RUTGERS

THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Syntax Analysis Part 3

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

Announcements

- Midterm has been graded. Exams will be handed out in recitation. Please see canvas for grades.

- Third homework:
  Due Thursday, March 10

- First project (local instruction scheduler) deadlines:
  code: March 10 @ 11:59pm – single tar file
  report: March 11 @ 11:59pm – single pdf file
  Late policy:
  → Grace period: 1 hour
  → 20% penalty for every started 24 hour period after the deadline.
  → Saturday/Sunday count as a single 24 hour period.

- Warning grades will be submitted by Friday, March 11

Parsing
(Syntax Analysis)

Top-Down Parsing
EAC Chapters 3.3

Parsing Techniques: Top-down parsers

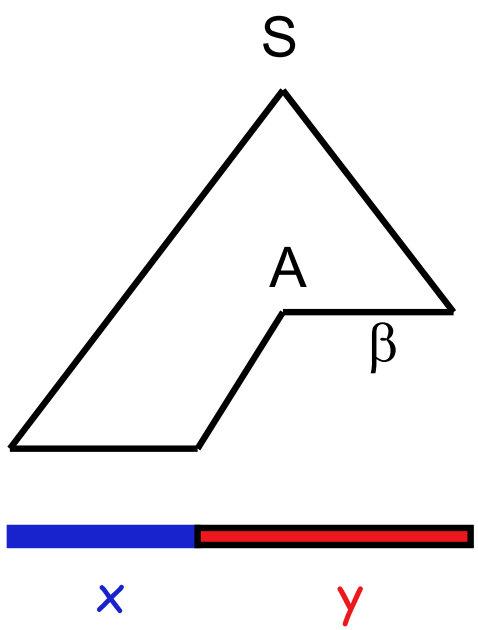
LL(1), recursive descent

1 input symbol lookahead

construct leftmost deriviation (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

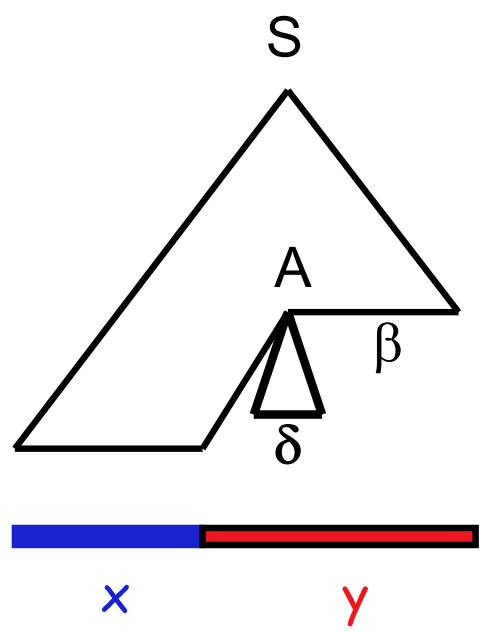
1 input symbol lookahead

construct leftmost deriviation (forwards)

input: read left-to-right

rule $A \rightarrow \delta$

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



The FIRST Set – 1 symbol lookahead

$$a \in \text{FIRST}_1(\alpha) \textit{ iff } \alpha \Rightarrow^* a\,\gamma, \text{ for some } \gamma$$

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

1. if X is a terminal (token), FIRST(X) := { X }
2. if $X \rightarrow \varepsilon$, then $\varepsilon \in$ FIRST(X)
3. <u>iterate until</u> no more terminals or ε can be added to any FIRST(X):

   if $X \rightarrow Y_1 Y_2 \ldots Y_k$ then

   $a \in$ FIRST(X) if $a \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \le j < i$

   $\varepsilon \in$ FIRST(X) if $\varepsilon \in$ FIRST(Y_i) for all $1 \le i \le k$

   <u>end iterate</u>

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

The FIRST Set

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

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

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

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

The FOLLOW Set – 1 symbol

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

$$\text{FOLLOW}(A) = \{\, a \in (T \cup \{\text{eof}\}) \mid S\ \text{eof} \Rightarrow^* \alpha\ A\ a\ \gamma \,\}$$

The FOLLOW Set

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

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

   <u>iterate until</u> no more terminals or eof 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

  $FIRST^+(\beta_1) \cap FIRST^+(\beta_2) = \varnothing$, $FIRST^+(\beta_1) \cap FIRST^+(\beta_3) = \varnothing$, and

  $FIRST^+(\beta_2) \cap FIRST^+(\beta_3) = \varnothing$ (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)?

$S \rightarrow a\ S\ b \mid \varepsilon$

LL(1) Parser Example

Is the following grammar LL(1)?

$$S \rightarrow a\ S\ b \mid \varepsilon$$

First(aSb) = { a }
First(ε) = { ε }

Follow(S). = { eof, b }

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

LL(1)?

LL(1) Parser Example

Is the following grammar LL(1)?

$$S \rightarrow a\ S\ b \mid \varepsilon$$

First(aSb) = { a }
First(ε) = { ε }

Follow(S). = { eof, b }

$First^{+}(aSb)$ = { a }
$First^{+}(\varepsilon)$ = (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

current input symbol

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* + *"eof"*
- Need an algorithm to build the table

Filling in TABLE[X,y], X $\in$ *NT*, y $\in$ $T \cup \{\text{eof}\}$

- entry is the rule $X \rightarrow \beta$, 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()  // scanner call
push EOF onto Stack // bottom of Stack marker
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
```

exit on success

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

How to parse input a a a b b b ?

Describe action as sequence of states

(PDA stack content, remaining input, next action)

use eof as bottom-of-stack marker

<u>PDA stack content</u>: [X, … Z], where Z is the TOS
<u>next actions</u>: rule or next input+pop or error or accept

LL(1) Parser Example

Table-driven LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

([eof, S], aaabbb, aSb) ⇒
([eof, b, S, a], aaabbb, next input+pop) ⇒
([eof, b, S], aabbb, aSb) ⇒
([eof, b, b, S, a], aabbb, next input+pop) ⇒
([eof, b, b, S], abbb, aSb) ⇒
([eof, b, b, b, S, a], abbb, next input+pop) ⇒
([eof, b, b, b, S], bbb, ε) ⇒
([eof, b, b, b], bbb, next input+pop) ⇒ ([eof, b, b], bb, next input+pop) ⇒
([eof, b], b, next input+pop) ⇒ ([eof], eof, accept)

LL(1) Parser Example

Recursive descent LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

1. Every NT is associated with a parsing procedure.

2. The parsing procedure for A ε NT, proc A, is responsible to parse and consume any (token) string that can be derived from A; it may recursively call other parsing procedures.

3. The parser is invoked by calling proc S for start symbol S.

LL(1) Parser Example

Recursive descent LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

```
main ( ) {
  token = next_token();
  if (S ( ) and token = eof )
     print "accept"
  else
     print "error";
}
```

```
bool S ( ) {
   switch token {
      case a: token = next_token();
                 if (not S ( ) ) return false;
                 if token = b
                    {token = next_token(); return true;}
                 else
                    return false;
                 break;
      case b, eof: return true; break;
    default: return false;
   }
}
```

LL(1) Parser Example

Recursive descent LL(1) parser

	a	b	eof	other
S	aSb	ε	ε	error

```
main ( ) {
  token = next_token();
  if (S ( ) and token = eof )
     print "accept"
  else
     print "error";
}
```

How to parse input a a a b b b ?

```
bool S ( ) {
   switch token {
      case a: token = next_token();
              if (not S ( ) ) return false;
              if token = b
                 {token = next_token(); return true;}
              else
                 return false;
              break;
      case b, eof: return true; break;
    default: return false;
   }
}
```

Reminder - 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

Left Factoring

What if my grammar does not have the LL(1) property?

$\Rightarrow$ Sometimes, we can transform the grammar

The Algorithm

$\forall\ A \in NT$,

find the longest prefix α *that occurs in two or more right-hand sides of A*

if $\alpha \neq \varepsilon$ *then replace all of the A productions,*

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma\ ,$$

with

$$A \rightarrow \alpha\ Z \mid \gamma$$

$$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where Z is a new element of NT

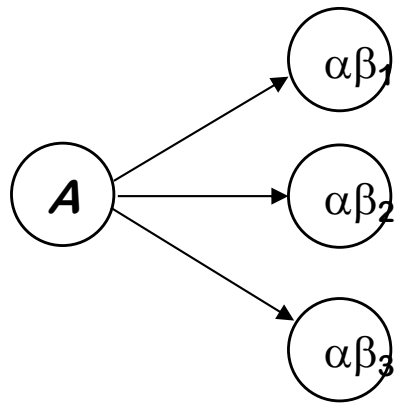
Repeat until no common prefixes remain

Left Factoring

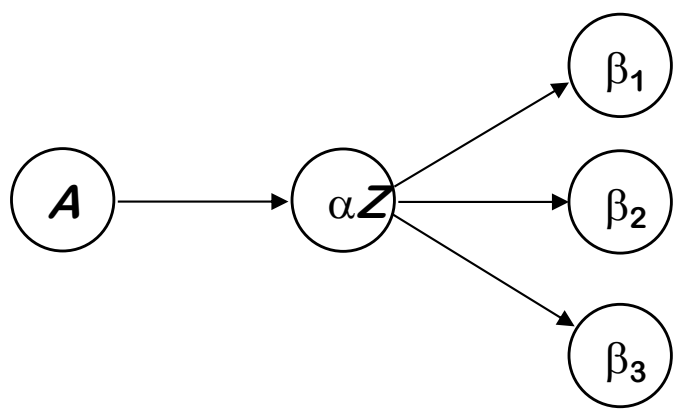
A graphical explanation for the same idea

$A \rightarrow \alpha\beta_1$
$\quad | \ \alpha\beta_2$
$\quad | \ \alpha\beta_3$

becomes …



$A \rightarrow \alpha\ Z$
$Z \rightarrow \beta_1$
$\quad | \ \beta_2$
$\quad | \ \beta_3$



Left Factoring (An example)

Consider the following fragment of the expression grammar

Factor	→	Identifier
	|	Identifier [*ExprList*]
	|	Identifier (*ExprList*)

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { Identifier }
FIRST(rhs_3) = { Identifier }

After left factoring, it becomes

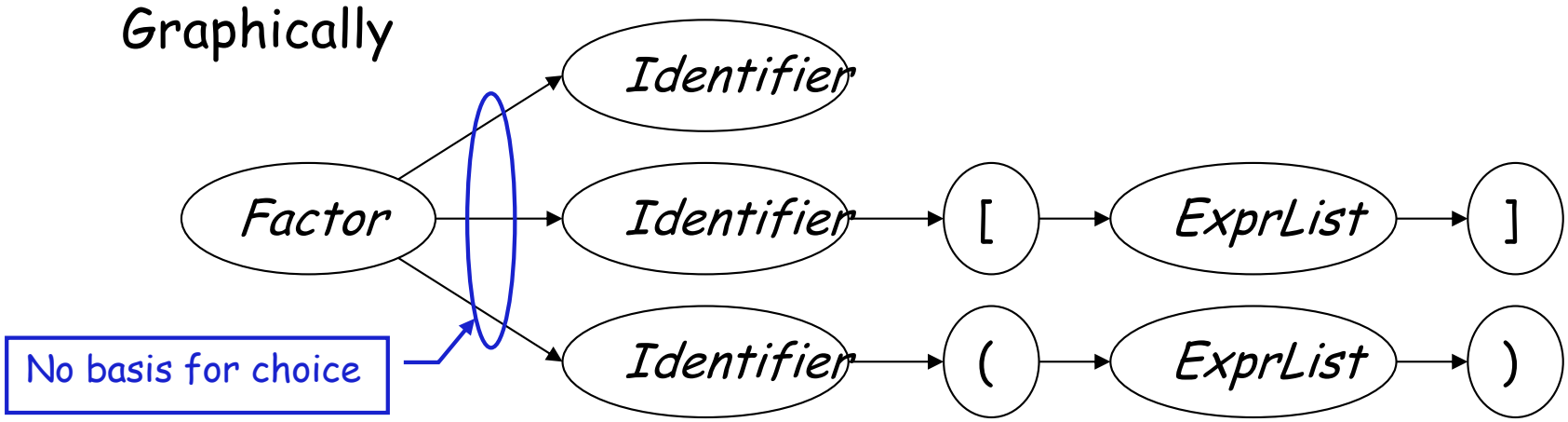
Factor	→	Identifier *Arguments*
Arguments	→	[*ExprList*]
	|	(*ExprList*)
	|	ε

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = FOLLOW(*Factor*)
⇒ It has the *LL(1)* property

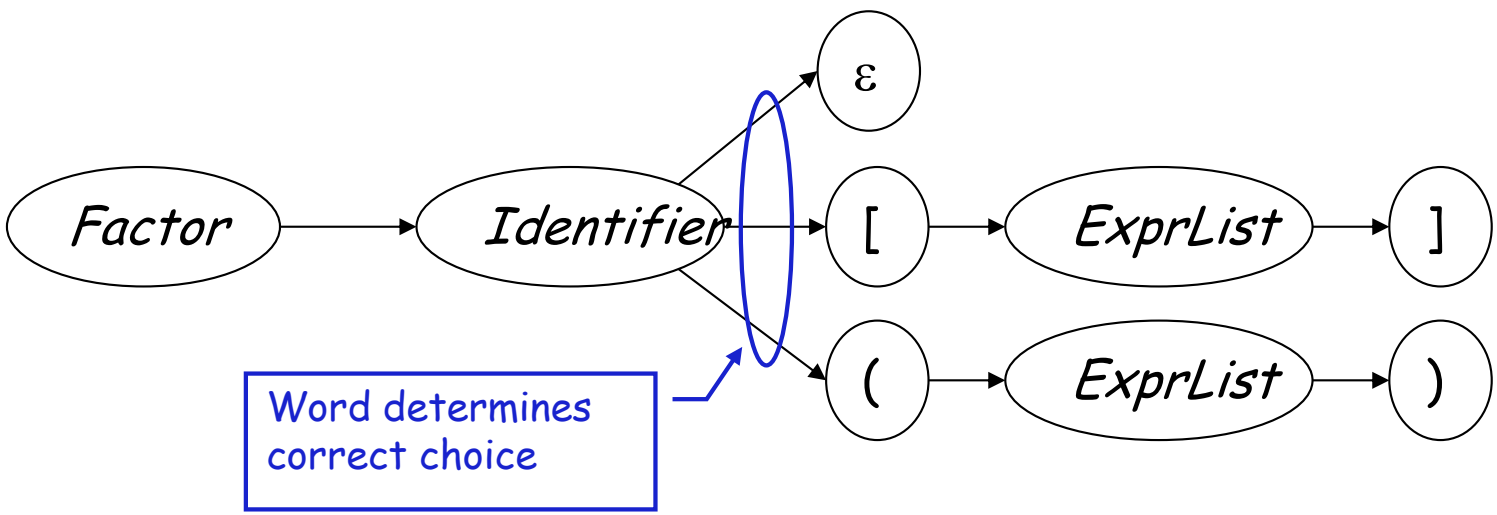
This form has the same syntax, with the *LL(1)* property

Left Factoring (An example)

Graphically



becomes …



LL(1) Grammars

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a CFG that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Language that cannot be LL(k)

Example

$\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(k)* grammar

$G \rightarrow \underline{a}A\underline{b}$
$\quad \mid \underline{a}B\underline{bb}$

$A \rightarrow \underline{a}A\underline{b}$
$\quad \mid \underline{0}$

$B \rightarrow \underline{a}B\underline{bb}$
$\quad \mid \underline{1}$

Problem: need an unbounded number of "$\underline{a}$" symbols before you can determine whether you are in the A group or the B group.

Next class

More Syntax Analysis (bottom-up)

LR(1) parsing

Read EaC: Chapter 3.4