RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers
Syntax Analysis
Top-Down Parsing

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Review: Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Parsing Techniques: Top-down parsers

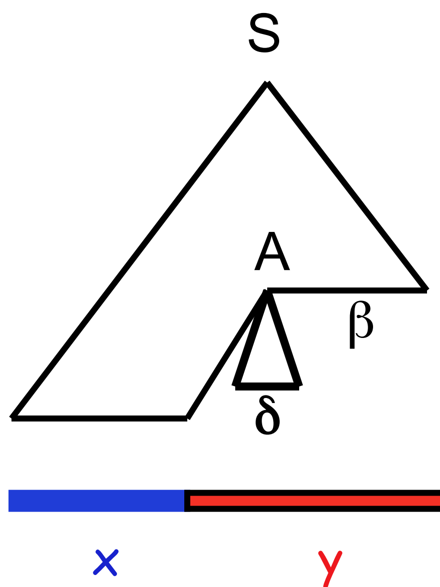
LL(1), recursive descent

1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

$$S \Rightarrow^*_{lm} x\, A\, \beta \;\Rightarrow_{lm} x\, \delta\, \beta \;\Rightarrow^*_{lm} x\, y$$



Parsing Techniques: Top-down parsers

LL(1), recursive descent

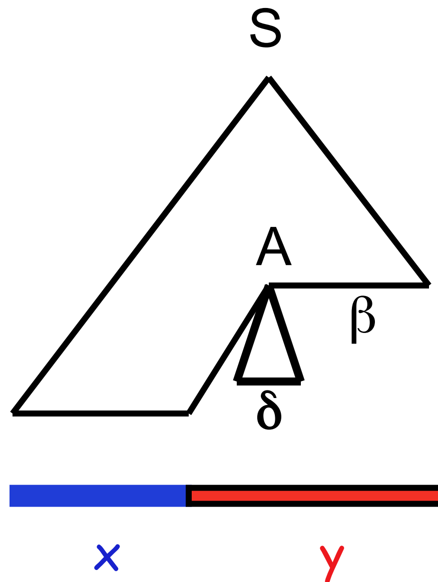
1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

rule $A \rightarrow \delta$

$$S \Rightarrow^*_{lm} x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow^*_{lm} x\, y$$



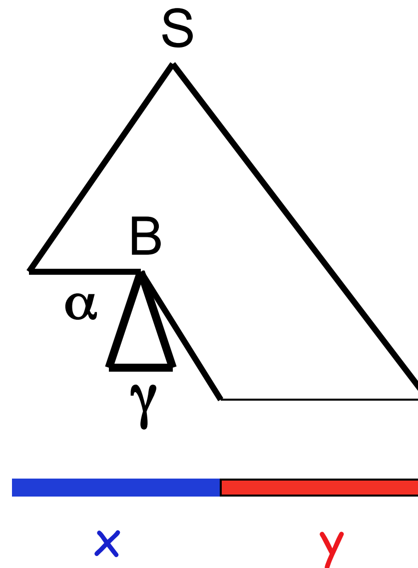
LR(1), operator precedence

1 input symbol lookahead

construct rightmost deriviation (backwards)

input: read left-to-right

$$S \Rightarrow^*_{rm} \alpha\ B\ y \Rightarrow_{rm} \alpha\ \gamma\ y \Rightarrow^*_{rm} x\ y$$



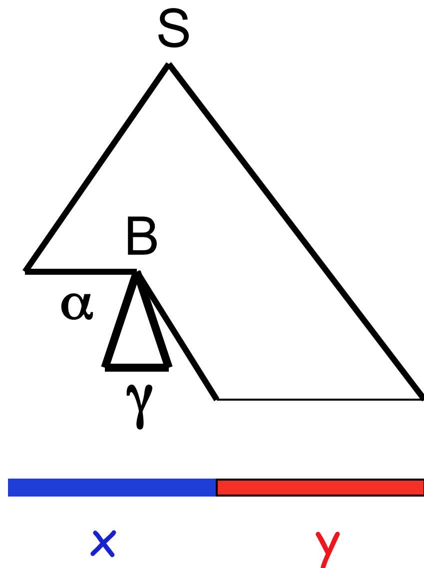
Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

- 1 input symbol lookahead
- construct rightmost deriviation (backwards)
- input: read left-to-right

rule $B \rightarrow \gamma$

$$S \Rightarrow^*_{rm} \alpha B y \Rightarrow_{rm} \alpha \gamma y \Rightarrow^*_{rm} x\ y$$



Generic Top-down Parsing

A top-down parser starts with the root of the parse tree
The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until the fringe of the parse tree matches the input string

1. At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
2. When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack
3. Find the next node to be expanded (label $\in$ NT)

- The key is picking the right production in step 1
  - → That choice should be guided by the input string

Remember the expression grammar?

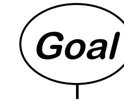
Version with precedence

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	<u>number</u>
9		|	<u>id</u>

And the input <u>x</u> - <u>2</u> * <u>y</u>

Example

GUESS



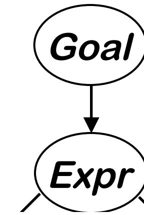
Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y

Leftmost derivation, choose productions in an order that exposes problems

Example

GUESS



Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y

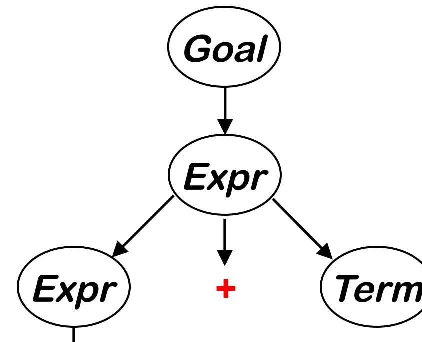
Leftmost derivation, choose productions in an order that exposes problems

Example

GUESS

Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y



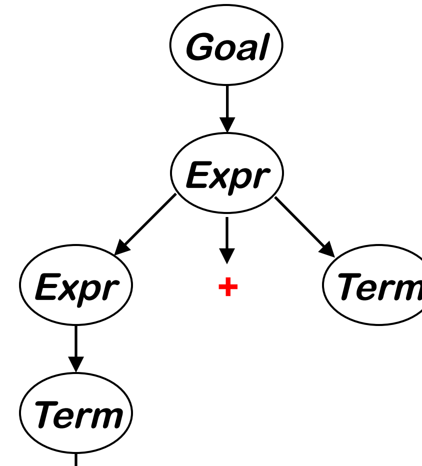
Leftmost derivation, choose productions in an order that exposes problems

Example

Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y

GUESS



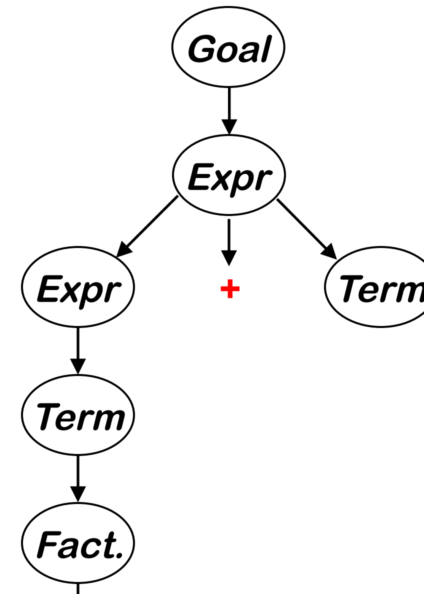
Leftmost derivation, choose productions in an order that exposes problems

Example

Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y

GUESS



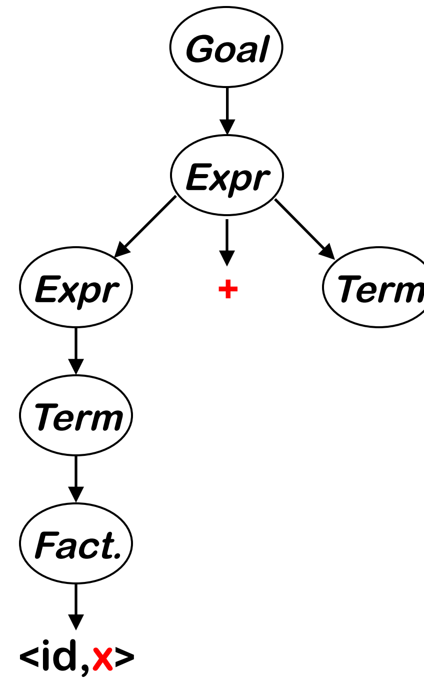
Leftmost derivation, choose productions in an order that exposes problems

Example

Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
4	*Term* + *Term*	↑x - 2 * y
7	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
9	<id,x> + *Term*	x ↑- 2 * y

GUESS



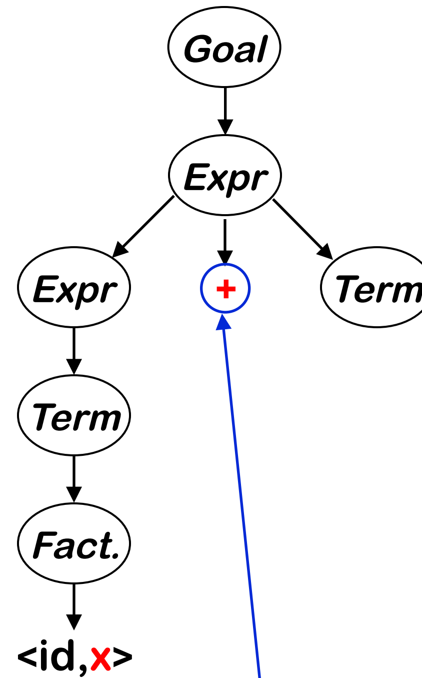
Leftmost derivation, choose productions in an order that exposes problems

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
1	Expr	↑x - 2 * y
2	Expr + Term	↑x - 2 * y
4	Term + Term	↑x - 2 * y
7	Factor + Term	↑x - 2 * y
9	<id,x> + Term	↑x - 2 * y
9	<id,x> + Term	x ↑- 2 * y

GUESS



This worked well, except that "–" doesn't match "+"

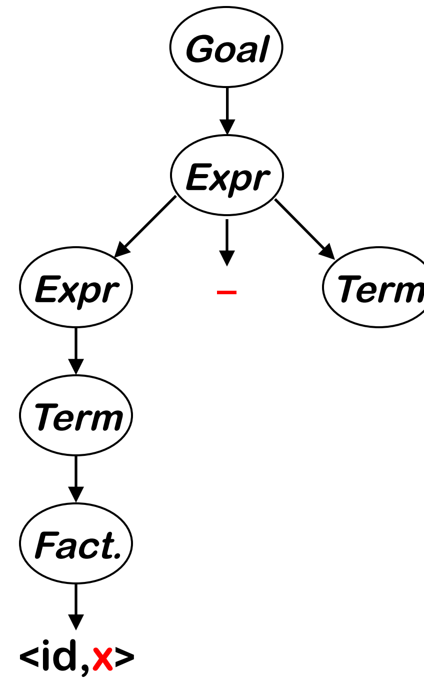
The parser must backtrack to here

Example

Continuing with x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr – Term*	↑x – 2 * y
4	*Term – Term*	↑x – 2 * y
7	*Factor – Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y

GUESS

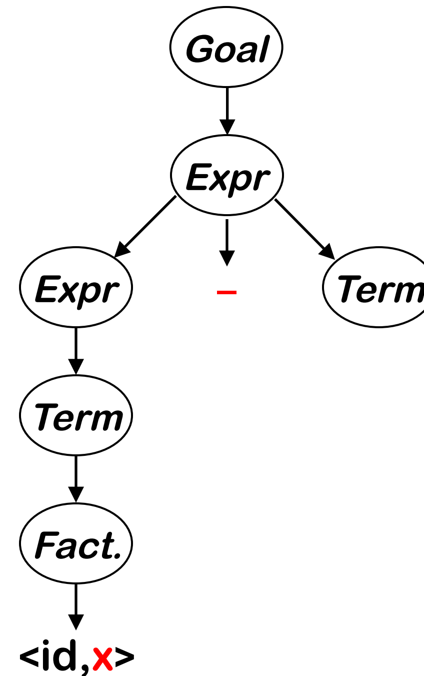


Example

Continuing with x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr – Term*	↑x – 2 * y
4	*Term – Term*	↑x – 2 * y
7	*Factor – Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y

GUESS



This time, "–" and "–" matched

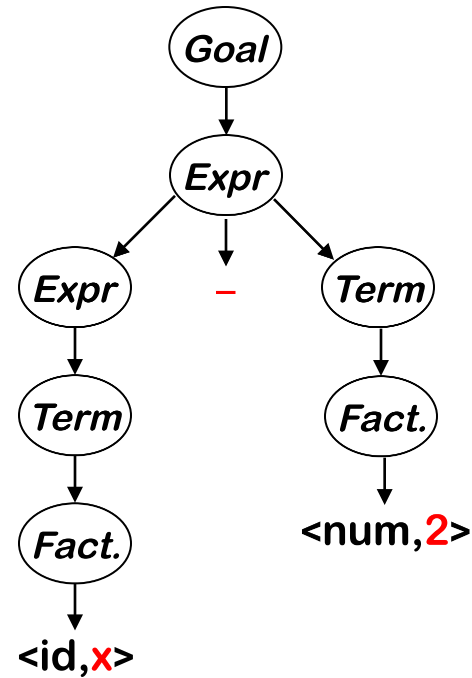
We can advance past "–" to look at "2"

⇒ Now, we need to expand *Term* - the last *NT* on the fringe

Example

Trying to match the "2" in x - 2 * y :

Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2 ↑* y



RUTGERS

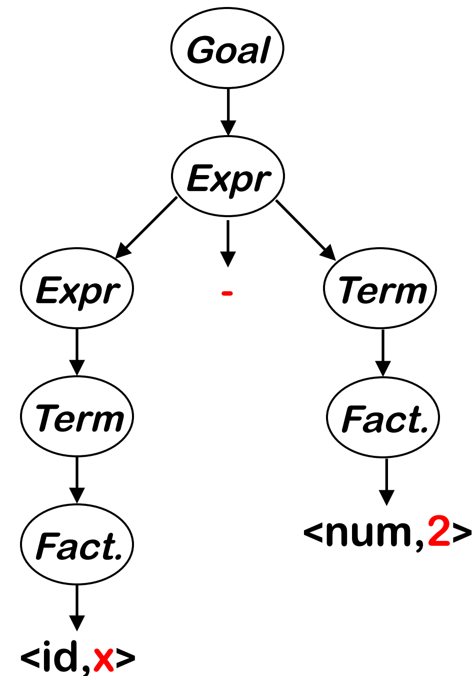
Example

GUESS

Trying to match the "2" in x - 2 * y :

Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2 ↑* y

Where are we?



- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

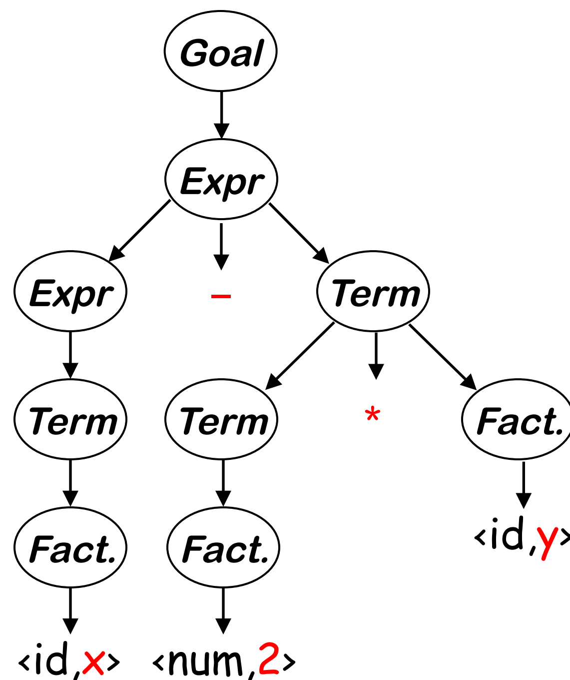
⇒ Need to backtrack

Example

GUESS : SUCESS

Trying again with "2" in x - 2 * y :

Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
5	<id,x> - *Term* * *Factor*	x - ↑2 * y
7	<id,x> - *Factor* * *Factor*	x - ↑2 * y
8	<id,x> - <num,2> * *Factor*	x - ↑2 * y
—	<id,x> - <num,2> * *Factor*	x - 2 ↑* y
—	<id,x> - <num,2> * *Factor*	x - 2 * ↑y
9	<id,x> - <num,2> * <id,y>	x - 2 * ↑y
—	<id,x> - <num,2> * <id,y>	x - 2 * y↑



This time, we matched & consumed all the input

⇒ Success!

Another possible parse

Other choices for expansion are possible

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr + Term*	↑x - 2 * y
2	*Expr + Term + Term*	↑x - 2 * y
2	*Expr + Term + Term + Term*	↑x - 2 * y
2	*Expr + Term + Term + …+ Term*	↑x - 2 * y

consuming no input !

This doesn't terminate *(obviously)*

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is *left recursive* if $\exists\ A \in NT$ such that

$\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

Our expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$Fee \rightarrow Fee\ \alpha$$
$$\quad\quad |\ \beta$$

where neither α nor β start with *Fee*

We can rewrite this as

$$Fee \rightarrow \beta\ Fie$$
$$Fie \rightarrow \alpha\ Fie$$
$$\quad\quad |\ \varepsilon$$

where *Fie* is a new non-terminal

This accepts the same language, but uses only right recursion

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
	|	*Expr* - *Term*		|	*Term* / *Factor*
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	→	*Term* *Expr′*	*Term*	→	*Factor* *Term′*
Expr′	|	+ *Term* *Expr′*	*Term′*	|	* *Factor* *Term′*
	|	- *Term* *Expr′*		|	/ *Factor* *Term′*
	|	ε		|	ε

These fragments use only right recursion

Eliminating Left Recursion

Substituting them back into the grammar yields

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	number
11		|	id
12		|	(*Expr*)

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- General left recursion removal algorithm p.103 EAC

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - Context-free grammars
  - Ambiguity
- Top-down parsers
  - Algorithm & its problem with left recursion
  - Left-recursion removal
- Predictive top-down parsing
  - The LL(1) condition
  - Table-driven LL(1) parsers
  - Recursive descent parsers
    - Syntax directed translation (example)

Picking the "Right" Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are *LL(1)* and *LR(1)* grammars

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first (terminal) symbol in some string that derives from α

That is, $a \in$ FIRST(α) *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

The FIRST Set

$a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

To build FIRST(X) for all grammar symbols X:

1. if X is a terminal (token), FIRST(X) := { X }
2. if X ::= ε, then $\varepsilon \in$ FIRST(X)

3. <u>iterate until</u> no more terminals or ε can be added to any FIRST(X):

   if X ::= $Y_1 Y_2 \ldots Y_k$ then
      $a \in \text{FIRST}(X)$ if $a \in \text{FIRST}(Y_i)$ and
         $\varepsilon \in \text{FIRST}(Y_j)$ for all $1 \le j < i$
      $\varepsilon \in \text{FIRST}(X)$ if $\varepsilon \in \text{FIRST}(Y_i)$ for all $1 \le i \le k$
   <u>end iterate</u>

Note: if $\varepsilon \notin \text{FIRST}(Y_1)$, then $\text{FIRST}(Y_i)$ is irrelevant, for $1 < i$

The FIRST Set

$a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \underline{a}\,\gamma$, for some γ

To build FIRST(α) for $\alpha = X_1 X_2 \ldots X_n$:

1. $x \in \text{FIRST}(\alpha)$ if $x \in \text{FIRST}(X_i)$ and
   $\varepsilon \in \text{FIRST}(X_j)$ for all $1 \le j < i$

2. $\varepsilon \in \text{FIRST}(\alpha)$ if $\varepsilon \in \text{FIRST}(X_i)$ for all $1 \le i \le n$

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $a \in$ FIRST(α) *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct, but not quite

The FOLLOW Set

For a non-terminal A, define FOLLOW(A) as

FOLLOW(A) := the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it; a terminal has no FOLLOW set

The FOLLOW Set

To build FOLLOW(X) for all non-terminal X:

1. Place eof in FOLLOW(<*goal*>)

   iterate until no more terminals or ε can be added to any FOLLOW(X):

2. If $A \rightarrow \alpha B \beta$ then
   put {FIRST(β) - ε} in FOLLOW(B)
3. If $A \rightarrow \alpha B$ then
   put FOLLOW(A) in FOLLOW(B)
4. If $A \rightarrow \alpha B \beta$ and $\varepsilon \in$ FIRST(β) then
   put FOLLOW(A) in FOLLOW(B)

Predictive Parsing

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(A), too

Define FIRST^{+}(δ) for rule $A \rightarrow \delta$ as

- (FIRST(δ) - { ε }) $\cup$ FOLLOW(A), if $\varepsilon \in$ FIRST(δ)
- FIRST(δ), otherwise

Predictive Parsing

The LL(1) Property

A grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$\text{FIRST}^{+}(\alpha) \cap \text{FIRST}^{+}(\beta) = \varnothing$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

<u>Question</u>: Can there be two rules $A \rightarrow \alpha$ and $A \rightarrow \beta$ in a *LL(1)* grammar such that $\varepsilon \in \text{FIRST}(\alpha)$ and $\varepsilon \in \text{FIRST}(\beta)$?

Predictive Parsing

Given a grammar that has the *LL(1)* property

- Problem: NT *A* needs to be replaced in next derivation step
- Assume $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

  $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \varnothing$, $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \varnothing$, and $\text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$ (pair-wise disjoint sets)

```
/* find rule for A */
if (current token ∈ FIRST+ (β1))
   select A → β1
else if (current token ∈ FIRST+(β2))
   select A → β2
else if (current token ∈ FIRST+(β3))
   select A → β3
else
   report an error and return false
```

Grammars with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the *LL(1)* property are called *predictive parsers*.

One kind of predictive parser is the *recursive descent* parser. *The other is a table-driven parser* *table-driven parser.*

LL(1) Parser Example

Is the following grammar LL(1)?

S ::= a S b | ε

LL(1) Parser Example

Is the following grammar LL(1)?

S ::= a S b | ε

First(aSb) = { a }
First(ε) = { ε }

First+(aSb) = { a }
First+(ε) = (First (ε) - { ε }) ∪ Follow (S) = { eof, b }

LL(1)?

LL(1) Parser Example

Is the following grammar LL(1)?

$S ::= a\ S\ b \mid \varepsilon$

First(aSb) = { a }
First(ε) = { ε }

$\text{First}^{+}(aSb)$ = { a }
$\text{First}^{+}(\varepsilon)$ = (First (ε) - { ε }) ∪ Follow (S) = { eof, b }

LL(1)? YES, since { a } ∩ { eof, b } = ∅

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

LL(1) Parser Example

Table-driven LL(1) parser

current input symbol

	a	b	eof	other
S	aSb	ε	ε	error

rules for non-terminal

non-terminal on top of the stack

Building Top Down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an algorithm to build the table

Filling in TABLE[X,y], X $\in$ *NT*, y $\in$ *T*

- entry is the rule $X ::= \beta$, if y $\in$ FIRST+(β)
- entry is error otherwise

If any entry is defined multiple times, G is not *LL(1)*

This is the *LL(1)* table construction algorithm

LL(1) Skeleton Parser

```
token ← next_token()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
   if TOS = EOF and token = EOF then
      break & report success
    else if TOS is a terminal then
      if TOS matches token then
         pop Stack                // recognized TOS
         token ← next_token()
      else report error looking for TOS
    else                          // TOS is a non-terminal
      if TABLE[TOS,token] is A→ B1B2…Bk then
         pop Stack                // get rid of A
         push Bk, Bk-1, …, B1     // in that order
      else report error expanding TOS
   TOS ← top of Stack
```

Announcement

Homework 4 will be posted this weekend.

Next class: more Syntax Analysis (bottom-up)

Read EaC: Chapter 3.4