathit{term} &\rightarrow \mathit{term} \,/\, \mathit{factor}\\
\mathit{term} &\rightarrow \mathit{factor}\\
\mathit{factor} &\rightarrow (\mathit{expression})\\
\mathit{factor} &\rightarrow \mathbf{id}
\end{aligned}
$$

Derivations, revisited

- Start with start symbol.
- Each step: a nonterminal is replaced with the body of a production.

Example:

$$E \to E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

Deriving **–(id + id)**

$$E \Rightarrow -E \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Derivations, revisited

$\Rightarrow$ means derive in one step.

$\overset{*}{\Rightarrow}$ means derive in zero or more steps.

$\overset{+}{\Rightarrow}$ means derive in one or more steps.

1. $\alpha \overset{*}{\Rightarrow} \alpha$ for any string α (reflective).
2. $\alpha \overset{*}{\Rightarrow} \beta$, and $\beta \overset{*}{\Rightarrow} \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$ (transitive).
3. Same properties for $\overset{+}{\Rightarrow}$.

Derivations, revisited

- **Leftmost derivations:** the leftmost nonterminal in each sentential is always chosen. $\alpha \Rightarrow_{lm} \beta$
- **Rightmost derivations:** the rightmost nonterminal in each sentential is always chosen. $\alpha \Rightarrow_{rm} \beta$

Example

$$S \to SS+ \quad | \quad SS* \quad | \quad \mathbf{a}$$

and the string **aa+a***

1. Give leftmost derivation for the string.
2. Give rightmost derivation for the string.
3. Give parse tree for the string.

Parse trees and derivations

Relationship:

- parse trees are graphical representations of derivations,
- parse trees filter out the order of nonterminal replacements,
- many-to-one relationship between derivations and parse trees.

Example

$$E \to E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

The string $-(\mathbf{id} + \mathbf{id})$ has two derivations:

- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$
- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$

...but one parse tree!

Context-free grammars vs regular expressions

- Grammars are more powerful notations than regular expressions.

Every construct that can be described by a regular expression can be described by a grammar, but not vice versa.

Example:

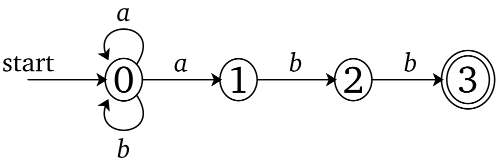
$$\{a^n b^n \mid n \geq 1\}$$

From RE to NFA

A regular expression:

$$(a|b)^*abb$$

as NFA:



From NFA to context-free grammar

Now apply the conversion algorithm:

1. For each state i of NFA, create nonterminal A_i.
2. If state i has transition to state j on input a, add the production $A_i \rightarrow aA_j$. If transition happens on ϵ, add the production $A_i \rightarrow A_j$.
3. If i is an accepting state, add $A_i \rightarrow \epsilon$.
4. If i is start state, make A_i the start symbol of the grammar.

which renders:

$$
\begin{aligned}
A_0 &\rightarrow \mathbf{a}A_0 \mid \mathbf{b}A_0 \mid aA_1 \\
A_1 &\rightarrow \mathbf{b}A_2 \\
A_2 &\rightarrow \mathbf{b}A_3 \\
A_3 &\rightarrow \epsilon
\end{aligned}
$$

Relevant question

If grammars are so much more powerful than regular expressions, why not use them during lexical analysis?

- Lexical (token) descriptions are quite simple patterns.
- REs easier to understand for simple patterns.
- Easier to generate an efficient lexical analyser from a simple REs.

How can we enhance our grammar?

1. Eliminate ambiguity.
2. Eliminate left-recursion.
3. Left factoring.

Eliminating ambiguity

Sometimes we can re-write the grammar to eliminate ambiguity.

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\ |\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ |\ & \textbf{other} \end{aligned} \tag{4.14}$$

consider the parse trees for:

$$\textbf{if}\ expr\ \textbf{then}\ \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$$

How can we fix it?

Eliminating ambiguity

$$
\begin{aligned}
\mathit{stmt} &\to \mathit{matched\_stmt} \\
&\mid \mathit{open\_stmt} \\[1em]
\mathit{matched\_stmt} &\to \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{matched\_stmt} \\
&\mid \textbf{other} \\[1em]
\mathit{open\_stmt} &\to \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
&\mid \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{open\_stmt}
\end{aligned}
$$

Eliminating Left-Recursion

$A \to A\alpha \mid \beta$ can be rewritten:
$$\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \epsilon \end{aligned}$$

$A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ can be rewritten:

$$\begin{aligned} A &\to \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

Example: Eliminating Left-Recursion

How about something like:

$$\begin{aligned} S &\to A\,a \mid b \\ A &\to A\,c \mid S\,d \mid \epsilon \end{aligned} \tag{4.18}$$

Second line can be rewritten: $A \to A\,c \mid A\,a\,d \mid b\,d \mid \epsilon$

Resulting in:

$$\begin{aligned} S &\to A\,a \mid b \\ A &\to b\,d\,A' \mid A' \\ A' &\to c\,A' \mid a\,d\,A' \mid \epsilon \end{aligned}$$

Left Factoring

A way to delay the decision of what production rule to expand by.

General rule: $A \to \alpha\, \beta_1 \mid \alpha\, \beta_2$ rewritten to: $A \to \alpha\, A'$, $A' \to \beta_1 \mid \beta_2$

Example:

$$\begin{aligned} \mathit{stmt} &\to \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ &\mid \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \end{aligned}$$

rewritten to:

$$\begin{aligned} \mathit{stmt} &\to \mathit{EXP}\ \textbf{else}\ \mathit{stmt} \mid \mathit{EXP} \\ \mathit{EXP} &\to \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \end{aligned}$$

Top-Down Parsing

Once grammar is on its optimal form:

1. Constructing a parse tree, for an input string, starting at the root:
   - parse tree build in preorder (depth-first).
2. Finding a left-most derivation.
3. At each step of the top-down parse:
   - determine which production to be applied,
   - matching terminal symbols in the production body with the input string symbols.

Example

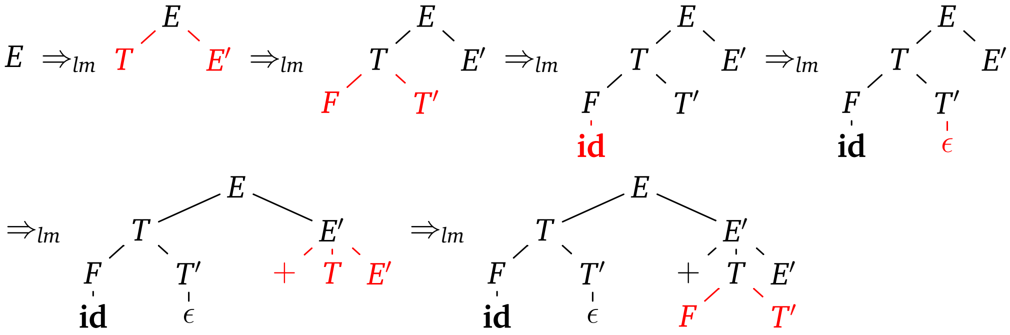
$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (\ E\ ) \mid \mathbf{id}
\end{aligned}
\tag{4.1}
$$

Left-recursion elimination:

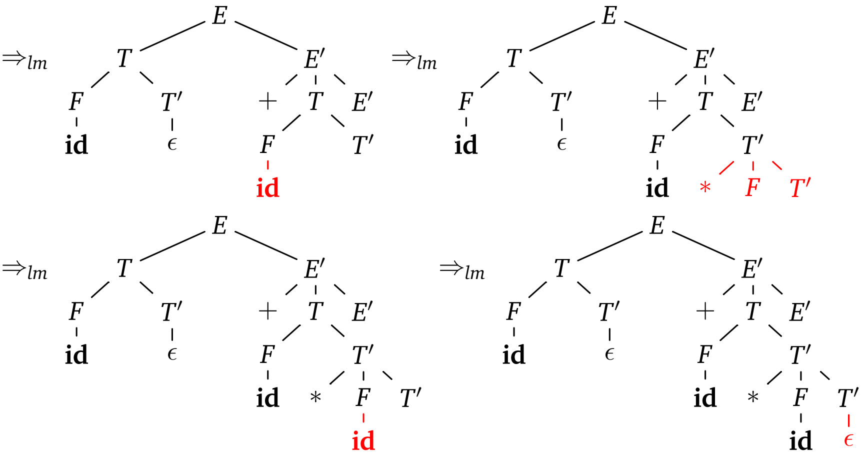
$$
\begin{aligned}
E &\to T\ E' \\
E' &\to \quad + T\ E' \mid \epsilon \\
T &\to F\ T' \\
T' &\to \quad * F\ T' \mid \epsilon \\
F &\to (\ E\ ) \mid \mathbf{id}
\end{aligned}
\tag{4.2}
$$

Example: Top-Down Parsing

Consider the string: **id** $+$ **id** $*$ **id**

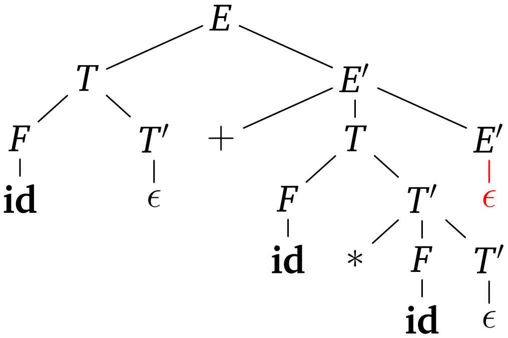


Example: Top-Down Parsing



Example: Top-Down Parsing

$\Rightarrow_{lm}$



Top-Down parser categories

Recursive-Descent Parser general form of top-down parsers.

- May require backtracking.

Predictive Parser special case of recursive-descent parsers.

- No backtracking necessary.
- Constructed from LL(k) grammars, $k \geq 1$.

An LL(k) grammar implies scanning input from Left, retrieving Leftmost derivation, using k lookahead symbols to reach a resolution.

Choosing the right production

Key parsing problem: determining the production to be applied for some nonterminal A.

FIRST and FOLLOW helps choosing the production based on the next input symbol.

Definition: FIRST and FOLLOW

Definition

Define $\text{FIRST}(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in $\text{FIRST}(\alpha)$.

Definition

Define $\text{FOLLOW}(A)$, for non-terminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$ for some α and β.

Algorithm: FIRST Set for a Grammar

Repeat for each production until a fixed point (with stable sets) is reached:

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$
2. If X is a nonterminal and $X \to Y_1 \dots Y_k$, $k \geq 1$, $a \in \text{FIRST}(X)$ if $a \in \text{FIRST}(Y_i)$, where ϵ is in all of $\text{FIRST}(Y_1) \dots \text{FIRST}(Y_{i-1})$; that is, $Y_1 \dots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If $\epsilon \in \text{FIRST}(Y_j)$, for all $j = 1, \dots, k, k \geq 1$, then add ϵ to $\text{FIRST}(X)$.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

Algorithm: FOLLOW Set for a Grammar

Repeat for each production until a fixed point (with stable sets) is reached:

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example

	FIRST	FOLLOW
$E \to T\ E'$	(**id**	) \$
$E' \to\ +\ T\ E' \mid \epsilon$	$+\ \ \epsilon$	) \$
$T \to F\ T'$	(**id**	$+$) \$
$T' \to\ *\ F\ T' \mid \epsilon$	$*\ \ \epsilon$	$+$) \$
$F \to (\ E\ ) \mid \mathbf{id}$	(**id**	$*\ \ +$) \$

LL(1)—Left (to right) Leftmost (derivation) with 1 (input symbol)

Three requirements for productions $A \to \alpha \mid \beta$:

1. $\textsc{first}(\alpha)$ and $\textsc{first}(\beta)$ are disjoint (including for ϵ).
2. If $\epsilon \in \textsc{first}(\alpha)$ then $\textsc{first}(\beta)$ and $\textsc{follow}(A)$ are disjoint.
3. If $\epsilon \in \textsc{first}(\beta)$ then $\textsc{first}(\alpha)$ and $\textsc{follow}(A)$ are disjoint.

Example (why is left recursion a problem?)

	FIRST	FOLLOW
$E \to E + T \mid T$	($\mathbf{id}$	$+$) \$
$T \to T * F \mid F$	($\mathbf{id}$	$+$ $*$) \$
$F \to (E) \mid \mathbf{id}$	($\mathbf{id}$	$+$ $*$) \$

We cannot decide how to parse from the first symbol!

Example (why is left recursion a problem?)

	FIRST	FOLLOW
$E \to E + T \mid T$	(**id**	+) \$
$T \to T * F \mid F$	(**id**	+ *) \$
$F \to (E) \mid \mathbf{id}$	(**id**	+ *) \$

We cannot decide how to parse from the first symbol!

Predictive Parsing Table

Input: LL(1) grammar G.

Output: Parsing table M.

Method: For each production $A \to \alpha$ in the grammar:

1. For each $a \in \text{FIRST}(\alpha)$ add $A \to \alpha$ to $M[A, a]$.
2. If $\epsilon \in \text{FIRST}(\alpha)$ then for each $b \in \text{FOLLOW}(A)$ add $A \to \alpha$ to $M[A, b]$. (If $\epsilon \in \text{FIRST}(\alpha)$ and $\$ \in \text{FOLLOW}(A)$ then add $A \to \alpha$ to $M[A, \$]$.)

Any entries $M[A, a]$ that have no content are set to **error**.

Parser Table

Non-terminal	Input Symbol					
	$\mathbf{id}$	$+$	$*$	$($	$)$	$\$$
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	$F \to \mathbf{id}$			$F \to (E)$		

Input	Stack	Output

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E\,\$$	

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E\,\$$	
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	$E \to TE'$

Parser Table

Non-terminal	Input Symbol					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

Input	Stack	Output
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E\,\$$	
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	$E \to TE'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$

Parser Table

Non-terminal	Input Symbol					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

Input	Stack	Output
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	$E \to TE'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$

Parser Table

NON-TERMINAL	INPUT SYMBOL id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \rightarrow FT'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \rightarrow \mathbf{id}$
$+\ \mathbf{id} * \mathbf{id}\$$	$T'\,E'\,\$$	

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\, T'\, E'\, \$$	$F \to \mathbf{id}$
$+\ \mathbf{id} * \mathbf{id}\$$	$T'\, E'\, \$$	
$+\ \mathbf{id} * \mathbf{id}\$$	$T'\, E'\, \$$	$T' \to \epsilon$

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$+\ \mathbf{id} * \mathbf{id}\$$	$T'\ E'\ \$$	
$+\ \mathbf{id} * \mathbf{id}\$$	$T'\ E'\ \$$	$T' \to \epsilon$
$+\ \mathbf{id} * \mathbf{id}\$$	$E'\ \$$	$E' \to +TE'$

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$+\ \mathbf{id} * \mathbf{id}\$$	$T'\,E'\,\$$	$T' \to \epsilon$
$+\ \mathbf{id} * \mathbf{id}\$$	$E'\,\$$	$E' \to +TE'$
$+\ \mathbf{id} * \mathbf{id}\$$	$+\,T\,E'\,\$$	

Parser Table

Non-terminal	Input Symbol					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

Input	Stack	Output
$+\ \mathbf{id} * \mathbf{id}\$$	$E'\,\$$	$E' \to +TE'$
$+\ \mathbf{id} * \mathbf{id}\$$	$+\,T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$+\ \mathbf{id} * \mathbf{id}\$$	$+\,T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$
$\mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$
$\mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$* \;\mathbf{id}\$$	$T'\,E'\,\$$	

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\, T'\, E'\, \$$	$F \to \mathbf{id}$
$* \ \mathbf{id}\$$	$T'\, E'\, \$$	
$* \ \mathbf{id}\$$	$*\, F\, T'\, E'\, \$$	$T' \to *FT'$

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$*\ \mathbf{id}\$$	$T'\ E'\ \$$	
$*\ \mathbf{id}\$$	$*\ F\ T'\ E'\ \$$	$T' \to *FT'$
$\mathbf{id}\$$	$F\ T'\ E'\ \$$	

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	$($	$)$	$\$$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$*\ \mathbf{id}\$$	$*\,F\,T'\,E'\,\$$	$T' \to *FT'$
$\mathbf{id}\$$	$F\,T'\,E'\,\$$	
$\mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id}\$$	$F\,T'\,E'\,\$$	
$\mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$\$$	$T'\,E'\,\$$	$T' \to \epsilon$

Parser Table

NON-TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

INPUT	STACK	OUTPUT
$\mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$\$$	$T'\,E'\,\$$	$T' \to \epsilon$
$\$$	$E'\,\$$	$E' \to \epsilon$

Parser Table

Non-terminal	Input Symbol					
	id	$+$	$*$	(	)	\$
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	$F \to \mathbf{id}$			$F \to (E)$		

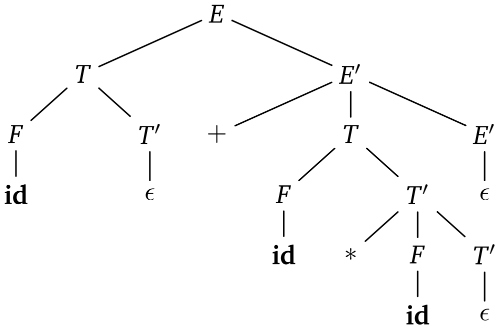
Input	Stack	Output
\$	$T'\,E'\,\$$	$T' \to \epsilon$
\$	$E'\,\$$	$E' \to \epsilon$
\$	\$	

Example: Top-Down Predictive Parsing

INPUT	STACK	OUTPUT
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E\,\$$	
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	$E \to TE'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$+ \mathbf{id} * \mathbf{id}\$$	$T'\,E'\,\$$	
$+ \mathbf{id} * \mathbf{id}\$$	$T'\,E'\,\$$	$T' \to \epsilon$
$+ \mathbf{id} * \mathbf{id}\$$	$E'\,\$$	$E' \to +TE'$
$+ \mathbf{id} * \mathbf{id}\$$	$+\,T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$
$\mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$* \mathbf{id}\$$	$T'\,E'\,\$$	
$* \mathbf{id}\$$	$*\,F\,T'\,E'\,\$$	$T' \to *FT'$
$\mathbf{id}\$$	$F\,T'\,E'\,\$$	
$\mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$\$$	$T'\,E'\,\$$	$T' \to \epsilon$
$\$$	$E'\,\$$	$E' \to \epsilon$
$\$$	$\$$	

Example: Top-Down Predictive Parsing

INPUT	STACK	OUTPUT
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E\,\$$	
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	$E \to TE'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$
$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$+ \mathbf{id} * \mathbf{id}\$$	$T'\,E'\,\$$	
$+ \mathbf{id} * \mathbf{id}\$$	$T'\,E'\,\$$	$T' \to \epsilon$
$+ \mathbf{id} * \mathbf{id}\$$	$E'\,\$$	$E' \to +TE'$
$+ \mathbf{id} * \mathbf{id}\$$	$+\,T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$T\,E'\,\$$	
$\mathbf{id} * \mathbf{id}\$$	$F\,T'\,E'\,\$$	$T \to FT'$
$\mathbf{id} * \mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$* \mathbf{id}\$$	$T'\,E'\,\$$	
$* \mathbf{id}\$$	$*\,F\,T'\,E'\,\$$	$T' \to *FT'$
$\mathbf{id}\$$	$F\,T'\,E'\,\$$	
$\mathbf{id}\$$	$\mathbf{id}\,T'\,E'\,\$$	$F \to \mathbf{id}$
$\$$	$T'\,E'\,\$$	$T' \to \epsilon$
$\$$	$E'\,\$$	$E' \to \epsilon$
$\$$	$\$$	



Errors

- Lexical.
- Syntactic.
- Semantic.
- Logical.

Detect early? Recover? Avoid repetition?

- Panic! (for a while, "skip to ;")
- Phrase-level.
- Error Productions.
- Global Correction.

Errors

- Lexical.
- Syntactic.
- Semantic.
- Logical.

Detect early? Recover? Avoid repetition?

- Panic! (for a while, "skip to ;")
- Phrase-level.
- Error Productions.
- Global Correction.

Errors

- Lexical.
- Syntactic.
- Semantic.
- Logical.

Detect early? Recover? Avoid repetition?

- Panic! (for a while, “skip to ;”)
- Phrase-level.
- Error Productions.
- Global Correction.

Errors

- Lexical.
- Syntactic.
- Semantic.
- Logical.

Detect early? Recover? Avoid repetition?

- Panic! (for a while, "skip to ;")
- Phrase-level.
- Error Productions.
- Global Correction.

Errors

- Lexical.
- Syntactic.
- Semantic.
- Logical.

Detect early? Recover? Avoid repetition?

- Panic! (for a while, “skip to ;”)
- Phrase-level.
- Error Productions.
- Global Correction.

HACS Encoding

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (\ E \ ) \mid \textbf{id}
\end{aligned}
\tag{4.1}
$$

sort E | ⟦⟨E⟩ + ⟨T⟩⟧ | ⟦⟨T⟩⟧ ;
sort T | ⟦⟨T⟩ * ⟨F⟩⟧ | ⟦⟨F⟩⟧ ;
sort F | ⟦ (⟨E⟩) ⟧ | ⟦⟨Id⟩⟧ ;

HACS Encoding

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (\ E\ ) \mid \mathbf{id}
\end{aligned}
\tag{4.1}
$$

$$\text{sort E} \mid [\![\langle\text{E}\rangle + \langle\text{T}\rangle]\!] \mid [\![\langle\text{T}\rangle]\!] ;$$
$$\text{sort T} \mid [\![\langle\text{T}\rangle * \langle\text{F}\rangle]\!] \mid [\![\langle\text{F}\rangle]\!] ;$$
$$\text{sort F} \mid [\![\ (\ \langle\text{E}\rangle\ )\ ]\!] \mid [\![\langle\text{Id}\rangle]\!] ;$$

HACS Encoding Details

```
sort E | ⟦⟨E⟩ + ⟨T⟩⟧ | ⟦⟨T⟩⟧ ;
sort T | ⟦⟨T⟩ * ⟨F⟩⟧ | ⟦⟨F⟩⟧ ;
sort F | ⟦ ( ⟨E⟩ ) ⟧ | ⟦⟨Id⟩⟧ ;
```

- Every choice (= production) introduced by "|"
- Explicit "⟦⟧" for concrete syntax
- Explicit "⟨⟩" for nonterminal and terminal references
- sort means kind of syntax tree node
- Direct left recursion is permitted

HACS Encoding Details

$$\text{sort } E \mid [\![\langle E\rangle + \langle T\rangle]\!] \mid [\![\langle T\rangle]\!] \;;$$
$$\text{sort } T \mid [\![\langle T\rangle * \langle F\rangle]\!] \mid [\![\langle F\rangle]\!] \;;$$
$$\text{sort } F \mid [\![\ (\ \langle E\rangle\ )\ ]\!] \mid [\![\langle Id\rangle]\!] \;;$$

- Every choice (= production) introduced by "|"
- Explicit "$[\![\,]\!]$" for concrete syntax
- Explicit "$\langle\,\rangle$" for nonterminal and terminal references
- sort means kind of syntax tree node
- Direct left recursion is permitted

HACS Encoding with Precedence & Associativity

$$\mathbf{sort}\ \mathrm{E} \mid [\![\langle \mathrm{E}\rangle + \langle \mathrm{T}\rangle]\!] \mid [\![\langle \mathrm{T}\rangle]\!]\ ;$$
$$\mathbf{sort}\ \mathrm{T} \mid [\![\langle \mathrm{T}\rangle * \langle \mathrm{F}\rangle]\!] \mid [\![\langle \mathrm{F}\rangle]\!]\ ;$$
$$\mathbf{sort}\ \mathrm{F} \mid [\![\ (\ \langle \mathrm{E}\rangle\ )\ ]\!] \mid [\![\langle \mathrm{Id}\rangle]\!]\ ;$$

$$\begin{aligned}\mathbf{sort}\ \mathrm{E}\ &\mid [\![\langle \mathrm{E@1}\rangle + \langle \mathrm{E@2}\rangle]\!]\mathrm{@1} \\ &\mid [\![\langle \mathrm{E@2}\rangle * \langle \mathrm{E@3}\rangle]\!]\mathrm{@2} \\ &\mid \mathbf{sugar}\ [\![(\langle \mathrm{E\#1@1}\rangle)]\!]\mathrm{@3} \to \mathrm{E\#1} \mid [\![\langle \mathrm{Id}\rangle]\!]\mathrm{@3}\ ;\end{aligned}$$

HACS Encoding with Precedence & Associativity

$$\textbf{sort } E \mid [\![\langle E\rangle + \langle T\rangle]\!] \mid [\![\langle T\rangle]\!] \;;$$
$$\textbf{sort } T \mid [\![\langle T\rangle * \langle F\rangle]\!] \mid [\![\langle F\rangle]\!] \;;$$
$$\textbf{sort } F \mid [\![\; (\; \langle E\rangle \;) \;]\!] \mid [\![\langle Id\rangle]\!] \;;$$

$$\begin{array}{l} \textbf{sort } E \mid [\![\langle E@1\rangle + \langle E@2\rangle]\!]@1 \\ \qquad\quad \mid [\![\langle E@2\rangle * \langle E@3\rangle]\!]@2 \\ \qquad\quad \mid \textbf{sugar } [\![(\langle E\#1@1\rangle)]\!]@3 \rightarrow E\#1 \mid [\![\langle Id\rangle]\!]@3 \;; \end{array}$$

HACS Encoding with Precedence & Associativity Details

$$
\begin{aligned}
\textbf{sort}\ \mathrm{E}\ & |\ [\![\langle \mathrm{E@1}\rangle + \langle \mathrm{E@2}\rangle]\!]\mathrm{@1} \\
& |\ [\![\langle \mathrm{E@2}\rangle * \langle \mathrm{E@3}\rangle]\!]\mathrm{@2} \\
& |\ \textbf{sugar}\ [\![(\langle \mathrm{E\#1@1}\rangle)]\!]\mathrm{@3} \to \mathrm{E\#1}\ |\ [\![\langle \mathrm{Id}\rangle]\!]\mathrm{@3}\ ;
\end{aligned}
$$

- Single sort captures abstract syntax tree (AST)
- Precedence marker "@"n for precedence and associativity.
- sugar specifies concrete-only syntax.

HACS Encoding with Precedence & Associativity Details

$$
\begin{array}{lll}
\text{sort } \mathrm{E} & | & [\![\langle \mathrm{E@1}\rangle + \langle \mathrm{E@2}\rangle]\!]\mathrm{@1} \\
 & | & [\![\langle \mathrm{E@2}\rangle * \langle \mathrm{E@3}\rangle]\!]\mathrm{@2} \\
 & | & \text{sugar } [\![(\langle \mathrm{E\#1@1}\rangle)]\!]\mathrm{@3} \rightarrow \mathrm{E\#1} \;|\; [\![\langle \mathrm{Id}\rangle]\!]\mathrm{@3}\ ;
\end{array}
$$

- Single sort captures abstract syntax tree (AST)
- Precedence marker "@"n for precedence and associativity.
- sugar specifies concrete-only syntax.

HACS Expression Parser

```
sort Stat | ⟦ ⟨Name⟩ := ⟨Exp⟩ ; ⟧| ⟦{ ⟨Stat∗⟩ } ⟧;

sort Exp  | ⟦ ⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1
          | ⟦ ⟨Exp@2⟩ ∗ ⟨Exp@3⟩ ⟧@2
          | ⟦ ⟨Int⟩ ⟧@3
          | ⟦ ⟨Float⟩ ⟧@3
          | ⟦ ⟨Name⟩ ⟧@3
          | sugar ⟦(⟨Exp#⟩) ⟧@3 →Exp# ;

sort Name | symbol ⟦⟨Id⟩ ⟧;
```

HACS Summary

- Unusual brackets $\llbracket\rrbracket\langle\rangle\ldots$
- Automatically does immediate left recursion elimination.
- sort covers multiple non-terminals.
- Precedence & associativity handled automatically (the @ markers).
- sugar.
- (symbol for names – that's next week.)

Oh, and it is just a front-end for JavaCC and the CRSX rewrite engine.

HACS Summary

- Unusual brackets ⟦⟧⟨⟩. . .
- Automatically does immediate left recursion elimination.
- sort covers multiple non-terminals.
- Precedence & associativity handled automatically (the @ markers).
- sugar.
- (symbol for names – that's next week.)

Oh, and it is just a front-end for JavaCC and the CRSX rewrite engine.

Project Milestone 1

Project Milestone 1 Released!

Due 10/6 (Monday) 8am

Questions?

evarose@cs.nyu.edu krisrose@cs.nyu.edu