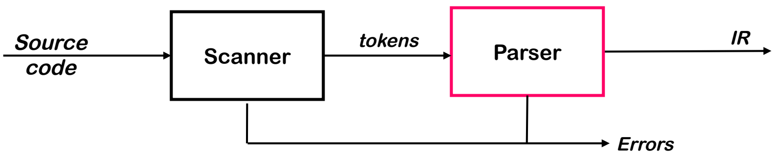
Parsing

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

The Front End



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Need a mathematical model of syntax — a grammar G
- Need an algorithm for testing membership in $L(G)$
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — a regular expression for arithmetic expressions

Term → [a-zA-Z] ([a-zA-Z] | [0-9])*

Op → + | - | * | /

Expr → (*Term Op*)* *Term*

([a-zA-Z] ([a-zA-Z] | [0-9])* (+ | - | * | /))* [a-zA-Z] ([a-zA-Z] | [0-9])

Limitation: Operator precedence?

2

Grammars

0	*Expr*	→	*Expr Op Expr*
1		|	number
2		|	id
3	*Op*	→	+
4		|	-
5		|	*
6		|	/

Rule	*Sentential Form*
—	*Expr*
0	*Expr Op Expr*
2	<id,x> *Op Expr*
4	<id,x> - *Expr*
0	<id,x> - *Expr Op Expr*
1	<id,x> - <num,2> *Op Expr*
5	<id,x> - <num,2> * *Expr*
2	<id,x> - <num,2> * <id,y>

- Such a sequence of rewrites is called a *derivation*
- Process of discovering a derivation is called *parsing*

We denote this derivation: $Expr \Rightarrow^*$ id – num * id

3

Derivations

The point of parsing is to construct a derivation

- At each step, we choose a nonterminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- *Leftmost derivation* — replace leftmost NT at each step
- *Rightmost derivation* — replace rightmost NT at each step

These are the two *systematic* derivations

(We don't care about randomly-ordered derivations!)

The Two Derivations for x – 2 * y

Rule	*Sentential Form*
—	*Expr*
0	*Expr Op Expr*
2	<id,x> *Op Expr*
4	<id,x> - *Expr*
0	<id,x> - *Expr Op Expr*
1	<id,x> - <num,2> *Op Expr*
5	<id,x> - <num,2> * *Expr*
2	<id,x> - <num,2> * <id,y>

Leftmost derivation

Rule	*Sentential Form*
—	*Expr*
0	*Expr Op Expr*
2	*Expr Op* <id,y>
5	*Expr* * <id,y>
0	*Expr Op Expr* * <id,y>
1	*Expr Op* <num,2> * <id,y>
4	*Expr* - <num,2> * <id,y>
2	<id,x> - <num,2> * <id,y>

Rightmost derivation

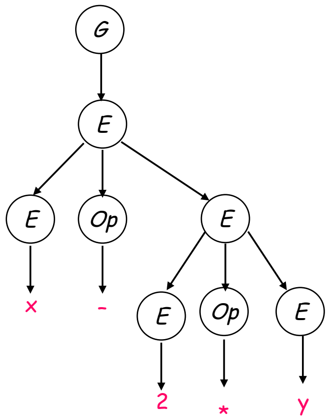
In both cases, *Expr* $\Rightarrow^*$ id – num * id

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

Leftmost derivation

Rule	Sentential Form
—	*Expr*
0	*Expr Op Expr*
2	<id,x> *Op Expr*
4	<id,x> - *Expr*
0	<id,x> - *Expr Op Expr*
1	<id,x> - <num,2> *Op Expr*
5	<id,x> - <num,2> * *Expr*
2	<id,x> - <num,2> * <id,y>

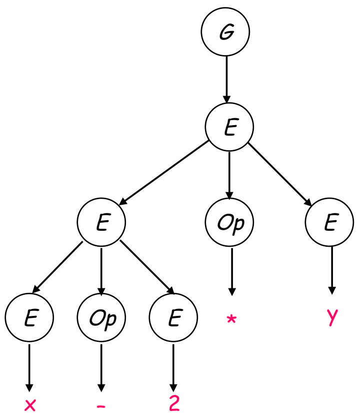


This evaluates as x - (2 * y)

Derivations and Parse Trees

Rightmost derivation

Rule	Sentential Form
—	*Expr*
0	*Expr Op Expr*
2	*Expr Op* <id,y>
5	*Expr* * <id,y>
0	*Expr Op Expr* * <id,y>
1	*Expr Op* <num,2> * <id,y>
4	*Expr* - <num,2> * <id,y>
2	<id,x> - <num,2> * <id,y>



This evaluates as (x - 2) * y

This ambiguity is NOT good

Derivations and Precedence

These two derivations point out a problem with the grammar:
It has no notion of precedence*, or implied order of evaluation*

To add precedence
- Create a nonterminal for each *level of precedence*
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions
- Parentheses first (*level 1*)
- Multiplication and division, next (*level 2*)
- Subtraction and addition, last (*level 3*)

8

Derivations and Precedence

Adding the standard algebraic precedence produces:

	0	*Goal*	→	*Expr*
level 3	1	*Expr*	→	*Expr + Term*
	2		|	*Expr - Term*
	3		|	*Term*
level 2	4	*Term*	→	*Term * Factor*
	5		|	*Term / Factor*
	6		|	*Factor*
level 1	7	*Factor*	→	(*Expr*)
	8		|	number
	9		|	id

This grammar is slightly larger
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations
- Correctness trumps the speed of the parser

*Let's see how it parses x - 2 * y*

Cannot handle precedence in an RE for expressions

Introduced parentheses, too (beyond power of an RE)

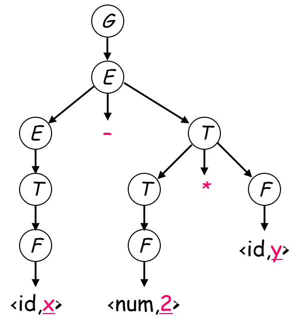
One form of the "classic expression grammar"

9

Derivations and Precedence

Rule	Sentential Form
—	*Goal*
0	*Expr*
2	*Expr* - *Term*
4	*Expr* - *Term* * *Factor*
9	*Expr* - *Term* * <id,y>
6	*Expr* - *Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
3	*Term* - <num,2> * <id,y>
6	*Factor* - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

The rightmost derivation



Its parse tree

It derives x – (2 * y), along with an appropriate parse tree.
Both the leftmost and rightmost derivations give the same expression, because the grammar directly and explicitly encodes the desired precedence.

10

Ambiguous Grammars

Let's leap back to our original expression grammar.
It had other problems.

0	*Expr*	→	*Expr Op Expr*
1		|	number
2		|	id
3	*Op*	→	+
4		|	-
5		|	*
6		|	/

Rule	Sentential Form
—	*Expr*
0	*Expr Op Expr*
2	<id,x> *Op Expr*
4	<id,x> - *Expr*
0	<id,x> - *Expr Op Expr*
1	<id,x> - <num,2> *Op Expr*
5	<id,x> - <num,2> * *Expr*
2	<id,x> - <num,2> * <id,y>

- This grammar allows multiple leftmost derivations for x - 2 * y
- Hard to automate derivation if > 1 choice
- The grammar is *ambiguous*

Different choice than the first step

11

Two Leftmost Derivations for *x* – *2* * *y*

The Difference:

- Different productions chosen on the second step

Rule	Sentential Form
—	*Expr*
0	*Expr Op Expr*
2	*<id,x> Op Expr*
4	*<id,x> - Expr*
0	*<id,x> - Expr Op Expr*
1	*<id,x> - <num,2> Op Expr*
5	*<id,x> - <num,2> * Expr*
1	*<id,x> - <num,2> * <id,y>*

Original choice

Rule	Sentential Form
—	*Expr*
0	*Expr Op Expr*
0	*Expr Op Expr Op Expr*
2	*<id,x> Op Expr Op Expr*
4	*<id,x> - Expr Op Expr*
1	*<id,x> - <num,2> Op Expr*
5	*<id,x> - <num,2> * Expr*
2	*<id,x> - <num,2> * <id,y>*

New choice

- Both derivations succeed in producing *x - 2 * y*

12

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single *sentential form*, the grammar is *ambiguous*
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is *ambiguous*
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar
  - — However, they must have the same parse tree!

Classic example — the *if*-*then*-*else* problem

Stmt → if *Expr* then *Stmt*
| if *Expr* then *Stmt* else *Stmt*
| … *other stmts* …

This ambiguity is inherent in the grammar

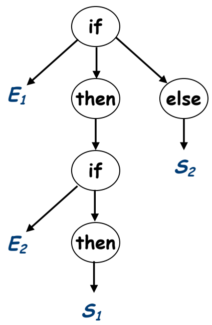
13

Ambiguity

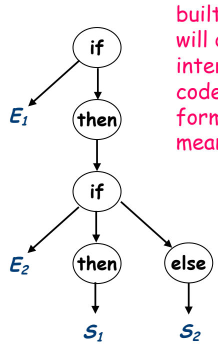
This sentential form has two derivations

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$

Part of the problem is that the structure built by the parser will determine the interpretation of the code, and these two forms have different meanings!



production 2, then production 1



production 1, then production 2

14

Ambiguity

The grammar forces the structure to match the desired meaning.

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if *(common sense rule)*

0	*Stmt*	→	if *Expr* then *Stmt*
1		|	if *Expr* then *WithElse* else *Stmt*
2		|	*Other Statements*
3	*WithElse*	→	if *Expr* then *WithElse* else *WithElse*
4		|	*Other Statements*

With this grammar, example has only one rightmost derivation

Intuition: once into *WithElse*, we cannot generate an unmatched else

… a final if without an else can only come through rule 2 …

15

Ambiguity

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$

Rule	*Sentential Form*
—	*Stmt*
0	if *Expr* then *Stmt*
1	if *Expr* then if *Expr* then *WithElse* else *Stmt*
2	if *Expr* then if *Expr* then *WithElse* else S_2
4	if *Expr* then if *Expr* then S_1 else S_2
?	if *Expr* then if E_2 then S_1 else S_2
?	if E_1 then if E_2 then S_1 else S_2

Other productions to derive *Exprs*

This grammar has only one rightmost derivation for the example

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

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
- Bottom-up parsers handle a large class of grammars

Top-down Parsing

Top-down Parsing

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until lower 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?

We will call this version "the classic expression grammar"

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr + Term*
2		|	*Expr - Term*
3		|	*Term*
4	*Term*	→	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

And the input x – 2 * y

20

Example

Let's try x – 2 * y :

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y

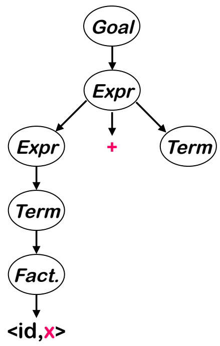
↑ *is the position in the input buffer*

21

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr* + *Term*	↑x - 2 * y
3	*Term* + *Term*	↑x - 2 * y
6	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
→	<id,x> + *Term*	x ↑- 2 * y



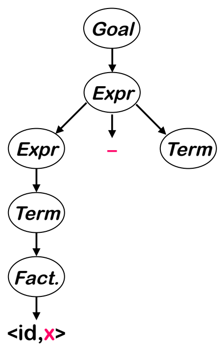
This worked well, except that "–" doesn't match "+"
The parser must backtrack to here

↑ *is the position in the input buffer*

Example

Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
2	*Expr* - *Term*	↑x - 2 * y
3	*Term* - *Term*	↑x - 2 * y
6	*Factor* - *Term*	↑x - 2 * y
9	<id,x> - *Term*	↑x - 2 * y
→	<id,x> - *Term*	x ↑- 2 * y
→	<id,x> - *Term*	x - ↑2 * y



Now, "-" and "-" match

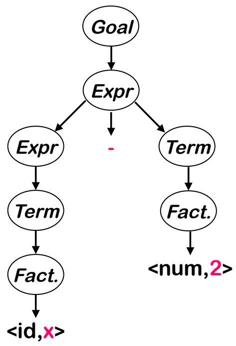
Now we can expand Term to match "2"

⇒ 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
6	<id,x> - *Factor*	x - ↑2 * y
8	<id,x> - <num,2>	x - ↑2 * y
→	<id,x> - <num,2>	x - 2 ↑* y



Where are we?

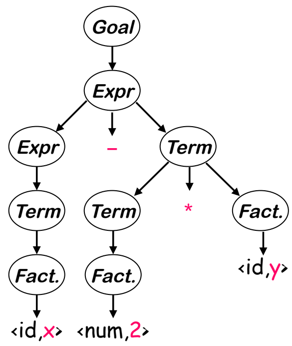
- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example

Trying again with "2" in x – 2 * y :

Rule	Sentential Form	Input
→	<id,x> - *Term*	x - ↑2 * y
4	<id,x> - *Term* * *Factor*	x - ↑2 * y
6	<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↑



The Point:

The parser must make the right choice when it expands a NT.
Wrong choices lead to wasted effort.

Another possible parse

Other choices for expansion are possible

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr + Term*	↑x - 2 * y
1	*Expr + Term + Term*	↑x - 2 * y
1	*Expr + Term + Term + Term*	↑x - 2 * y
1	*And so on ....*	↑x - 2 * y

Consumes no input!

This expansion doesn't terminate

- 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 classic expression grammar is left recursive

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

Non-termination is <u>always</u> a bad property in a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$
\begin{aligned}
\mathit{Fee} &\rightarrow \mathit{Fee}\ \alpha \\
&\mid\ \beta
\end{aligned}
$$

where neither α nor β start with *Fee*

We can rewrite this fragment as

$$
\begin{aligned}
\mathit{Fee} &\rightarrow \beta\ \mathit{Fie} \\
\mathit{Fie} &\rightarrow \alpha\ \mathit{Fie} \\
&\mid\ \varepsilon
\end{aligned}
$$

where *Fie* is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string

28

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \mathit{Expr} + \mathit{Term} \qquad & \mathit{Term} &\rightarrow \mathit{Term} * \mathit{Factor} \\
&\mid\ \mathit{Expr} - \mathit{Term} & &\mid\ \mathit{Term} * \mathit{Factor} \\
&\mid\ \mathit{Term} & &\mid\ \mathit{Factor}
\end{aligned}
$$

Applying the transformation yields

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \mathit{Term}\ \mathit{Expr}' \qquad & \mathit{Term} &\rightarrow \mathit{Factor}\ \mathit{Term}' \\
\mathit{Expr}' &\rightarrow +\ \mathit{Term}\ \mathit{Expr}' & \mathit{Term}' &\rightarrow *\ \mathit{Factor}\ \mathit{Term}' \\
&\mid\ -\ \mathit{Term}\ \mathit{Expr}' & &\mid\ /\ \mathit{Factor}\ \mathit{Term}' \\
&\mid\ \varepsilon & &\mid\ \varepsilon
\end{aligned}
$$

These fragments use only right recursion

Right recursion often means right associativity. In this case, the grammar does not display any particular associative bias.

Eliminating Left Recursion

Substituting them back into the grammar yields

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	number
11		|	id

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
  - $\Rightarrow$ The naïve transformation yields a right recursive grammar, which changes the implicit associativity
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.

30

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

We will focus, for now, on LL(1) grammars & predictive parsing

31

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

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, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\, \gamma$, 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
See the next slide

Predictive Parsing

What about ε-productions?

⇒ They complicate the definition of LL(1)

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, where

FOLLOW(A) = the set of terminal symbols that can immediately follow A in a sentential form

Define FIRST$^+$($A \rightarrow \alpha$) as

- FIRST(α) ∪ FOLLOW(A), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

Then, a grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies FIRST$^+$($A \rightarrow \alpha$) ∩ FIRST$^+$($A \rightarrow \beta$) = ∅

34

Classic Expression Grammar

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

FIRST$^+$($A \rightarrow \beta$) is identical to FIRST(β) except for productiond 4 and 8

FIRST$^+$(Expr' → ε) is {ε,), eof}

FIRST$^+$(Term' → ε) is {ε,+,-,), eof}

Symbol	FIRST	FOLLOW
num	num	Ø
id	id	Ø
+	+	Ø
-	-	Ø
*	*	Ø
/	/	Ø
(	(	Ø
)	)	Ø
eof	eof	Ø
ε	ε	Ø
Goal	(,id,num	eof
Expr	(,id,num	), eof
Expr'	+, -, ε	), eof
Term	(,id,num	+, -,), eof
Term'	*, /, ε	+,-,), eof
Factor	(,id,num	+,-,*,/,),eof

Classic Expression Grammar

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

Prod'n	FIRST+
0	(,id,num
1	(,id,num
2	+
3	-
4	ε,), eof
5	(,id,num
6	*
7	/
8	ε,+,-,), eof
9	number
10	id
11	(

36

Example

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Expr*
3		|	- *Expr*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Term*
7		|	/ Term
8		|	ε
9	*Factor*	→	number
10		|	id

Clearly,

FIRST+(2), FIRST+(3), & FIRST+(4)

are disjoint, as are

FIRST+(6), FIRST+(7), & FIRST+(8)

The grammar now has the LL(1) property

37

Predictive Parsing

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

- Can write a simple routine to recognize each *lhs*
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$$FIRST^+(A \rightarrow \beta_i) \cap FIRST^+(A \rightarrow \beta_j) = \emptyset \text{ if } i \neq j$$

```
/* find an A */
if (current_word ∈ FIRST(A→β1))
   find a β1 and return true
else if (current_word ∈ FIRST(A→β2))
   find a β2 and return true
else if (current_word ∈ FIRST(A→β3))
   find a β3 and return true
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.

Of course, there is more detail to "find a β_i" (p. 103 in EAC, 1st Ed.)

38

Recursive Descent Parsing

Recall the expression grammar, after transformation

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	number
11		|	id

This produces a parser with six *mutually recursive* routines:

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *Factor*

Each recognizes one *NT* or *T*

The term *descent* refers to the direction in which the parse tree is built.

39

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
   token ← next_token( );
   if (Expr( ) = true & token = EOF)
      then report success;
      else
         report syntax error;
         return false;

Expr( )
  if (Term( ) = false)
     then return false;
     else return Eprime( );
```

looking for Number, Identifier, or "(", found token instead, or failed to find Expr or ")" after "("

```
Factor( )
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
     token ← next_token( );
     return true;
  else if (token = Lparen)
     token ← next_token( );
     if (Expr( ) = true & token = Rparen) then
        token ← next_token( );
        return true;
  // fall out of if statement
  report syntax error;
  return false;
```

EPrime, Term, & TPrime follow the same basic lines (Figure 3.10, EaC2)

Bottom-up Parsing

Parsing Techniques

Bottom-up parsers *LR(1)*

- 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
- Bottom-up parsers build a rightmost derivation in reverse
- Parsers can be auto-generated from grammars

- We will skip this topic…

42

Summary

	Advantages	*Disadvantages*
Top-down Recursive descent, LL(1)	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

43