First

Motivation

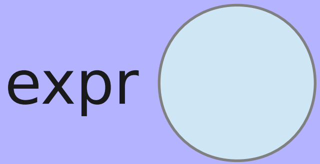
- Consider top down parsing
- We are at a node representing nonterminal X
- Need to decide which production to expand X into
  - Graft new nodes as children of X
  - Repeat for new nodes

Example

- Grammar:
  expr $\rightarrow$ term expr'
  expr' $\rightarrow$ + term expr' | - term expr' | λ
  term $\rightarrow$ factor term'
  term' $\rightarrow$ * factor term' | / factor term' | λ
  factor $\rightarrow$ id | num | (expr)
- Consider parsing "a + b * c"

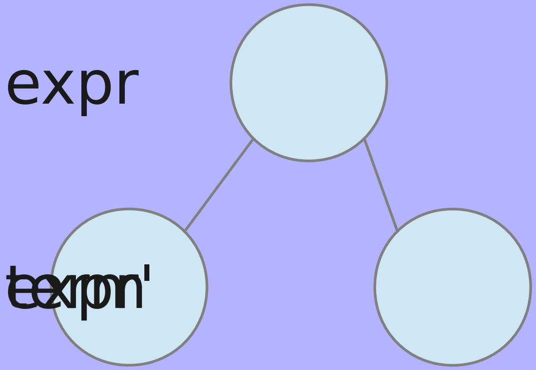
Example

- Start symbol is expr
- Create tree



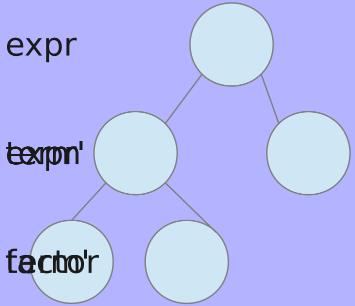
Example

- Only one choice, so expand expr



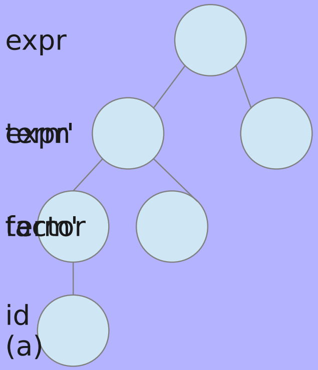
Example

- First, we parse term
- term $\rightarrow$ factor term’
- So expand it using this rule



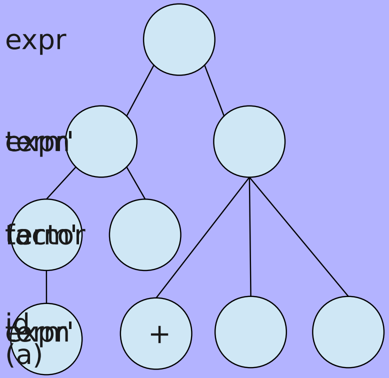
Example

- factor $\rightarrow$ id | num | (expr)
- Unambiguous, so expand to identifier
- Consume from input



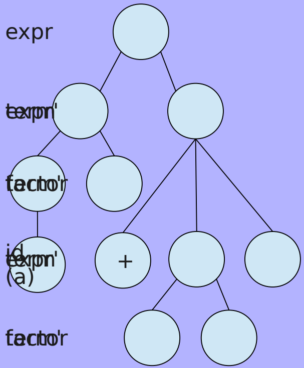
Example

- Remaining tokens: + b * c
- Now for the lookahead part
  - term’ $\rightarrow$ * factor term’ | / factor term’ | λ
- Next token doesn’t match * or / so we must choose λ
- Next tree node to expand: expr’
- Choose expr’ $\rightarrow$ + term expr’



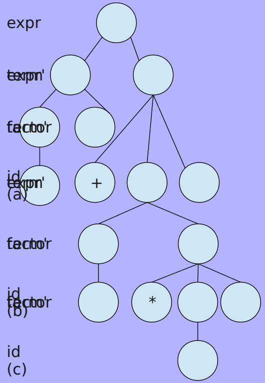
Example

- Remaining tokens: b * c
- Choose term $\rightarrow$ factor term’



Example

- Expand factor $\rightarrow$ id
- Expand term’ $\rightarrow$ factor term’
- Expand factor $\rightarrow$ id
- Remaining productions become λ



Predictive Parsing

- This grammar is LL(1)
- An LL(k) grammar can be parsed by scanning input left to right with leftmost derivations only and a lookahead of k tokens
  - Interesting grammars are usually either LL(1) or else LL(∞)

Automation

- We'd like to automate the process. How?
- For each nonterminal, determine what the initial tokens could be for its expansion
  - Then by looking at what we want to expand and what next token of input is: Can tell which production to do
  - To determine when we're done with nonterminal N, we must then know what tokens could follow a legal derivation of N

Example

- Suppose $S \to A \mid x$ $A \to y \mid z$
  - first[S] = {x,y,z}
  - first[A] = {y,z}

Example

- $S \rightarrow A \mid B$
  $A \rightarrow x \mid y\ S$
  $B \rightarrow A \mid z$
  - first[A] = {x,y}
  - first[B] = first[A] $\cup$ {z} = {x,y,z}
  - first[S] = first[A] $\cup$ first[B] = {x,y,z}

First Sets

- To generate *first* sets
- Let first be a map
  - Key = string: Symbol
  - Value = set of strings: First set for the key

Initialization

- Initialization: first[t] = {t} for all terminals t
- Any terminal always leads off with itself

Compute

- Now use repeated cycling:

```
do
    for all nonterminals N:
        for all productions P with lhs of N:
            for x in P:
                first[N] = union( first[N], first[x] )
                if x is not nullable:
                    break
until first stabilizes
```

Example

- $S \rightarrow a \mid A\ b\ B \mid B\ C\ D\ e$
  $A \rightarrow x \mid C\ y$
  $B \rightarrow D\ C \mid q$
  $C \rightarrow D \mid w \mid C\ z$
  $D \rightarrow b \mid \lambda$

- Nullable: B,C,D
- First:
  - A : b w x y z
  - B : b q w z
  - C : b w z
  - D : b
  - S : a b e q w x y z

Assignment

- Extend your code to compute the first set for a grammar
- Your program must work with this test suite

Sources

- Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers:Principles,Techniques and Tools (2nd ed).
- K. Louden. Compiler Construction: Principles and Practice.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org