Building a Parser II

CS164
3:30-5:00 TT
10 Evans

Grammars

- Programming language constructs have recursive structure.
  - which is why our hand-written parser had this structure, too
- An expression is either:
  - number, or
  - variable, or
  - expression + expression, or
  - expression - expression, or
  - (expression), or
  - …

Context-free grammars (CFG)

- a natural notation for this recursive structure
- grammar for our balanced parens expressions:
  BalancedExpression → **a** | **(** BalancedExpression **)**
- describes (generates) strings of symbols:
  - a, (a), ((a)), (((a))), …
- like regular expressions but can refer to
  - other expressions (here, BalancedExpression)
  - and do this recursively (giving is "non-finite state")

Example: arithmetic expressions

- Simple arithmetic expressions:
  E → n | id | (E) | E + E | E * E
- Some elements of this language:
  - id
  - n
  - (n)
  - n + id
  - id * (id + id)

Symbols: Terminals and Nonterminals

- grammars use two kinds of <u>symbols</u>
- terminals:
  - no rules for replacing them
  - once generated, terminals are permanent
  - these are tokens of our language
- nonterminals:
  - to be replaced (expanded)
  - in regular expression lingo, these serve as names of expressions
  - start non-terminal: the first symbol to be expanded

Notational Conventions

- In these lecture notes, let's adopt a notation:
  - Non-terminals are written upper-case
  - Terminals are written lower-case
    or as symbols, e.g., token LPAR is written as (
  - The start symbol is the left-hand side of the first production

Derivations

- This is how a grammar generates strings:
  - think of grammar rules (called <u>productions</u>) as rewrite rules
- <u>Derivation</u>: the process of generating a string
  1. begin with the start non-terminal
  2. rewrite the non-terminal with some of its productions
  3. select a non-terminal in your current string
     i. if no non-terminal left, done.
     ii. otherwise go to step 2.

Example: derivation

Grammar: E → n | id | (E) | E + E | E * E

- a derivation:

E	rewrite E with (E)
(E)	rewrite E with n
(n)	this is the final string of terminals

- another derivation (written more concisely):
  E → (E) → (E * E) → (E + E * E) → (n + E * E) → (n + id * E)
  → (n + id * id)

So how do derivations help us in parsing?

- A program (a string of tokens) has no syntax error if it can be derived from the grammar.
  - but so far you only know how to derive some (any) string, not how to check if a given string is derivable
- So how to do parsing?
  - a naïve solution: derive all possible strings and check if your program is among them
  - not as bad as it sounds: there are parsers that do this, kind of. Coming soon.

Decaf Example

A fragment of Decaf:

STMT → **while (** EXPR **)** STMT
| **id (** EXPR **) ;**

EXPR → EXPR **+** EXPR
| EXPR **-** EXPR
| EXPR **<** EXPR
| **(** EXPR **)**
| **id**

Decaf Example (Cont.)

Some elements of the (fragment of) language:

id (id) ;
id ((((id)))) ;
while (id < id) id (id) ;
while (while (id)) id (id) ;
while (id) while (id) while (id) id (id) ;

Question: One of the strings is not from the language. Which one?

STMT → **while (** EXPR **)** STMT
| **id (** EXPR **) ;**

EXPR → EXPR **+** EXPR | EXPR **-** EXPR
| EXPR **<** EXPR | **(** EXPR **)** | **id**

CFGs (definition)

- A CFG consists of
  - A set of <u>terminal</u> symbols T
  - A set of <u>non-terminal</u> symbols N
  - A <u>start symbol</u> S (a non-terminal)
  - A set of <u>productions</u>:

produtions are of two forms ($X \in N$)

$X \to \varepsilon$, or
$X \to Y_1 Y_2 \dots Y_n$ where $Y_i \in N \cup T$

context-free grammars

- what is "context-free"?
  - means the grammar is <u>not</u> context-sensitive
- context-sensitive gramars
  - can describe more languages than CFGs
  - because their productions restrict when a non-terminal can be rewritten. An example production:
    d N → d A B c
  - meaning: N can be rewritten into ABc only when preceded by d
  - can be used to encode semantic checks, but parsing is hard

Now let's parse a string

- recursive descent parser derives all strings
  - until it matches derived string with the input string
  - or until it is sure there is a syntax error

Recursive Descent Parsing

- Consider the grammar
  $E \to T + E \mid T$
  $T \to int \mid int * T \mid (E)$
- Token stream is: $int_5 * int_2$
- Start with top-level non-terminal E

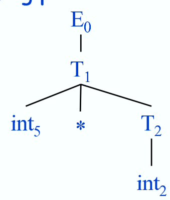
- Try the rules for E in order

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \to T_1 + E_2$
- Then try a rule for $T_1 \to (E_3)$
  - But (does not match input token int_5
- Try $T_1 \to int$. Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \to int * T_2$
  - This will match but + after T_1 will be unmatched
- Have exhausted the choices for T_1
  - Backtrack to choice for E_0

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \to T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \to$ int * T_2 and $T_2 \to$ int
  - With the following parse tree



A Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of
  - A given token terminal

    bool term(TOKEN tok) { return in[next++] == tok; }
  - A given production of S (the n^{th})

    bool S_n() { … }
  - Any production of S:

    bool S() { … }
- These functions advance next

A Recursive Descent Parser (3)

- For production $E \to T + E$

  bool E_1() { return T() && term(PLUS) && E(); }
- For production $E \to T$

  bool E_2() { return T(); }
- For all productions of E (with backtracking)

```
bool E() {
  int save = next;
  return   (next = save, E1())
         || (next = save,  E2());   }
```


A Recursive Descent Parser (4)

- Functions for non-terminal T

bool T_1() { return term(OPEN) && E() && term(CLOSE); }
bool T_2() { return term(INT) && term(TIMES) && T(); }
bool T_3() { return term(INT); }

```
bool T() {
  int save = next;
  return   (next = save, T1())
         || (next = save,  T2())
         || (next = save,  T3()); }
```


Recursive Descent Parsing. Notes.

- To start the parser
  - Initialize next to point to first token
  - Invoke E()
- Notice how this simulates our backtracking example from lecture
- Easy to implement by hand
- Predictive parsing is more efficient

Recursive Descent Parsing. Notes.

- Easy to implement by hand
  - An example implementation is provided as a supplement "Recursive Descent Parsing"
- But does not always work …

Recursive-Descent Parsing

- Parsing: given a string of tokens $t_1\ t_2\ \ldots\ t_n$, find its parse tree
- Recursive-descent parsing: Try all the productions exhaustively
  - At a given moment the fringe of the parse tree is: $t_1\ t_2\ \ldots\ t_k\ A\ \ldots$
  - Try all the productions for A: if $A \to BC$ is a production, the new fringe is $t_1\ t_2\ \ldots\ t_k\ B\ C\ \ldots$
  - Backtrack when the fringe doesn't match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does Not Work

- Consider a production $S \to S\ a$:
  - In the process of parsing S we try the above rule
  - What goes wrong?
- A left-recursive grammar has a non-terminal S

  $S \to^+ S\alpha$ for some α
- Recursive descent does not work in such cases
  - It goes into an ∞ loop

Elimination of Left Recursion

- Consider the left-recursive grammar
  $S \rightarrow S\ \alpha \mid \beta$
- S generates all strings starting with a β and followed by a number of α
- Can rewrite using right-recursion
  $S \rightarrow \beta\ S'$
  $S' \rightarrow \alpha\ S' \mid \varepsilon$

Elimination of Left-Recursion. Example

- Consider the grammar
  $S \rightarrow 1 \mid S\ 0$ ($\beta = 1$ and $\alpha = 0$)

  can be rewritten as
  $S \rightarrow 1\ S'$
  $S' \rightarrow 0\ S' \mid \varepsilon$

More Elimination of Left-Recursion

- In general
  $S \rightarrow S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as
  $S \rightarrow \beta_1\ S' \mid \ldots \mid \beta_m\ S'$
  $S' \rightarrow \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$

General Left Recursion

- The grammar
  $S \rightarrow A\ \alpha \mid \delta$
  $A \rightarrow S\ \beta$
  is also left-recursive because
  $S \rightarrow^+ S\ \beta\alpha$
- This left-recursion can also be eliminated
- See [ASU], Section 4.3 for general algorithm

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar