CSCI312 Principles of Programming Languages

LL Parsing

Xu Liu

Derived from Keith Cooper's COMP 412 at Rice University

Recap

Syntactic Analysis

Phase also known as: parser

Purpose is to recognize source structure

Input: tokens

Output: parse tree or abstract syntax tree

A recursive descent parser is one in which each nonterminal in the grammar is converted to a function which recognizes input derivable from the nonterminal.

Terms

Nullable non-terminals

First set

T(EBNF) = Code: A → w

1 If w is nonterminal, call it.

2 If w is terminal, match it against given token.

3 If w is { w' }:

while (token in First(w')) T(w')

4 If w is: w1 | ... | wn,

switch (token) {

case First(w1): T(w1); break;

...

case First(wn): T(wn); break;

5 Switch (cont.): If some wi is empty, use:

default: break;

Otherwise

default: error(token);

6 If w = [w'], rewrite as (| w') and use rule 4.

7 If w = X1 ... Xn, T(w) =

T(X1); ... T(Xn);

Outline

See more general problems in a top down parser

- Backtracking — select appropriate productions
- Left recursion — revise grammars
- Predictive parsing — more than recursive descent
- Table-driven 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"

— from last lecture

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

Let's try x – 2 * y :

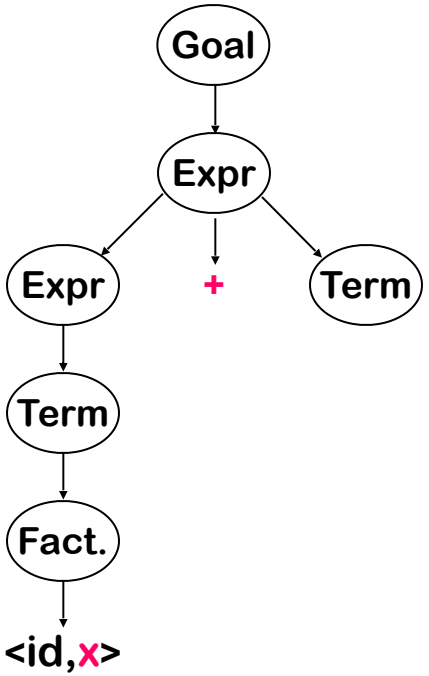
Rule	Sentential Form	Input
—	Goal	↑x - 2 * y

↑ is the position in the input buffer

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 "+"

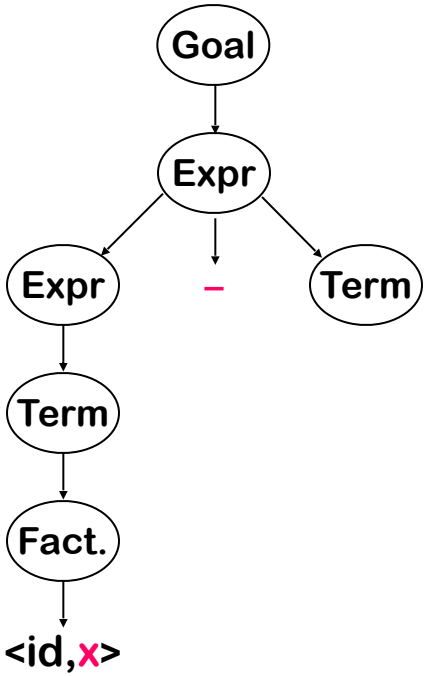
The parser must backtrack to here

↑ 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

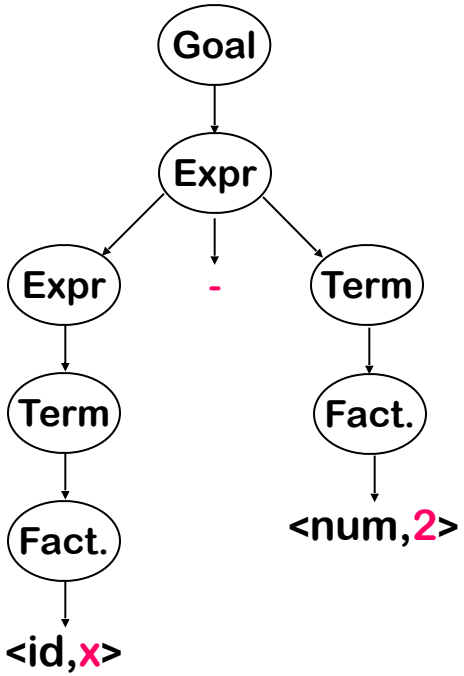
Now we can expand Term to match "2"

⇒ 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?

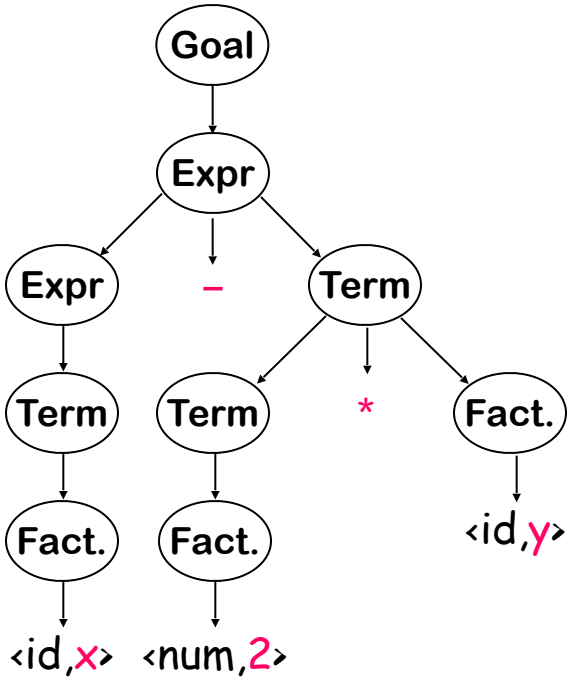
- "2" matches "2"
- We have more input, but no NTs left to expand
- The expansion terminated too soon

⇒ 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 always 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 → Fee α

| β

where neither α nor β start with Fee

We can rewrite this fragment as

Fee → β Fie

Fie → α Fie

| ε

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

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

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

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

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, $\underline{x} \in \text{FIRST}(\alpha)$ 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?

$\Rightarrow$ They complicate the definition of LL(1)

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

$\text{FOLLOW}(A)$ = the set of terminal symbols that can immediately follow A in a sentential form

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

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

Then, a grammar is LL(1) iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

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

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	→	(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.

What If My Grammar Is Not LL(1) ?

Can we transform a non-LL(1) grammar into an LL(1) grammar?

- In general, the answer is no
- In some cases, however, the answer is yes

Assume a grammar G with productions $A \rightarrow \alpha\,\beta_1$ and $A \rightarrow \alpha\,\beta_2$

- If α derives anything other than ε, then

$$\text{FIRST}^{+}(A \rightarrow \alpha\,\beta_1) \cap \text{FIRST}^{+}(A \rightarrow \alpha\,\beta_2) \neq \varnothing$$

- And the grammar is not LL(1)

If we pull the common prefix, α, into a separate production, we may make the grammar LL(1).

$$A \rightarrow \alpha\,A',\ A' \rightarrow \beta_1 \text{ and } A' \rightarrow \beta_2$$

Now, if $\text{FIRST}^{+}(A' \rightarrow \beta_1) \cap \text{FIRST}^{+}(A' \rightarrow \beta_2) = \varnothing$, G may be LL(1)

What If My Grammar Is Not LL(1) ?

Left Factoring

For each nonterminal A

find the longest prefix α common to 2 or more alternatives for A

if $\alpha \neq \varepsilon$ then

replace all of the productions

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \alpha\beta_n \mid \gamma$

with

$A \rightarrow \alpha A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

Repeat until no nonterminal has alternative rhs' with a common prefix

This transformation makes some grammars into LL(1) grammars

There are languages for which no LL(1) grammar exists

Left Factoring Example

Consider a simple right-recursive expression grammar

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

To choose between 1, 2, & 3, an LL(1) parser must look past the number or id to see the operator.

$$\text{FIRST}^+(1) = \text{FIRST}^+(2) = \text{FIRST}^+(3)$$

and

$$\text{FIRST}^+(4) = \text{FIRST}^+(5) = \text{FIRST}^+(6)$$

Let's left factor this grammar.

Left Factoring Example

After Left Factoring, we have

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

This transformation makes some grammars into LL(1) grammars.

There are languages for which no LL(1) grammar exists.

FIRST and FOLLOW Sets

FIRST(α)

For some $\alpha \in (T \cup NT)^*$, 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

FOLLOW(*A*)

For some $A \in NT$, define FOLLOW(*A*) as the set of symbols that can occur immediately after *A* in a valid sentential form

FOLLOW(S) = {EOF}, where S is the start symbol

To build FOLLOW sets, we need FIRST sets …

Computing FIRST Sets

Already studied in previous lectures

Computing FOLLOW Sets

for each $A \in NT$, $FOLLOW(A) \leftarrow \emptyset$

$FOLLOW(S) \leftarrow \{EOF\}$

while (FOLLOW sets are still changing)
    for each $p \in P$, of the form $A \rightarrow B_1 B_2 \ldots B_k$
        $TRAILER \leftarrow FOLLOW(A)$
        for $i \leftarrow k$ down to 1
            if $B_i \in NT$ then // domain check
                $FOLLOW(B_i) \leftarrow FOLLOW(B_i) \cup TRAILER$
                if $\varepsilon \in FIRST(B_i)$ // add right context
                  then $TRAILER \leftarrow TRAILER \cup (FIRST(B_i) - \{\varepsilon\})$
                  else $TRAILER \leftarrow FIRST(B_i)$ // no ε => no right context
            else $TRAILER \leftarrow \{B_i\}$ // $B_i \in T$ => only 1 symbol

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
7		|	/ Factor
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}

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
7		|	/ Factor
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	(

Building Top-down Parsers for LL(1) Grammars

Given an LL(1) grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs's
  - Each returns true on success and throws an error on false
  - Simple, working (perhaps ugly) code
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

I don't know of a system that does this …

Building Top-down Parsers

Strategy

- Encode knowledge in a table
- Use a standard "skeleton" parser to interpret the table

Example

- The non-terminal Factor has 3 expansions
  - (Expr) or Identifier or Number
- Table might look like:

		+	-	*	/	Id.	Num.	(	)	EOF
Non-terminal Symbols	Factor	—	—	—	—	10	9	11	—	—

Terminal Symbols

Cannot expand Factor into an operator ⇒ error

Expand Factor by rule 9 with input "number"

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)

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T

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

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) 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
```

Building Top-down Parsers

Building the complete table

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

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

1. entry is the rule $X \rightarrow \beta$, if $y \in FIRST^{+}(X \rightarrow \beta)$

— entry is error if rule 1 does not define

If any entry has more than one rule, G is not LL(1)

We call this algorithm the LL(1) table construction algorithm