Bottom-up parsing

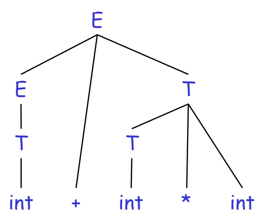
CS164
3:30-5:00 TT
10 Evans

Welcome to the running example

- we'll build a parser for this grammar:
  - E → E + T | E – T | T
  - T → T * int | int
- see, the grammar is
  - left-recursive
  - not left-factored
- … and our parser won't mind!
  - we can make the grammar ambiguous, too

Example input, parse tree

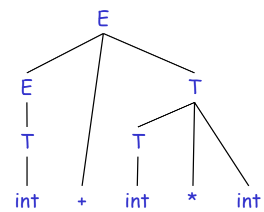
- input:
  int + int * int
- its parse tree:



Chaotic bottom-up parsing

Key idea: build the derivation in reverse

E
E + T
T + T
T + T * int
int + T * int
int + int * int



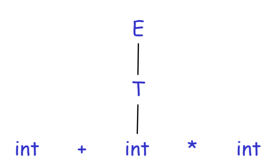
Chaotic bottom-up parsing

- The algorithm:
  1. stare at the input string *s*
     - feel free to look anywhere in the string
  2. find in *s* a right-hand side *r* of a production *N*→*r*
     - ex.: found int for a production T → int
  3. **reduce** the found string *r* into its non-terminal *N*
     - ex.: replace int with T
  4. if string reduced to start non-terminal
     - we're done, string is parsed, we got a parse tree
  5. otherwise continue in step 1

Don't celebrate yet!

- not guaranteed to parse a correct string
  - is this surprising?
- example:

and we are stuck
int + E * int
int + T * int
int + int * int



Lesson from chaotic parser

- Lesson:
  - if you're lucky in selecting the string to reduce next, then you will successfully parse the string
- How to "beat the odds"?
  - that is, how to find a lucky sequence of reductions that gives us a derivation of the input string?
  - use non-determinism!

What's this non-determinism, again?

- You took cs164, then became a stock broker:
  - want 16 celebrities sign you as their private broker
  - here's how: send free advice to 1024 celebrities
    - to half of them: "MSFT will go up tomorrow, buy now"
    - guess what's your advice for the other 512 folks
  - send free advice to 512 who got the correct advice
    - to half of them: "AAPL will go down tomorrow, sell now"
  - …
  - then apply for a broker job with the 16 who got six correct predictions in a row
    - that's sorta how we'll parse the string

Non-deterministic chaotic parser

The algorithm:

1. find in input all strings that can be reduced
   - assume there are k of them
2. create k copies of the (partially reduced) input
   - it's like spawning k identical instances of the parser
3. in each instance, perform one of k reductions
   - and then go to step 1, advancing and further spawning all parser instances
4. stop when at least one parser instance reduced the string to start non-terminal

Properties of the n.d. chaotic parser

Claim:
- the input will be parsed by (at least) one parser instance

But:
- exponential blowup: k*k*k*…*k parser copies
- (how many k's are there?)

Also:
- Multiple (usually many) instances of the parser produce the correct parse tree. This is wasteful.

Overview

- Chaotic bottom-up parser
  - it will give us the parse tree, but only if it's lucky
- Non-deterministic bottom-up parser
  - creates many parser instances to make sure at least one builds the parse trees for the string
  - an instance either builds the parse tree or gets stuck
- Non-deterministic LR parser (next)
  - restrict where a reduction can be made
  - as a result, fewer instances necessary

Non-deterministic LR parser

- What we want:
  - create multiple parser instances
    - to find the lucky sequence of reductions
  - but the parse tree is found by at most one instance
    - zero if the input has syntax error

Two simple rules to restrict # of instances

1. split the input in two parts:
   - **right:** unexamined by parser
   - **left:** in the parser (we'll do the reductions here)

   int ▸ + int * int   after reduction: T ▸ + int * int

2. reductions allowed only on right part next to split

allowed: T + int ▸ * int   after reduction: T + T ▸ * int
not allowed: int + int ▸ * int   after reduction: T + int ▸ * int

☛ hence, left part of string can be kept on the stack

Wait a minute!

Aren't these restrictions fatally severe?
- **the doubt:** no instance succeeds to parse the input

No. recall: one parse tree ⇔ multiple derivations
- in n.d. chaotic parser, the instances that build the same parse tree each follow a different derivation

Wait a minute! (cont)

recall: two interesting derivations
- left-most derivation, right-most derivation

LR parser builds right-most derivation
- but does so in reverse: first step of derivation is the last reduction (the reduction to start nonterminal)
- example coming in two slides

hence the name:
- L: scan input left to right
- R: right-most derivation

so, if there is a parse tree, LR parser will build it!
- this is the key theorem

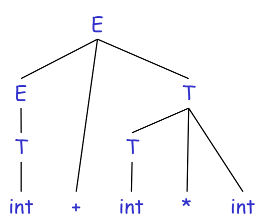
LR parser actions

- The left part of the string will be on the stack
  - the ▸ symbol is the top of stack
- Two simple actions
  - **reduce:**
    - like in chaotic parser,
    - but must replace a string on top of stack
  - **shift:**
    - shifts ▸ to the right,
    - which moves a new token from input onto stack, potentially enabling more reductions
- These actions will be chosen non-deterministically

Example of a correct LR parser sequence

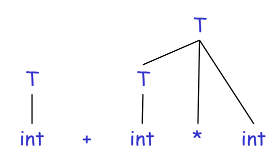
A "lucky" sequence of shift/reduce actions (string parsed!):

E ▸
E + T ▸
E + T * int ▸
E + T * ▸ int
E + T ▸ * int
E + int ▸ * int
E + ▸ int * int
E ▸ + int * int
T ▸ + int * int
int ▸ + int * int
▸ int + int * int



Example of an <u>incorrect</u> LR parser sequence

stuck! why can't we reduce to E + T ?
T + T ▸
T + T * int ▸
T + T * ▸ int
T + T ▸ * int
T + int ▸ * int
T + ▸ int * int
T ▸ + int * int
int ▸ + int * int
▸ int + int * int



Where did the parser instance make the mistake?

Non-deterministic LR parser

The algorithm: (compare with chaotic n.d. parser)

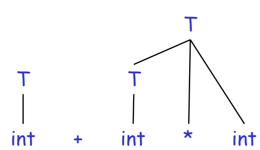
1. find all reductions allowed on top of stack
   - assume there are k of them
2. create k new identical instances of the parser
3. in each instance, perform one of the k reductions; in original instance, do no reduction, shift instead
   - and go to step 1
4. stop when a parser instance reduced the string to start non-terminal

Overview

- Chaotic bottom-up parser
  - tries one derivation (in reverse)
- Non-deterministic bottom-up parser
  - tries all ways to build the parse tree
- Non-deterministic LR parser
  - restricts where a reduction can be made
  - as a result,
    - only one instance succeeds (on an unambiguous grammar)
    - all others get stuck
- Generalized LR parser (next)
  - **idea:** kill off instances that are going to get stuck ASAP

Revisit the <u>incorrect</u> LR parser sequence

T + T ▸
T + T * int ▸
T + T * ▸ int
T + T ▸ * int
T + int ▸ * int
T + ▸ int * int
T ▸ + int * int
int ▸ + int * int
▸ int + int * int



Key question:

What was the earliest stack configuration where we could tell this instance was doomed to get stuck?

Doomed stack configurations

The parser made a mistake to shift to

T + ▸ int * int

rather than reducing to

E ▸ + int * int

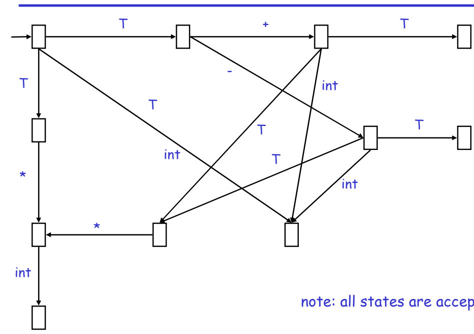
The first configuration is doomed

- because the T will never appear on top of stack so that it can be reduced to E
- hence this instance of the parser can be killed (it will never produce a parse tree)

How to find doomed parser instances?

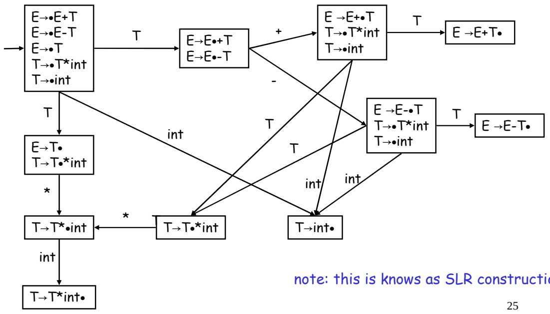
- Look at their stack!
- How to tell if a stack is doomed:
  - list all <u>legal</u> (non yet doomed) stack configurations
  - if a stack is not legal, kill the instance
- Listing legal stack configurations
  - list prefixes of all right-most derivations until you see a pattern
  - describe the pattern as a DFA
  - if the stack configuration is not from the DFA, it's doomed

The stack-checking DFA



note: all states are accepting states

Constructing the stack-checking DFA



note: this is knows as SLR construction