Compiler Construction 2016/2017
Syntax Analysis

Peter Thiemann

November 2, 2016

Outline

1. Syntax Analysis
   - Recursive top-down parsing
   - Nonrecursive top-down parsing
   - Bottom-up parsing

Syntax Analysis

```
tokens --> [parser] --> IR
```

Parser

- recognize context-free syntax
- guide context-sensitive analysis
- construct IR(s)
- produce meaningful error messages
- attempt error correction

Syntax/Expressions

An expression grammar in BNF

```
1 <goal> ::= <expr>
2 <expr> ::= <expr> <op> <expr>
3          | num
4          | id
5 <op>   ::= +
6          | -
7          | *
8          | /
```

Simple expressions with numbers and identifiers

Derivation

- To derive a word/sentence from a BNF, we start with the goal variable
- In each derivation step, we replace a variable with the right hand side of one of its rules
- leftmost derivation: choose the leftmost variable in each step
- rightmost derivation: choose the rightmost variable in each step
- parsing is the discovery of a derivation:
  given a sentence, find a derivation from the goal variable that produces this sentence

Derivation, formally

- We write α, β, γ for (possibly empty) strings of terminal symbols and variables
- If *N* is a variable and *N* $::= \beta$ is a rule for *N*, then we write $\alpha N \gamma \Rightarrow \alpha\beta\gamma$ for a single derivation step
- We write $\alpha \Rightarrow^* \beta$ if there is a (possibly empty) sequence of derivation steps from α to β
- We write $\alpha \Rightarrow^+ \beta$ if there is a non-empty sequence of derivation steps from α to β

Example: `x + 2 * y`

- Compare leftmost derivation with rightmost derivation

Grammar with built-in precedence

G-ETF

```
1 <goal>    ::= <expr>
2 <expr>    ::= <expr> + <term>
3             | <expr> - <term>
4             | <term>
5 <term>    ::= <term> * <factor>
6             | <term> / <factor>
7             | <factor>
8 <factor> ::= num
9             | id
```

- Enforces precedence

Example: `x + 2 * y` with precedence

- Only one possible derivation

Ambiguity

If a grammar has more than one derivation for a single sentence, then it is ambiguous
Example:

```
<stmt> ::= if <expr> then <stmt>
         | if <expr> then <stmt> else <stmt>
         | other stmts
```

Consider deriving the sentence:

```
if E1 then if E2 then S1 else S2
```

It has two derivations.

Ambiguity fixed

May be able to eliminate ambiguities by rearranging the grammar:

```
<stmt> ::= <matched>
         | <unmatched>
<matched> ::= if <expr> then <matched> else <matched>
            | other stmts
<unmatched> ::= if <expr> then <stmt>
              | if <expr> then <matched> else <unmatched>
```

This grammar generates the same language as the ambiguous grammar, but applies the common sense rule:
match each `else` with the closest unmatched `then`
Generally accepted resolution of this ambiguity

Two approaches to parsing

Top-down parsing

- start at the root of the derivation tree
- extend derivation by “guessing” the next production
- may require backtracking

Bottom-up parsing

- start at the leaves of the derivation tree
- combine tree fragments to larger fragments
- bookkeeping of successful fragments (by finite automaton)

Outline

1. Syntax Analysis
   - Recursive top-down parsing
   - Nonrecursive top-down parsing
   - Bottom-up parsing

Top-down parsing / Recursive descent

Approach

- Define a procedure for each syntactic variable
- The procedure for N
  - consumes token sequences derivable from N
  - has an alternative path for each rule for N
- The code for a rule $N ::= \alpha$ consists of a code fragment for each symbol in α
  - consider the symbols in α from left to right
  - a terminal symbol consumes the symbol
  - a variable N' calls the procedure for N'

Recursive descent for **G-ETF**

```
// <factor> ::= NUM | ID

IR factor() {
  if( cur_token == NUM ) {
    next_token(); return IR_NUM;
  }
  if( cur_token == ID  ) {
    next_token(); return IR_ID;
  }
  syntax_error("NUM or ID expected");
}
```

Recursive descent for **G-ETF**/2

```
/* <term>    ::= <term> * <factor>
               | <term> / <factor>
               | <factor>
**/

IR term() {
  if( ??? ) {
    term(); check(MUL); factor();
    return IR_MUL(...);
  }
  if( ??? ) {
    term(); check(DIV); factor();
    return IR_DIV(...);
  }
  return factor();
}
```

Trouble with recursive descent / top-down parsing

Alternatives

- How do we decide on one of the productions?
- Would it help to look at the next token?

Trouble with recursive descent / top-down parsing

Alternatives

- How do we decide on one of the productions?
- Would it help to look at the next token?

Left recursion

- If we choose the first or second rule, we end up in an infinite recursion without consuming input.

Fact: Top-down parsers cannot deal with left recursion

- A variable N in a BNF is left recursive, if there is a derivation such that $N \Rightarrow^{+} N\alpha$, where α is an arbitrary string of variables and terminals.

Fact: Top-down parsers cannot deal with left recursion

- A variable N in a BNF is left recursive, if there is a derivation such that $N \Rightarrow^{+} N\alpha$, where α is an arbitrary string of variables and terminals.
- Fact: if a BNF has a left recursive variable, then top-down parsing for this BNF may not terminate.

Fact: Top-down parsers cannot deal with left recursion

- A variable N in a BNF is left recursive, if there is a derivation such that $N \Rightarrow^{+} N\alpha$, where α is an arbitrary string of variables and terminals.
- Fact: if a BNF has a left recursive variable, then top-down parsing for this BNF may not terminate.
- Cure: remove left recursion by transforming the BNF

Elimination of direct left recursion

- Suppose G has a left recursive variable A with rules

$$A \to A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

  where none of the β_i starts with A and all $\alpha_j \neq \varepsilon$
- Add a new variable A'
- Remove all productions for A
- Add new productions

$$A \to \beta_1 A' \mid \cdots \mid \beta_m A'$$
$$A' \to \varepsilon \mid \alpha_1 A' \mid \cdots \mid \alpha_n A'$$

- This transformation is always possible: same language, but A no longer left recursive

Example: transforming **G-ETF**

Original grammar G-ETF

```
<goal>   ::= <expr>
<expr>   ::= <expr> + <term>
           | <expr> - <term>
           | <term>

<term>   ::= <term> * <factor>
           | <term> / <factor>
           | <factor>

<factor> ::= num
           | id
```

→

After elimination of left recursion

```
<goal>   ::= <expr>
<expr>   ::= <term> <expr'>
<expr'>  ::= + <term> <expr'>
           | - <term> <expr'>
           |
<term>   ::= <factor> <term'>
<term'>  ::= * <factor> <term'>
           | / <factor> <term'>
           |
<factor> ::= num
           | id
```

Twist: different derivation trees

- elimination of left recursion changes the derivation trees
- consider `10 - 2 + 3`

What about alternatives?

```
/* <term'>  ::= * <factor> <term'>
              | / <factor> <term'>
              |
**/
IR term1(IR arg1) {
  if( cur_token == MUL ) {
    next_token(); IR arg2 = factor();
    return term1(IR_MUL(arg1, arg2));
  }
  if( cur_token == DIV ) {
    next_token(); IR arg2 = factor();;
    return term1(IR_DIV(arg1, arg2));
  }
  if ( cur_token ??? ) {
    return arg1;
  }
  syntax_error ("MUL, DIV, or ??? expected");
}
```

How do we select between several alternative rules?

- Solution: lookahead — what is the next token?

How do we select between several alternative rules?

- Solution: lookahead — what is the next token?
- But how do we determine lookahead symbols?

How do we select between several alternative rules?

- Solution: lookahead — what is the next token?
- But how do we determine lookahead symbols?

First symbols

For each right hand side α of a rule and $k > 0$, we want to determine

$$FIRST_k(\alpha) = \{w|_k \mid \alpha \Rightarrow^* w\}$$

How do we select between several alternative rules?

- Solution: lookahead — what is the next token?
- But how do we determine lookahead symbols?

First symbols

For each right hand side α of a rule and $k > 0$, we want to determine

$$FIRST_k(\alpha) = \{w|_k \mid \alpha \Rightarrow^* w\}$$

k-Cutoff of a word

$$w|_k = \begin{cases} w & |w| \leq k \\ w_1 & w = w_1 w_2, |w_1| = k \end{cases}$$

First symbols

Computing first symbols

$$
\begin{aligned}
FIRST_k(\varepsilon) &= \{\varepsilon\} \\
FIRST_k(a\alpha) &= \{(aw)|_k \mid w \in FIRST_k(\alpha)\} \\
FIRST_k(N\alpha) &= \{(vw)|_k \mid v \in FIRST_k(N), w \in FIRST_k(\alpha)\}
\end{aligned}
$$

First symbols

Computing first symbols

$$\begin{aligned}
FIRST_k(\varepsilon) &= \{\varepsilon\} \\
FIRST_k(a\alpha) &= \{(aw)|_k \mid w \in FIRST_k(\alpha)\} \\
FIRST_k(N\alpha) &= \{(vw)|_k \mid v \in FIRST_k(N), w \in FIRST_k(\alpha)\}
\end{aligned}$$

Computing first symbols for N

$$FIRST_k(N) = \bigcup\{FIRST_k(\alpha) \mid N ::= \alpha \text{ is a rule}\}$$

Example G-ETF/LR and $k = 1$

```
<goal>   ::= <expr>
<expr>   ::= <term> <expr'>
<expr'>  ::= + <term> <expr'>
           | - <term> <expr'>
           |
<term>   ::= <factor> <term'>
<term'>  ::= * <factor> <term'>
           | / <factor> <term'>
           |
<factor> ::= num
           | id
```

$$FIRST_1(G) = FIRST_1(E)$$

$$FIRST_1(E) = FIRST_1(T) \odot FIRST_1(E')$$

$$FIRST_1(E') = \{+, -, \varepsilon\}$$

$$FIRST_1(T) = FIRST_1(F) \odot FIRST_1(T')$$

$$FIRST_1(T') = \{*, /, \varepsilon\}$$

$$FIRST_1(F) = \{\mathtt{num}, \mathtt{id}\}$$

Example G-ETF/LR and $k = 1$

```
<goal>   ::= <expr>
<expr>   ::= <term> <expr'>
<expr'>  ::= + <term> <expr'>
           | - <term> <expr'>
           |
<term>   ::= <factor> <term'>
<term'>  ::= * <factor> <term'>
           | / <factor> <term'>
           |
<factor> ::= num
           | id
```

$$FIRST_1(G) = FIRST_1(E)$$
$$FIRST_1(E) = FIRST_1(T) \odot FIRST_1(E')$$

$$FIRST_1(E') = \{+, -, \varepsilon\}$$
$$FIRST_1(T) = FIRST_1(F) \odot FIRST_1(T') = \{\mathtt{num}, \mathtt{id}\}$$
$$FIRST_1(T') = \{*, /, \varepsilon\}$$
$$FIRST_1(F) = \{\mathtt{num}, \mathtt{id}\}$$

Example G-ETF/LR and $k = 1$

```
<goal>   ::= <expr>
<expr>   ::= <term> <expr'>
<expr'>  ::= + <term> <expr'>
           | - <term> <expr'>
           |
<term>   ::= <factor> <term'>
<term'>  ::= * <factor> <term'>
           | / <factor> <term'>
           |
<factor> ::= num
           | id
```

$$\begin{aligned}
FIRST_1(G) &= FIRST_1(E) \\
FIRST_1(E) &= FIRST_1(T) \odot FIRST_1(E') \\
&= \{\mathtt{num}, \mathtt{id}\} \\
FIRST_1(E') &= \{+, -, \varepsilon\} \\
FIRST_1(T) &= FIRST_1(F) \odot FIRST_1(T') \\
&= \{\mathtt{num}, \mathtt{id}\} \\
FIRST_1(T') &= \{*, /, \varepsilon\} \\
FIRST_1(F) &= \{\mathtt{num}, \mathtt{id}\}
\end{aligned}$$

Lookahead, 1. Attempt

- If $A ::= \alpha_1 \mid \cdots \mid \alpha_n$ is the list of all rules for A, then the first-sets of all right hand sides must be disjoint:

$$\forall i \neq j : FIRST_1(\alpha_i) \neq FIRST_1(\alpha_j)$$

- On input aw, the parser for N chooses the right hand side i with $a \in FIRST_1(\alpha_i)$
- Signal syntax error, if no such i exists.

Another example for FIRST

$$
\begin{array}{ll}
A ::= B \mid Cx \mid \varepsilon & FIRST_1(A) = FIRST_1(B) \cup FIRST_1(C) \odot \{x\} \cup \{\varepsilon\} \\
B ::= C \mid yA & FIRST_1(B) = FIRST_1(C) \cup \{y\} \\
C ::= B \mid z & FIRST_1(C) = FIRST_1(B) \cup \{z\}
\end{array}
$$

Another example for FIRST

$$A ::= B \mid Cx \mid \varepsilon \qquad FIRST_1(A) = FIRST_1(B) \cup FIRST_1(C) \odot \{x\} \cup \{\varepsilon\}$$

$$B ::= C \mid yA \qquad FIRST_1(B) = FIRST_1(C) \cup \{y\}$$

$$C ::= B \mid z \qquad FIRST_1(C) = FIRST_1(B) \cup \{z\}$$

Computing FIRST by fixpoint

	A	B	C	
0	–	–	–	
1	ε	y	z	
2	y, z, ε	z, y	y, z	
3	y, z, ε	z, y	y, z	no further change, fixpoint reached

Epsilon rules and FOLLOW

- What if there is a rule $A ::= \varepsilon$?
- As $\mathit{FIRST}(\varepsilon) = \varepsilon$, this rule is always applicable!

Epsilon rules and FOLLOW

- What if there is a rule $A ::= \varepsilon$?
- As $FIRST(\varepsilon) = \varepsilon$, this rule is always applicable!

Solution

Consider the symbols that can possibly follow A!

$$FOLLOW_k(A) = \{w|_k \mid S \Rightarrow^* \alpha A w\}$$

Computing FOLLOW

$$FOLLOW_k(A) = \bigcup \{FIRST_k(\beta) \odot_k FOLLOW_k(B) \mid B ::= \alpha A \beta \text{ is a rule}\}$$

We assume that the goal variable G does not appear on the right hand side of productions and that $FOLLOW_k(G) = \{\varepsilon\}$

Example G-ETF/LR and $FOLLOW_1$

```
<goal>   ::= <expr>
<expr>   ::= <term> <expr'>
<expr'>  ::= + <term> <expr'>
           | - <term> <expr'>
           |
<term>   ::= <factor> <term'>
<term'>  ::= * <factor> <term'>
           | / <factor> <term'>
           |
<factor> ::= num
           | id
```

$$FOLLOW_1(G) = \{\varepsilon\}$$
$$FOLLOW_1(E) = FOLLOW_1(G)$$

$$FOLLOW_1(E') = FOLLOW_1(E)$$

$$FOLLOW_1(T) = FIRST_1(E') \odot (FOLLOW_1(E) \cup FOLLOW_1(E'))$$

$$FOLLOW_1(T') = FOLLOW_1(T)$$

$$FOLLOW_1(F) = FIRST_1(T') \odot (FOLLOW_1(T) \cup FOLLOW_1(T'))$$

Example G-ETF/LR and $FOLLOW_1$

```
<goal>    ::= <expr>
<expr>    ::= <term> <expr'>
<expr'>   ::= + <term> <expr'>
            | - <term> <expr'>
            |
<term>    ::= <factor> <term'>
<term'>   ::= * <factor> <term'>
            | / <factor> <term'>
            |
<factor> ::= num
            | id
```

$$FOLLOW_1(G) = \{\varepsilon\}$$

$$FOLLOW_1(E) = FOLLOW_1(G) = \{\varepsilon\}$$

$$FOLLOW_1(E') = FOLLOW_1(E) = \{\varepsilon\}$$

$$FOLLOW_1(T) = FIRST_1(E') \odot (FOLLOW_1(E) \cup FOLLOW_1(E')) = \{+, -, \varepsilon\} \odot FOLLOW_1(E')$$

$$FOLLOW_1(T') = FOLLOW_1(T)$$

$$FOLLOW_1(F) = FIRST_1(T') \odot (FOLLOW_1(T) \cup FOLLOW_1(T'))$$

Example G-ETF/LR and $FOLLOW_1$

```
<goal>   ::= <expr>
<expr>   ::= <term> <expr'>
<expr'>  ::= + <term> <expr'>
           | - <term> <expr'>
           |
<term>   ::= <factor> <term'>
<term'>  ::= * <factor> <term'>
           | / <factor> <term'>
           |
<factor> ::= num
           | id
```

$$
\begin{aligned}
FOLLOW_1(G) &= \{\varepsilon\} \\
FOLLOW_1(E) &= FOLLOW_1(G) \\
&= \{\varepsilon\} \\
FOLLOW_1(E') &= FOLLOW_1(E) \\
&= \{\varepsilon\} \\
FOLLOW_1(T) &= FIRST_1(E') \odot (FOLLOW_1(E) \cup FOLLOW_1(E')) \\
&= \{+, -, \varepsilon\} \odot FOLLOW_1(E') \\
&= \{+, -, \varepsilon\} \\
FOLLOW_1(T') &= FOLLOW_1(T) \\
&= \{+, -, \varepsilon\} \\
FOLLOW_1(F) &= FIRST_1(T') \odot (FOLLOW_1(T) \cup FOLLOW_1(T')) \\
&= \{*, /, \varepsilon\} \odot \{+, -, \varepsilon\} \\
&= \{*, /, +, -, \varepsilon\}
\end{aligned}
$$

Lookahead, 2. Attempt

Lookahead set for a rule

$$LA_k(A ::= \alpha) = FIRST_k(\alpha) \odot_k FOLLOW_k(A)$$

Definition LL(k) grammar

A BNF is an LL(k) grammar, if for each variable A its rule set $A ::= \alpha_1 \mid \cdots \mid \alpha_n$ fulfills

$$\forall i \neq j : LA_k(A ::= \alpha_i) \cap LA_k(A ::= \alpha_j) = \emptyset$$

LL(k) Parsing

On input w, the parser for A chooses the unique right hand side α_i such that $w|_k \in LA_k(A ::= \alpha_i)$ and signals a syntax error if no such i exists.

Outline

1. Syntax Analysis
   - Recursive top-down parsing
   - Nonrecursive top-down parsing
   - Bottom-up parsing

Nonrecursive top-down parsing

- A top-down parser may be implemented without recursion using a pushdown automaton
- The pushdown keeps track of the terminals and variables that still need to be matched

The pushdown automaton

- $Q = \{q\}$ a single state, also serves as initial state
- Σ is the input alphabet
- $\Gamma = \Sigma \cup VAR$, the set of terminals and variables, is the pushdown alphabet
- $Z_0 = G \in VAR$, the pushdown bottom symbol is the goal variable
- δ is defined by
  - $\delta(q, \varepsilon, A) \ni (q, \alpha)$ if $A ::= \alpha$ is a rule
  - $\delta(q, a, a) = (q, \varepsilon)$
- In the PDA, the choice of the rule is nondeterministic, in practice we disambiguate using LL(k) grammars with lookahead

Nonrecursive top-down parser

with parse table *M*

```
push EOF
push Start_Symbol
token ← next_token()
repeat
  X ← Stack[tos]
  if X is a terminal or EOF then
    if X = token then
      pop X
      token ← next_token()
    else error()
  else /* X is a variable */
    if M[X,token] = X ::= Y1Y2 ... Yk then
      pop X
      push Yk,...,Y2,Y1
    else error()
until X = EOF
```

Parse table

The parse table *M* maps a variable and a lookahead symbol (from the input) to a production (number)

```
1  <goal>   ::= <expr>
2  <expr>   ::= <term> <expr'>
3  <expr'>  ::= + <term> <expr'>
4             | - <term> <expr'>
5             |
6  <term>   ::= <factor> <term'>
7  <term'>  ::= * <factor> <term'>
8             | / <factor> <term'>
9             |
10 <factor> ::= id
11            | num
```

	id	num	+	−	*	/	$
G	1	1					
E	2	2					
E'			3	4			5
T	6	6					
T'			9	9	7	8	9
F	10	11					

LL(1) parse table construction

Input: a BNF
Output: parsing table M
Algorithm:

1. For all rules p of the form $A ::= \alpha$
    1. For each $a \in FIRST_1(\alpha)$: add p to $M[A, a]$
    2. If $\varepsilon \in FIRST_1(\alpha)$
        1. For each $b \in FOLLOW_1(A)$: add p to $M[A, b]$
        2. If $\varepsilon \in FOLLOW_1(A)$: add p to $M[A, \varepsilon]$
2. Set each undefined entry of M to `error`

Check: The BNF is LL(1) if $|M[A, a]| \leq 1$ for all A and a.

A BNF that is not LL(1)

```
<stmt> ::= if <expr> then <stmt>
         | if <expr> then <stmt> else <stmt>
```

Both rules have the same FIRST sets because their right hand sides have the same prefix!

A BNF that is not LL(1)

```
<stmt> ::= if <expr> then <stmt>
         | if <expr> then <stmt> else <stmt>
```

Both rules have the same FIRST sets because their right hand sides have the same prefix!

Left factorization

- Coalesce to a new rule
- Introduce new variable that derives the different suffixes

```
<stmt>  ::= if <expr> then <stmt> <stmt'>
<stmt'> ::= else <stmt>
          |  /*empty*/
```

Still not LL(1) . . .

```
<stmt>   ::= if <expr> then <stmt> <stmt'>
<stmt'>  ::= else <stmt>
           |  /*empty*/
```

$$FIRST_1(S) = \{\texttt{if}\}$$
$$FIRST_1(S') = \{\texttt{else}, \varepsilon\}$$
$$FOLLOW_1(S) = FIRST_1(S') \cup FOLLOW_1(S')$$

$$FOLLOW_1(S') = FOLLOW_1(S)$$

Still not LL(1) . . .

```
<stmt>   ::= if <expr> then <stmt> <stmt'>
<stmt'>  ::= else <stmt>
           |   /*empty*/
```

$$
\begin{aligned}
FIRST_1(S) &= \{\texttt{if}\} \\
FIRST_1(S') &= \{\texttt{else}, \varepsilon\} \\
FOLLOW_1(S) &= FIRST_1(S') \cup FOLLOW_1(S') \\
&= \{\texttt{else}, \varepsilon\} \cup FOLLOW_1(S')
\end{aligned}
$$

$$FOLLOW_1(S') = FOLLOW_1(S)$$

Still not LL(1) . . .

```
<stmt>   ::= if <expr> then <stmt> <stmt'>
<stmt'>  ::= else <stmt>
           |  /*empty*/
```

$$
\begin{aligned}
\mathit{FIRST}_1(S) &= \{\texttt{if}\} \\
\mathit{FIRST}_1(S') &= \{\texttt{else}, \varepsilon\} \\
\mathit{FOLLOW}_1(S) &= \mathit{FIRST}_1(S') \cup \mathit{FOLLOW}_1(S') \\
&= \{\texttt{else}, \varepsilon\} \cup \mathit{FOLLOW}_1(S') \\
&= \{\texttt{else}, \varepsilon\} \\
\mathit{FOLLOW}_1(S') &= \mathit{FOLLOW}_1(S) \\
&= \{\texttt{else}, \varepsilon\}
\end{aligned}
$$

Still not LL(1) . . .

```
<stmt>  ::= if <expr> then <stmt> <stmt'>
<stmt'> ::= else <stmt>
          |  /*empty*/
```

$$
\begin{aligned}
FIRST_1(S) &= \{\texttt{if}\} \\
FIRST_1(S') &= \{\texttt{else}, \varepsilon\} \\
FOLLOW_1(S) &= FIRST_1(S') \cup FOLLOW_1(S') \\
&= \{\texttt{else}, \varepsilon\} \cup FOLLOW_1(S') \\
&= \{\texttt{else}, \varepsilon\} \\
FOLLOW_1(S') &= FOLLOW_1(S) \\
&= \{\texttt{else}, \varepsilon\} \\
LA_1(S' ::= \texttt{else}\ S) &= \{\texttt{else}\} \\
LA_1(S' ::= \varepsilon) &= \{\texttt{else}, \varepsilon\}
\end{aligned}
$$

Practical fix

The rule

```
<stmt'> ::= else <stmt>
```

gets precedence on input `else`.

Outline

1. Syntax Analysis
   - Recursive top-down parsing
   - Nonrecursive top-down parsing
   - Bottom-up parsing

Bottom-up parsing

Goal

Given an input string and a BNF, construct a parse tree starting at the leaves and working up to the root.

Recall

Sentential form

For a BNF with start variable S, every α such that $S \Rightarrow^* \alpha$ is a sentential form.
If $\alpha \in \Sigma^*$, then α is a sentence.
A left sentential form occurs in a left derivation.
A right sentential form occurs in a right derivation.

Bottom-up parsing

Procedure:

- The bottom-up parser repeatedly matches a right-sentential form of the language against the tree's upper frontier.
- At each match, it applies a reduction to build on the frontier:
  - each reduction matches an upper frontier of the partially built tree to the RHS of some production
  - each reduction adds a node on top of the frontier
- The final result is a rightmost derivation, in reverse.

Example

Consider the BNF

$$
\begin{array}{c|lcl}
1 & S & ::= & aABe \\
2 & A & ::= & Abc \\
3 & & | & b \\
4 & B & ::= & d
\end{array}
$$

and the input string

`abbcde`

(Construct a rightmost derivation)

Handles

To construct a the rightmost derivation, we need to find handles.

Handle

A handle is a substring α of the trees frontier that matches a rule $A ::= \alpha$ which is applied at that point in a rightmost derivation

Handles

To construct a the rightmost derivation, we need to find handles.

Handle

A handle is a substring α of the trees frontier that matches a rule $A ::= \alpha$ which is applied at that point in a rightmost derivation

Formally

- In a right-sentential form $\alpha\beta w$, the string β is a handle for production $A ::= \beta$
- if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$ then β is a handle for $A ::= \beta$ in $\alpha\beta w$

Example: G-ETF

```
1 <goal>    ::= <expr>
2 <expr>    ::= <expr> + <term>
3             | <expr> - <term>
4             | <term>
5 <term>    ::= <term> * <factor>
6             | <term> / <factor>
7             | <factor>
8 <factor> ::= num
9             | id
```

- A rightmost derivation for this grammar has unique handles.

Stack implementation

One scheme to implement a handle-pruning, bottom-up parser is called a shift-reduce parser.
Shift-reduce parsers use a stack and an input buffer

1. initialize stack with $\$$
2. Repeat until the top of the stack is the goal symbol and the input token is $\$$
   - find the handle
   - if we don't have a handle on top of the stack, shift an input symbol onto the stack
   - prune the handle if we have a handle for $A ::= \beta$ on the stack, reduce:
     - pop $|\beta|$ symbols off the stack
     - push A onto the stack

Example

Apply shift-reduce parsing to `x-2*y`

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

Shift-reduce parsing

A shift-reduce parser has just four canonical actions:

1. shift — next input symbol is shifted onto the top of the stack
2. reduce — right end of handle is on top of stack; locate left end of handle within the stack; pop handle off stack and push appropriate non-terminal LHS
3. accept — terminate parsing and signal success
4. error — call an error recovery routine

But how do we know

- that there is a complete handle on the stack?
- which handle to use?

LR parsing

Recognize handles with a DFA [Knuth 1965]

- DFA transitions shift states instead of symbols
- accepting states trigger reductions

Skeleton for LR parsing

```
push s0
token ← next_token()
while(true)
  s ← top of stack
  if action[s,token] = SHIFT(si) then
    push si
    token ← next_token()
  else if action[s,token] = REDUCE(A ::= β) then
    pop |β| states
    s' ← top of stack
    push goto[s',A]
  else if action[s, token] = ACCEPT then
    return
  else error()
```

- Accepting a sentence takes k shifts, l reduces, and 1 accept, where k is the length of the input string and l is the length of the rightmost derivation

Example tables

```
1 <goal> ::= <expr>
2 <expr> ::= <term>+<expr>
3            | <term>
4 <term> ::= <factor>*<term>
5            | <factor>
6 <factor> ::= id
```

state	ACTION				GOTO		
	id	+	*	$	*E*	*T*	*F*
0	*s*4	–	–	–	1	2	3
1	–	–	–	acc	–	–	–
2	–	*s*5	–	*r*3	–	–	–
3	–	*r*5	*s*6	*r*5	–	–	–
4	–	*r*6	*r*6	*r*6	–	–	–
5	*s*4	–	–	–	7	2	3
6	*s*4	–	–	–	–	8	3
7	–	–	–	*r*2	–	–	–
8	–	*r*4	–	*r*4	–	–	–

Example using the tables

Start with
Stack `$ 0`
Input `id * id + id $`

Formal definition of LR(k)

A BNF is LR(k) if

$$S \Rightarrow_{rm}^{*} \alpha A w \Rightarrow_{rm} \alpha\beta w$$

and

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

and

$$w|_k = y|_k$$

implies that $\alpha A y = \gamma B x$.

Why study LR grammars?

- almost all context-free programming language constructs can be expressed naturally with an LR(1) grammar
- LR grammars are the most general grammar that can be parsed by a deterministic bottom-up parser
- LR(1) grammars have efficient parsers
- LR parsers detect errors as early as possible
- LR grammars are more general: Every LL(k) grammar is also a LR(k) grammar

LR parsing

- LR(1) — not useful in practice because tables are too big
  - all deterministic languages have LR(1) grammar
  - very large tables
  - slow construction
- SLR(1)
  - smallest class of grammars
  - smallest tables
  - simple, fast construction
- LALR(1)
  - expressivity between SLR(1) and LR(1)
  - same number of states as SLR(1)
  - clever algorithm yields fast construction

Constructing LR parse tables

Parser states are modeled with LR(k) items.

Definition

An LR(k) item is a pair $[A \rightarrow \alpha \bullet \beta, w]$ where

- $A \rightarrow \alpha\beta$ is a rule; the $\bullet$ can divide the right hand side arbitrarily; intution: how much of the right hand side has been seen already
- w is a lookahead string containing at most k symbols

Constructing LR parse tables

Parser states are modeled with LR(k) items.

Definition

An LR(k) item is a pair $[A \to \alpha \bullet \beta, w]$ where

- $A \to \alpha\beta$ is a rule; the $\bullet$ can divide the right hand side arbitrarily; intution: how much of the right hand side has been seen already
- w is a lookahead string containing at most k symbols

Cases of interest

- $k = 0$: LR(0) items play a role in SLR(1) construction
- $k = 1$: LR(1) items are used for constructing LR(1) and LALR(1) parsers

Examples

Consider LR(0) items without lookahead

- $[A \rightarrow \bullet aBC]$ indicates that the parser is now looking for input derived from aBC
- $[A \rightarrow aB \bullet C]$ indicates that input derived from aB has been seen, now looking for something derived from C

There are four items associated with $A \rightarrow aBC$

The characteristic finite state machine (CFSM)

The CFSM for a grammar is a DFA which recognizes viable prefixes of right-sentential forms:

> *A viable prefix is any prefix that does not extend beyond the handle.*

The CFSM accepts when a handle has been discovered and needs to be reduced.
A state of the CFSM is a set of items.
To construct the CFSM we need two functions wherer I is a set of items and X is a grammar symbol:

- `closure0`(I) to build its states
- `goto0`(I, X) to determine its transitions

States of the CFSM — Closure

The closure of an item $[A \rightarrow \alpha \bullet B\beta$ contains the item itself and any other item that can generate legal strings following α. Thus if the parser has a viable prefix α on the stack, the remaining input should be derivable from $B\beta$.

Closure0

For a set I of LR(0) items, the set $\texttt{closure0}(I)$ is the smallest set such that

1. $I \subseteq \texttt{closure0}(I)$
2. If $[A \rightarrow \alpha \bullet B\beta] \in \texttt{closure0}(I)$ then $[B \rightarrow \bullet\gamma] \in \texttt{closure0}(I)$, for all rules $B \rightarrow \gamma$.

Implementation: start with first rule, repeat second rule until no further items can be added.

Transitions of the CFSM — Goto

Let I be a set of LR(0) items and X be a grammar symbol.

$$\texttt{goto0}(I, X) = \texttt{closure0}(\{[A \to \alpha X \bullet \beta] \mid [A \to \alpha \bullet X\beta] \in I)$$

If I is the set of items for a viable prefix $\gamma\alpha$, then $\texttt{goto0}(I, X)$ is the set of items for viable prefix $\gamma\alpha X$.
Accepting states of the CFSM: I is accepting if it contains a reduce item of the form $[A \to \alpha\bullet]$.

Construction of the CFSM

New start item $[S' \to \bullet S\$]$, where

- S' is a new start symbol that does not occur on any RHS
- S is the previous start symbol
- $ marks the end of input

Compute the set of states $\mathcal{S}$ where each state is a set of items

Construction of the state set

```
function items(G + S')
  I ← closure0({[S' → •S$]})
  𝒮 ← {I}
  𝒲 ← {I}
  while 𝒲 ≠ ∅
    remove some I from 𝒲
    for each grammar symbol X
      let I' ← goto0(I, X)
      if I' ≠ ∅ and I' ∉ 𝒮 then
        add I' to 𝒲 and 𝒮
  return 𝒮
```

LR(0) example

1	S	$\rightarrow$	$E\$$
2	E	$\rightarrow$	$E+T$
3		$\mid$	T
4	T	$\rightarrow$	$\mathtt{id}$
5		$\mid$	(E)

Construction of the LR(0) parse table

Output: tables *ACTION* and *GOTO*

1. Let $\{I_0, I_1, \ldots\} = \mathtt{items}(G)$
2. State i of the CFSM corresponds to item I_i
   - if $[A \to \alpha \bullet a\beta] \in I_i$, $a \in \Sigma$ and $\mathtt{goto0}(I_i, a) = I_j$ then $ACTION[i, a] = \mathtt{SHIFT}\ j$
   - If $[A \to \alpha\bullet] \in I_i$ and $A \neq S'$ then $ACTION[i, a] = \mathtt{REDUCE}\ A \to \alpha$, for all a
   - If $[S' \to S\$\bullet] \in I_i$ then $ACTION[i, a] = \mathtt{ACCEPT}$, for all a
3. If $\mathtt{goto0}(I_i, A) = I_j$ for variable A then $GOTO[i, A] = j$
4. set all undefined entries in *ACTION* and *GOTO* to $\mathtt{ERROR}$
5. initial state of parser corresponds to $I_0 = \mathtt{closure0}(\{[S' \to \bullet S\$]\})$

Example *ACTION* table

Conflicts

- If there are multiply defined entries in the *ACTION* table, then the grammar is not LR(0).
- There are two kinds of conflict
  - shift-reduce: shift and reduce are possible in the same item set
  - reduce-reduce: two different reduce actions are possible in the same item set
- Examples
  - $A \to \varepsilon \mid a\alpha$
    on input *a*, shift *a* or reduce $A \to \varepsilon$?
  - `a = b+c*d` with expression grammar
    after reading `c`, should we shift or reduce?
- Use lookahead to resolve conflicts

SLR(1) — simple lookahead LR

Add lookahead after computing the LR(0) item sets

1. Let $\{I_0, I_1, \ldots\} = \texttt{items}(G)$
2. State i of the CFSM corresponds to item I_i
   - if $[A \rightarrow \alpha \bullet a\beta] \in I_i$, for $a \neq \$$ and $\texttt{goto0}(I_i, a) = I_j$ then $ACTION[i, a] = \texttt{SHIFT}\ j$
   - If $[A \rightarrow \alpha\bullet] \in I_i$ and $A \neq S'$ then $ACTION[i, a] = \texttt{REDUCE}\ A \rightarrow \alpha$, for all $a \in FOLLOW(A)$
   - If $[S' \rightarrow S \bullet \$] \in I_i$ then $ACTION[i, \$] = \texttt{ACCEPT}$
3. If $\texttt{goto0}(I_i, A) = I_j$ for variable A then $GOTO[i, A] = j$
4. set all undefined entries in $ACTION$ and $GOTO$ to $\texttt{ERROR}$
5. initial state of parser corresponds to $I_0 = \texttt{closure0}(\{[S' \rightarrow \bullet S\$]\})$

LR(1) items

- Items of the form $[A \rightarrow \alpha \bullet \beta, a]$ for $a \in \Sigma$
- Propagate lookahead in construction to choose the correct reduction
- Lookahead has only effect on reduce items
- We can decide between reductions $[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \alpha \bullet, b]$ by examining the lookahead.

Closure1

Closure1

For a set I of LR(1) items, the set `closure1`(I) is the smallest set such that

1. $I \subseteq$ `closure1`(I)
2. If $[A \to \alpha \bullet B\beta, a] \in$ `closure0`(I) then $[B \to \bullet\gamma, b] \in$ `closure0`(I), for all rules $B \to \gamma$ and for all $b \in FIRST_1(\beta a)$.

Goto1

Let I be a set of LR(1) items and X be a grammar symbol.

$$\texttt{goto1}(I, X) = \texttt{closure1}(\{[A \to \alpha X \bullet \beta, a] \mid [A \to \alpha \bullet X\beta, a] \in I\})$$

Construction of the CFSM

as before

Construction of the LR(1) parse table

Output: tables *ACTION* and *GOTO*

1. Let $\{I_0, I_1, \ldots\} = \mathtt{items}(G)$
2. State i of the CFSM corresponds to item I_i
   - if $[A \to \alpha \bullet a\beta, b] \in I_i$, $a \neq \$$ and $\mathtt{goto1}(I_i, a) = I_j$ then $ACTION[i, a] = \mathtt{SHIFT}\ j$
   - If $[A \to \alpha\bullet, b] \in I_i$ and $A \neq S'$ then $ACTION[i, b] = \mathtt{REDUCE}\ A \to \alpha$
   - If $[S' \to S\bullet, \$] \in I_i$ then $ACTION[i, \$] = \mathtt{ACCEPT}$
3. If $\mathtt{goto1}(I_i, A) = I_j$ for variable A then $GOTO[i, A] = j$
4. set all undefined entries in *ACTION* and *GOTO* to $\mathtt{ERROR}$
5. initial state of parser corresponds to $I_0 = \mathtt{closure1}(\{[S' \to \bullet S, \$]\})$

LALR(1) parsing

The core of a set of LR(1) items is the set of LR(0) items obtained by stripping off all lookahead.
The following two sets have the same core:

- $\{[A \rightarrow \alpha \bullet \beta, a], [A \rightarrow \alpha \bullet \beta, c]\}$
- $\{[A \rightarrow \alpha \bullet \beta, b], [A \rightarrow \alpha \bullet \beta, c]\}$

Key idea

If two LR(1) item sets have the same core, then we can merge their states in the ACTION and GOTO tables.

LALR(1) table construction

Compared to LR(1) we need a single new step:

> *For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union.*
>
> *The goto function must be updated to reflect the replacement sets.*

The resulting algorithm is very expensive.
There is a more efficient algorithm that analyses the LR(0) CFSM.

Precedence and associativity

Precedence and associativity can be used to resolve shift-reduce conflicts in ambiguous grammars.

- lookahead symbol has higher precedence $\Rightarrow$ shift
- same precedence, left associative $\Rightarrow$ reduce

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees $\Rightarrow$ fewer reductions

Classic application: expression grammars

Precedence applied

With precedence and associativity we can use a very simple expression grammar without useless productions.

$$E \rightarrow E{*}E \mid E/E \mid E{+}E \mid E{-}E \mid (E) \mid -E \mid \texttt{id} \mid \texttt{num}$$

Left recursion vs right recursion

Right recursion

- required in top-down parsers for termination
- more stack space
- right-associative operators

Left recursion

- works fine in bottom-up parsers
- limits required stack
- left associative operators