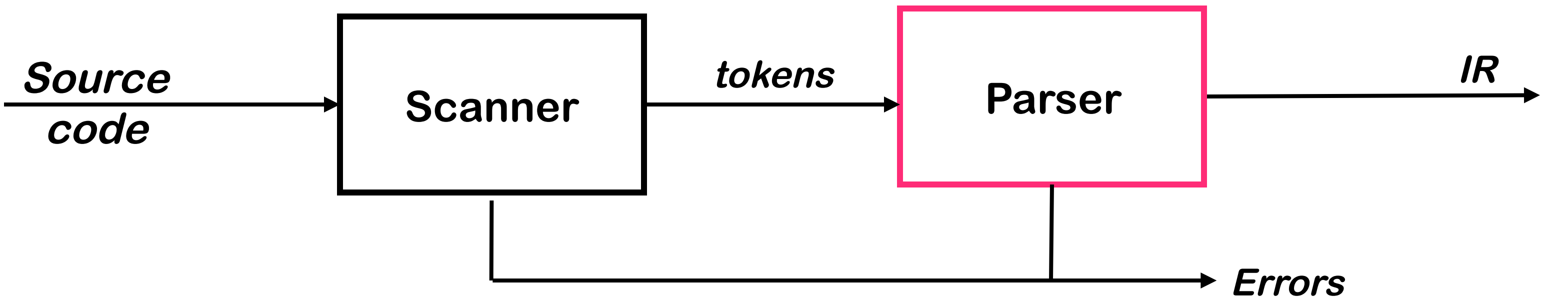
COMP 412
FALL 2010

Introduction to Parsing

Comp 412

Modified

The Front End



Parser

- Checks the stream of <u>words</u> and their <u>parts of speech</u> (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

Think of this chapter as the mathematics of diagramming sentences

The Study of Parsing

The process of discovering a *derivation* for some sentence

- 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

Roadmap for our study of parsing

1. Context-free grammars and derivations
2. Top-down parsing
   - Generated LL(1) parsers & hand-coded recursive descent parsers
3. Bottom-up parsing
   - Generated LR(1) parsers

Specifying Syntax with a Grammar

Context-free syntax is specified with a context-free grammar

$$\begin{array}{lcl} \textit{SheepNoise} & \rightarrow & \textit{SheepNoise}\ \underline{\text{baa}} \\ & | & \underline{\text{baa}} \end{array}$$

This *CFG* defines the set of noises sheep normally make

It is written in a variant of Backus–Naur form

Formally, a grammar is a four tuple, $G = (S,N,T,P)$

- S is the *start symbol* (set of strings in L(G))
- N is a set of *nonterminal symbols* (syntactic variables)
- T is a set of *terminal symbols* (words)
- P is a set of *productions* or *rewrite rules* $(P : N \rightarrow (N \cup T)^{+})$

Example due to Dr. Scott K. Warren

From Lecture 1

Why Not Use Regular Languages & DFAs?

Removed for time

Context-free Grammars

What makes a grammar "context free"?

The SheepNoise grammar has a specific form:

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \textit{SheepNoise}\ \underline{\text{baa}} \\ &\ \mid\ \underline{\text{baa}} \end{aligned}$$

Productions have a single nonterminal on the left hand side, which makes it impossible to encode left or right context.

⇒ The grammar is context free.

A context-sensitive grammar can have ≥ 1 nonterminal on lhs.

Notice that *L(SheepNoise)* is actually a regular language: $\underline{\text{baa}}^{+}$

Classic definition: any language that can be recognized by a push-down automaton is a context-free language.

A More Useful Grammar Than Sheep Noise

To explore the uses of CFGs,we need a more complex grammar

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

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

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

We denote this derivation: *Expr* $\Rightarrow^*$ <u>id</u> – <u>num</u> * <u>id</u>

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!)

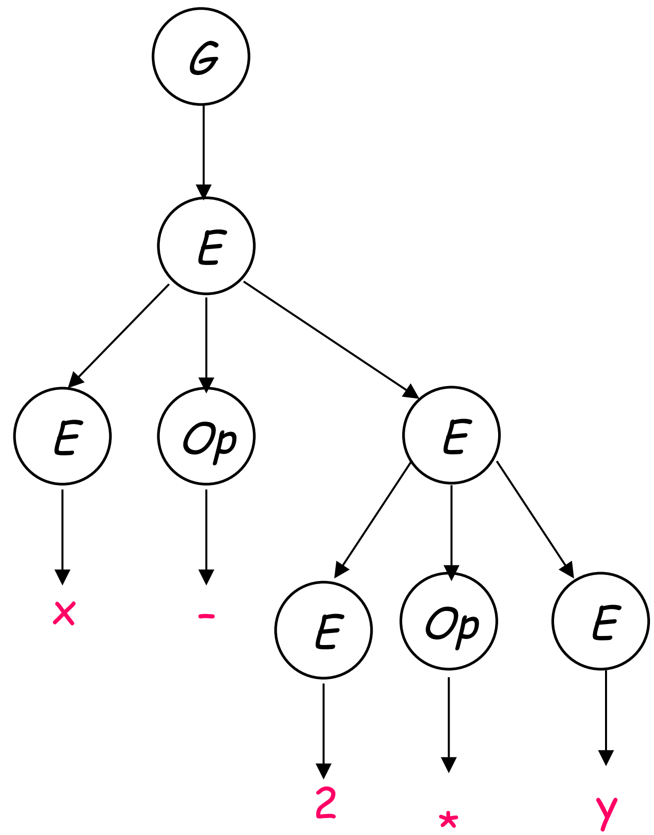
The example on the preceding slide was a *leftmost* derivation

- Of course, there is also a *rightmost* derivation
- Interestingly, it turns out to be different

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>

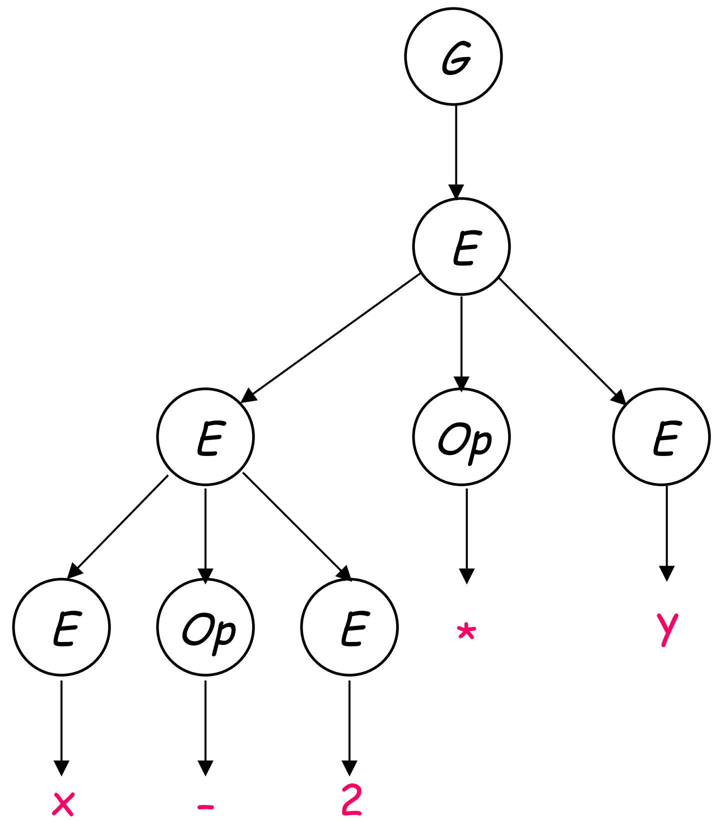


This evaluates as x - (2 * y)

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 <u>precedence,</u> *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*)

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

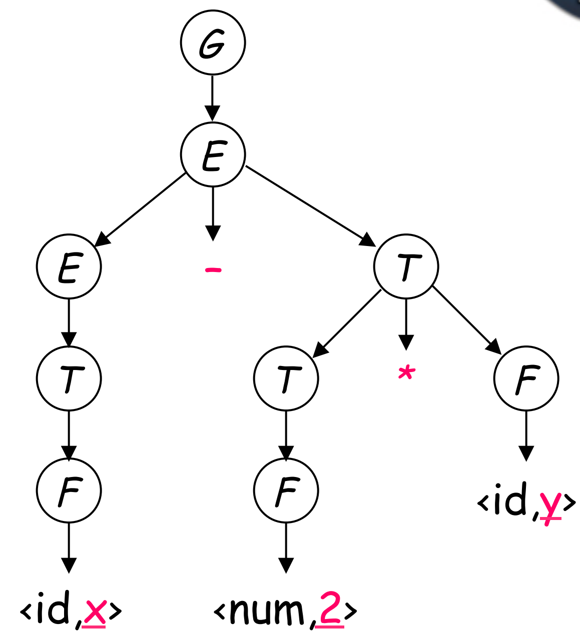
Introduced parentheses, too (beyond power of an RE)

One form of the "classic expression grammar"

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

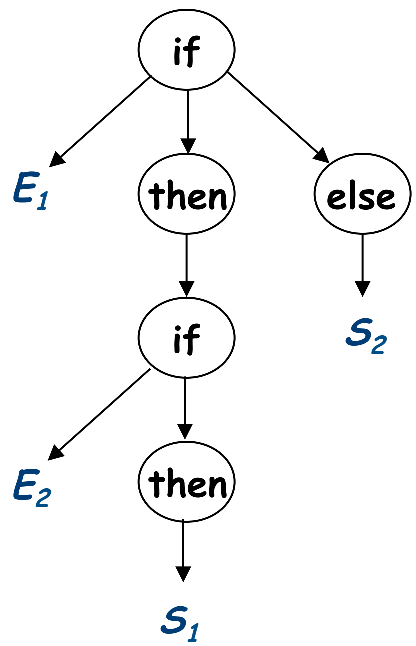
It derives x – (2 * y), along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same expression, because the grammar directly and explicitly encodes the desired precedence.

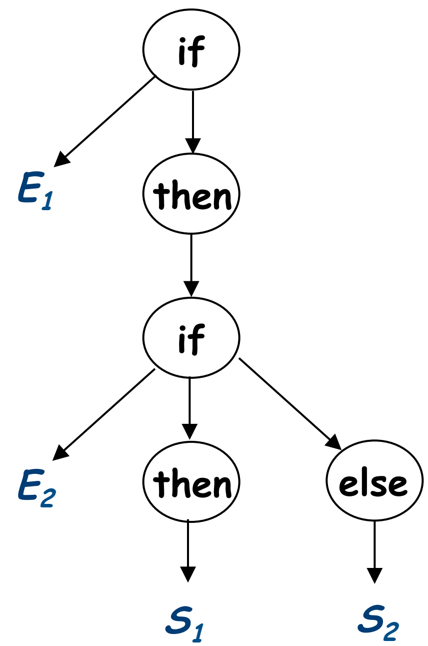
Ambiguity

Removed for time

<u>if</u> *Expr*$_1$ <u>then</u> <u>if</u> *Expr*$_2$ <u>then</u> *Stmt*$_1$ <u>else</u> *Stmt*$_2$



production 2, then production 1



production 1, then production 2

Top Down Parsing

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

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 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"

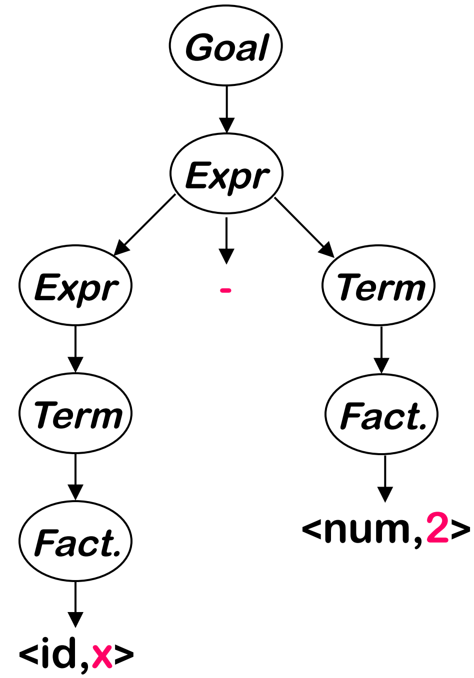
0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

And the input x – 2 * y

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?

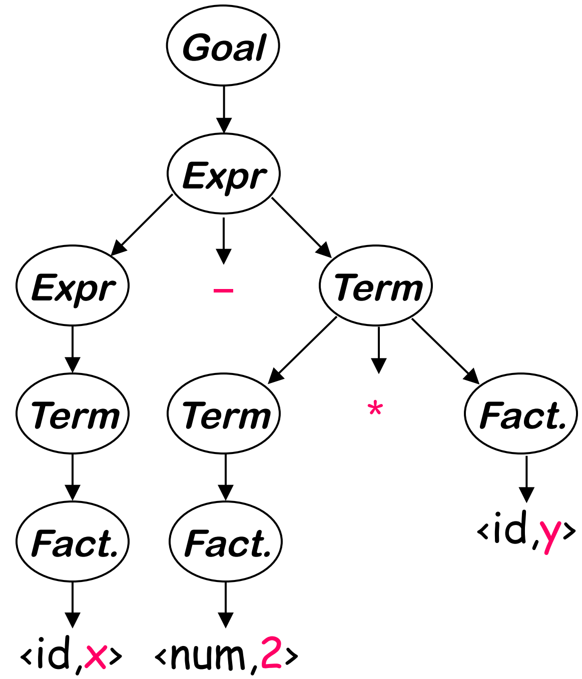
- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

⇒ 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.

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

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

where neither α nor β start with *Fee*

We can rewrite this fragment as

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

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

where *Fie* is a new non-terminal

Added a reference to the empty string

Eliminating Left Recursion

Substituting them back into the grammar yields

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*	→	(*Expr*)
10		|	number
11		|	id

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
  - ⇒ 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.

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

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

We will defer the problem of how to compute FIRST sets for the moment.

Predictive Parsing

What about ε-productions?

$\Rightarrow$ 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) $\cup$ 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$^+$($A\rightarrow\alpha$) $\cap$ FIRST$^+$($A\rightarrow\beta$) = $\varnothing$

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

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

```
/* 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.)

Recursive Descent Parsing

Recall the expression grammar, after transformation

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*	→	<u>(</u> *Expr* <u>)</u>
10		|	<u>number</u>
11		|	<u>id</u>

This produces a parser with six <u>*mutually recursive*</u> routines:

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

Each recognizes one *NT* or *T*

The term <u>*descent*</u> refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
    token ← next_token( );
    if (Expr( ) = true & token = EOF)
       then next compilation step;
       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 3.7, EAC)

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→β) 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

Building Top-down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an interpreter for the table (*skeleton parser*)

LL(1) Expression Parsing Table

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Row we built earlier

LL(1) Skeleton Parser

```
word ← NextWord()                    // Initial conditions, including
push EOF onto Stack                  // a stack to track local goals
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and word = EOF then
     break & report success          // exit on success
  else if TOS is a terminal then
     if TOS matches word then
        pop Stack                    // recognized TOS
        word ← NextWord()
     else report error looking for TOS  // error exit
  else                               // TOS is a non-terminal
     if TABLE[TOS,word] is A→ B1B2…Bk then
        pop Stack                    // get rid of A
        push Bk, Bk-1, …, B1         // in that order
     else break & report error expanding TOS
  TOS ← top of Stack
```

Bottom-up Parsing

Bottom-up Parsing (definitions)

The point of parsing is to construct a derivation

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

- Each γ_i is a sentential form
  - If γ contains only terminal symbols, γ is a sentence in $L(G)$
  - If γ contains 1 or more non-terminals, γ is a sentential form
- To get γ_i from γ_{i-1}, expand some NT $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
  - Replace the occurrence of $A \in \gamma_{i-1}$ with β to get γ_i
  - In a leftmost derivation, it would be the first NT $A \in \gamma_{i-1}$

A *left-sentential form* occurs in a *leftmost* derivation

A *right-sentential form* occurs in a *rightmost* derivation

Bottom-up parsers build a rightmost derivation in reverse

We saw this definition earlier

Bottom-up Parsing (definitions)

A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol S

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

bottom-up

To reduce γ_i to γ_{i-1} match some *rhs* β against γ_i then replace β with its corresponding *lhs*, A. *(assuming the production $A \rightarrow \beta$)*

In terms of the parse tree, it works from leaves to root

- Nodes with no parent in a partial tree form its *upper fringe*
- Since each replacement of β with A shrinks the upper fringe, we call it a *reduction*.
- "Rightmost derivation in reverse" processes words *left to right*

The parse tree need not be built, it can be simulated

$$|\textit{parse tree nodes}| = |\textit{terminal symbols}| + |\textit{reductions}|$$

Finding Reductions

Consider the grammar

0	*Goal*	→	<u>a</u> *A* *B* <u>e</u>
1	*A*	→	*A* <u>b</u> <u>c</u>
2		|	<u>b</u>
3	*B*	→	<u>d</u>

And the input string <u>abbcde</u>

Sentential Form	*Next Reduction Prod'n*	*Pos'n*
<u>abbcde</u>	2	2
<u>a</u> *A* <u>bcde</u>	1	4
<u>a</u> *A* <u>de</u>	3	3
<u>a</u> *A* *B* <u>e</u>	0	4
Goal	—	—

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

"Position" specifies where the right end of β occurs in the current sentential form.

While the process of finding the next reduction appears to be almost oracular, it can be automated in an efficient way for a large class of grammars

Finding Reductions (Handles)

The parser must find a substring β of the tree's frontier that *matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation* *($\Rightarrow \beta \rightarrow A$ is in RRD)*

Informally, we call this substring β a *handle*

Formally,

A *handle* of a right-sentential form γ is a pair $\langle A \rightarrow \beta, k \rangle$ where $A \rightarrow \beta \in P$ and k is the position in γ of β's rightmost symbol.

If $\langle A \rightarrow \beta, k \rangle$ is a handle, then replacing β at k with A produces the right sentential form from which γ is derived in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols

$\Rightarrow$ the parser doesn't need to scan (*much*) past the handle

Most students find handles mystifying; bear with me for a couple more slides.

Example

derivation

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

A simple left-recursive form of the classic expression grammar

Prod'n	*Sentential Form*	*Handle*
—	*Goal*	—
0	*Expr*	0,1
2	*Expr* - *Term*	2,3
4	*Expr* - *Term* * *Factor*	4,5
8	*Expr* - *Term* * <id,<u>y</u>>	8,5
6	*Expr* - *Factor* * <id,<u>y</u>>	6,3
7	*Expr* - <num,<u>2</u>> * <id,<u>y</u>>	7,3
3	*Term* - <num,<u>2</u>> * <id,<u>y</u>>	3,1
6	*Factor* - <num,<u>2</u>> * <id,<u>y</u>>	6,1
8	<id,<u>x</u>> - <num,<u>2</u>> * <id,<u>y</u>>	8,1

parse

Handles for rightmost derivation of <u>x - 2 * y</u>

Bottom-up Parsing (Abstract View)

A bottom-up parser repeatedly finds a handle $A \rightarrow \beta$ in the current right-sentential form and replaces β with A.

To construct a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

Apply the following conceptual algorithm

for $i \leftarrow n$ *to 1 by –1*
    Find the handle $\langle A_i \rightarrow \beta_i , k_i \rangle$ in γ_i
    Replace β_i *with* A_i *to generate* γ_{i-1}

of course, *n* is unknown until the derivation is built

This takes *2n* steps

Some authors refer to this algorithm as a *handle-pruning parser.*

The idea is that the parser finds a *handle* on the upper fringe of the partially complete parse tree and *prunes* it out of the fringe.

The analogy is somewhat strained, so I will try to avoid using it.

More on Handles

Bottom-up reduce parsers find a rightmost derivation in reverse order

- Rightmost derivation ⇒ rightmost NT expanded at each step in the derivation
- Processed in reverse ⇒ parser proceeds left to right

These statements are somewhat counter-intuitive

More on Handles

Bottom-up parsers find a reverse rightmost derivation

- Process input left to right
  - Upper fringe of partially completed parse tree is *(NT | T)* T**
  - The handle always appears with its right end at the junction between *(NT | T)** and *T** (*the hot spot for LR parsing*)
  - We can keep the prefix of the upper fringe of the partially completed parse tree on a stack
  - The stack makes the position information irrelevant
- Handles appear at the top of the stack
- All the information for the decision is at the hot spot
  - The next word in the input stream
  - The rightmost NT on the fringe & its immediate left neighbors
  - In an LR parser, additional information in the form of a "state"

Shift-reduce Parsing

To implement a bottom-up parser, we adopt the shift-reduce paradigm

A shift-reduce parser is a stack automaton with four actions

- *Shift* — next word is shifted onto the stack
- *Reduce* — right end of handle is at top of stack
  - Locate left end of handle within the stack
  - Pop handle off stack & push appropriate *lhs*
- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Accept & Error are simple

Shift is just a push and a call to the scanner

Reduce takes |*rhs*| pops & 1 push

But how does the parser know when to shift and when to reduce?
It shifts until it has a handle at the top of the stack.

Bottom-up Parser

A simple *shift-reduce parser:*

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle A→β
        then      // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
        else if (token ≠ EOF)
            then // shift
                push token
                token ← next_token( )
        else     // need to shift, but out of input
            report an error
```

What happens on an error?

- It fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.

Error localization is an issue in the handle-finding process that affects the practicality of shift-reduce parsers…

We will fix this issue later.

Figure 3.7 in EAC

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	none	shift
$ id	- num * id	8,1	reduce 8
$ *Factor*	- num * id	6,1	reduce 6
$ *Term*	- num * id	3,1	reduce 3
$ *Expr*	- num * id	none	shift
$ *Expr* -	num * id	none	shift
$ *Expr* - num	* id	7,3	reduce 7
$ *Expr* - *Factor*	* id	6,3	reduce 6
$ *Expr* - *Term*	* id	none	shift
$ *Expr* - *Term* *	id	none	shift
$ *Expr* - *Term* * id		8,5	reduce 8
$ *Expr* - *Term* * *Factor*		4,5	reduce 4
$ *Expr* - *Term*		2,3	reduce 2
$ *Expr*		0,1	reduce 0
$ *Goal*		none	accept

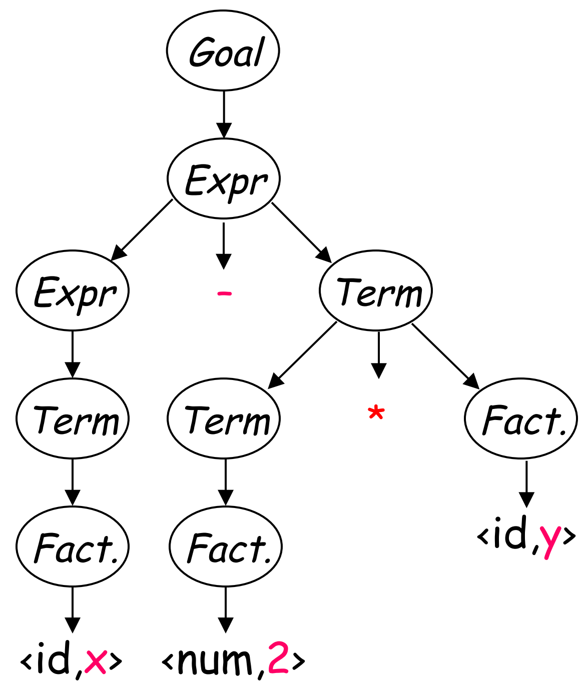
0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	number
8		|	id
9		|	(*Expr*)

5 shifts +
9 reduces +
1 accept

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce

Back to x - 2 * y

Stack	Input	Action
$	id - num * id	*shift*
$ id	- num * id	*reduce 8*
$ *Factor*	- num * id	*reduce 6*
$ *Term*	- num * id	*reduce 3*
$ *Expr*	- num * id	*shift*
$ *Expr* -	num * id	*shift*
$ *Expr* - num	* id	*reduce 7*
$ *Expr* - *Factor*	* id	*reduce 6*
$ *Expr* - *Term*	* id	*shift*
$ *Expr* - *Term* *	id	*shift*
$ *Expr* - *Term* * id		*reduce 8*
$ *Expr* - *Term* * *Factor*		*reduce 4*
$ *Expr* - *Term*		*reduce 2*
$ *Expr*		*reduce 0*
$ *Goal*		*accept*



Corresponding Parse Tree

LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- The class of grammars that these parsers recognize is called the set of LR(1) grammars

Informal definition:

A grammar is LR(1) if, given a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

We can

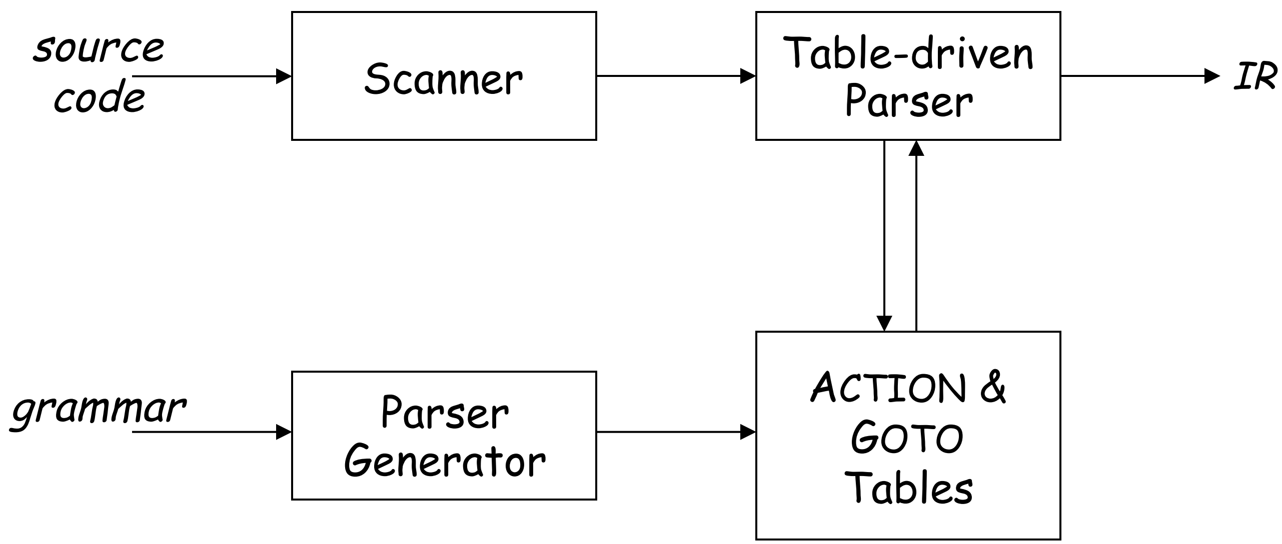
1. *isolate the handle of each right-sentential form* γ_i, and
2. *determine the production by which to reduce,*

by scanning γ_i from *left-to-right*, going at most *1* symbol beyond the right end of the handle of γ_i

LR(1) means left-to-right scan of the input, rightmost derivation (in reverse), and 1 word of lookahead.

LR(1) Parsers

A table-driven LR(1) parser looks like



Tables *can* be built by hand

However, this is a perfect task to automate

LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(s0);                    // initial state
token = scanner.next_token();
loop forever {
      s = stack.top();
      if ( ACTION[s,token] == "reduce A→β" ) then {
         stack.popnum(2*|β|);      // pop 2*|β| symbols
         s = stack.top();
         stack.push(A);            // push A
         stack.push(GOTO[s,A]);    // push next state
      }
      else if ( ACTION[s,token] == "shift si" ) then {
            stack.push(token); stack.push(si);
            token ← scanner.next_token();
      }
      else if ( ACTION[s,token] == "accept"
                        & token == EOF )
            then break;
      else throw a syntax error;
}
report success;
```

The skeleton parser

- relies on a stack & a scanner
- uses two tables, called ACTION & GOTO

  ACTION: state x word → state

  GOTO: state x *NT* → state
- shifts |*words*| times
- reduces |derivation| times
- accepts at most once
- detects errors by failure of the other three cases
- follows basic scheme for shift-reduce parsing from last lecture

To make a parser for *L(G)*, need a set of tables

Remember, this is the left-recursive SheepNoise; EaC shows the right-recursive version.

The grammar

1	*Goal*	→	*SheepNoise*
2	*SheepNoise*	→	*SheepNoise* baa
3		|	baa

The tables

ACTION Table		
State	EOF	baa
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 1

The string baa

Stack	Input	Action
\$ s_0	baa EOF	*shift 2*
\$ s_0 baa s_2	EOF	*reduce 3*
\$ s_0 *SN* s_1	EOF	*accept*

1	*Goal*	→	*SheepNoise*
2	*SheepNoise*	→	*SheepNoise* baa
3		|	baa

ACTION Table		
State	EOF	baa
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

Stack	Input	Action
$ s_0	baa baa EOF	*shift 2*
$ s_0 baa s_2	baa EOF	*reduce 3*
$ s_0 *SN* s_1	baa EOF	*shift 3*
$ s_0 *SN* s_1 baa s_3	EOF	*reduce 2*
$ s_0 *SN* s_1	EOF	*accept*

1	*Goal*	→	*SheepNoise*
2	*SheepNoise*	→	*SheepNoise* baa
3		|	baa

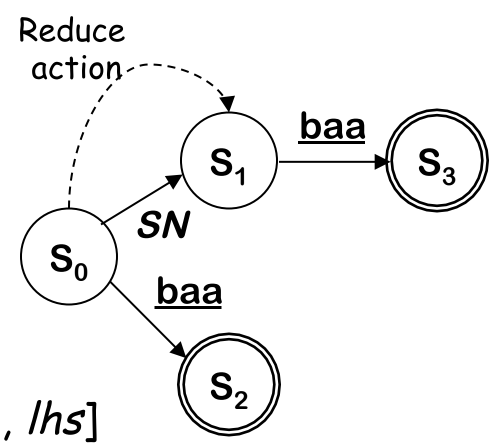
ACTION Table		
State	EOF	baa
0	—	*shift 2*
1	*accept*	*shift 3*
2	*reduce 3*	*reduce 3*
3	*reduce 2*	*reduce 2*

GOTO Table	
State	*SheepNoise*
0	1
1	0
2	0
3	0

LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - All active handles include top of stack (TOS)
  - Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - Build a handle-recognizing DFA
  - ACTION & GOTO tables encode the DFA
- To match subterm, invoke subterm DFA & leave old DFA's state on stack
- Final state in DFA ⇒ a *reduce* action
  - New state is GOTO[state at TOS (after pop), *lhs*]
  - For *SN*, this takes the DFA to s_1



Control DFA for SN

The Parentheses Language

Language of balanced parentheses

- Beyond power of REs
- Exhibits role of context in LR(1) parsing

0	*Goal*	→	*List*
1	*List*	→	*List Pair*
2		|	*Pair*
3	*Pair*	→	(*Pair*)
4		|	( )

The Parentheses Language

ACTION TABLE			
State	eof	(	)
0		S 3	
1	*acc*	S 3	
2	R 2	R 2	
3		S 6	S 7
4	R 1	R 1	
5			S 8
6		S 6	S 10
7	R 4	R 4	
8	R 3	R 3	
9			S 11
10			R 4
11			R 3

GOTO TABLE		
State	List	Pair
0	1	2
1		4
2		
3		5
4		
5		
6		9
7		
8		
9		
10		
11		

0	*Goal*	→	List
1	List	→	List Pair
2		|	Pair
3	Pair	→	(Pair)
4		|	( )

The Parentheses Language

State	Lookahead	Stack	Handle	Action
—	(	$ 0	—*none*—	—
0	(	$ 0	—*none*—	*shift 3*
3	)	$ 0 (3	—*none*—	*shift 7*
7	eof	$ 0 (3) 7	( )	*reduce 4*
2	eof	$ 0 *Pair* 2	*Pair*	*reduce 2*
1	eof	$ 0 *List* 1	*List*	*accept*

Parsing "( )"

0	Goal	→	List
1	List	→	List Pair
2		|	Pair
3	Pair	→	(Pair)
4		|	( )

0	Goal	→	List
1	List	→	List Pair
2		|	Pair
3	Pair	→	(Pair)
4		|	( )

The Parentheses Language

State	L'ahead	Stack	Handle	Action
—	(	$ 0	—none—	—
0	(	$ 0	—none—	*shift 3*
3	(	$ 0 (3	—none—	*shift 6*
6	)	$ 0 (3 (6	—none—	shift 10
10	)	$ 0 (3 (6) 10	( )	*reduce 4*
5	)	$ 0 (3 *Pair* 5	—none—	shift 8
8	(	$ 0 (3 *Pair* 5) 8	(*Pair*)	*reduce 3*
2	(	$ 0 *Pair* 2	*Pair*	*reduce 2*
1	(	$ 0 *List* 1	—none—	*shift 3*
3	)	$ 0 *List* 1 (3	—none—	*shift 7*
7	eof	$ 0 *List* 1 (3) 7	( )	*reduce 4*
4	eof	$ 0 *List* 1 *Pair* 4	*List Pair*	*reduce 1*
1	eof	$ 0 *List* 1	*List*	*accept*

Parsing
"(( )) ( )"

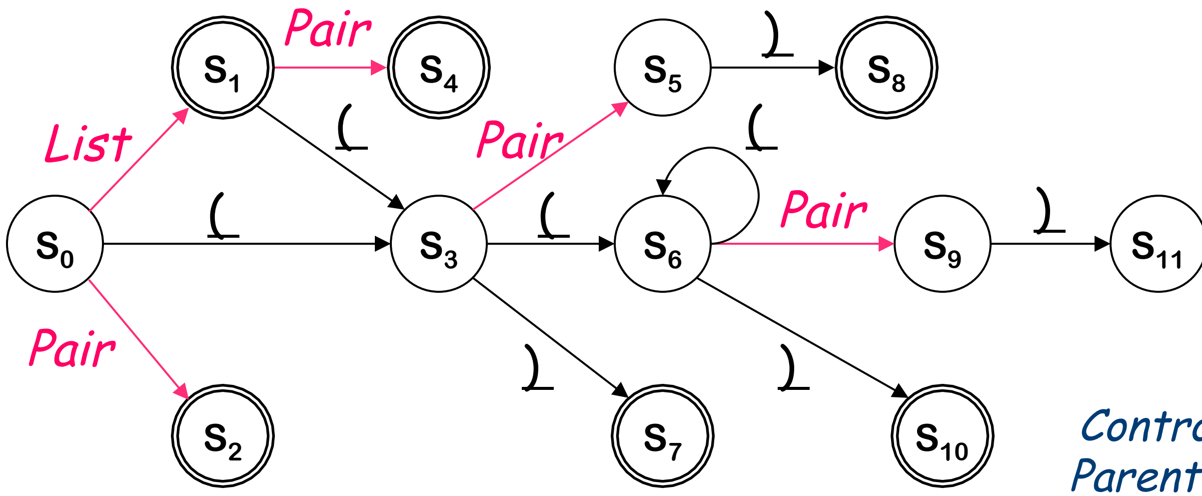
LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - All active handles include top of stack (TOS)
  - Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - Build a handle-recognizing DFA to control the stack-based recognizer
  - ACTION & GOTO tables encode the DFA
- To match a subterm, invoke the DFA recursively
  - leave old DFA's state on stack and go on
- Final state in DFA ⇒ a *reduce* action
  - Pop rhs off the stack to reveal invoking state
    - → "It would be legal to recognize an *x*, and we did …"
  - New state is GOTO[revealed state, *lhs*]
  - Take a DFA transition on the new *NT* — the lhs we just pushed…

LR(1) Parsers

The Control DFA for the Parentheses Language



Control DFA for the Parentheses Language

Transitions on terminals represent shift actions [ACTION]

Transitions on nonterminals represent reduce actions [GOTO]

The table construction derives this DFA from the grammar

Building LR(1) Tables

Slides removed for time

Summary

	Advantages	*Disadvantages*
Top-down recursive descent	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