COMP 520 - Compilers

Lecture 3 (Tue Jan 18, 2022)

EBNF Grammars and Top-down Parsing

- PLPJ Reading for 1/18, 1/20
  - Parsing, Secn 4.3 (pp 83 – 84)
  - Top-down parsing, Secn 4.3.2, (pp 87 – 89)
  - Recursive descent parsing, Secn 4.3.3 (pp 89 – 93)
  - Systematic development of recursive-descent parsers, Secn 4.3.4 (pp 93 – 109)

Topics

- Context-free grammars and context-free languages
  - Leftmost derivations
- Parsing context free grammars
  - Top-down parsing
- Extended BNF form for grammars
  - Definitions
  - Grammar transformations
- Recursive descent parsers
  - Approach
  - Example

Context-free grammar

- A CFG consists of
  - a set of nonterminal symbols N (start with upper case)
  - a set of terminal symbols T (start with lowercase)
  - a distinguished nonterminal start symbol
  - a set of rewrite rules of the form A ::= α where
    - A $\in$ N
    - α is a sequence of N $\cup$ T or ε (empty sequence)

- Example (CFG G_0)
  - N = { S, A },
  - T = { (,) , x , $ }
  - start symbol S
  - rules

```
S ::= A $
A ::= ( A )
A ::= x
```

Context-free language

- A *sentence w* is generated by a CFG G if
  - $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ . \ . \ . \Rightarrow \alpha_n = w$ where
    - S is the start symbol
    - w consists exclusively of terminal symbols
    - $\alpha_i \Rightarrow \alpha_{i+1}$ if
      - $\alpha_i = \beta W \gamma$, and $\alpha_{i+1} = \beta \omega \gamma$ and W ::= ω is a rule in G

- The *context free language* generated by a CFG G
  - L(G) is the set of all sentences generated by G

$$L(G) = \{w \mid w \in T^* \text{ and } S \overset{*}{\Rightarrow} w\}$$

- What sentences are generated by CFG G_0?

S ::= A $

A ::= (A)

A ::= x

Leftmost derivation

- Order of substitution does not affect the sentences generated by G
  - *Example*

    S ::= B C $

    B ::= b

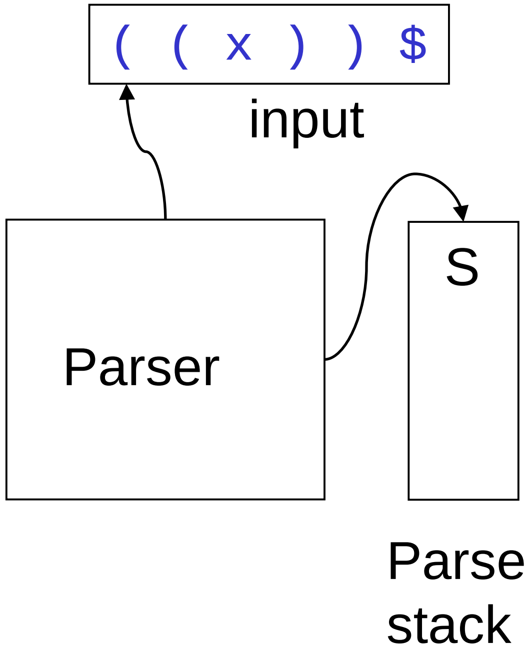
    C ::= (C)

    C ::= c
  - Any sentence in L(G) can be generated using a *leftmost* derivation

- Leftmost derivation
  - $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ ... \Rightarrow \alpha_n = w$ where
    - S is the start symbol
    - w consists exclusively of terminal symbols
    - $\alpha_i \Rightarrow \alpha_{i+1}$ if
      - $\alpha_i = uB\gamma$ and $\alpha_{i+1} = u\beta\gamma$ where
      - *u* consists zero or more terminal symbols and
      - $B ::= \beta$ is a rule in G

Top-down parsing

- How can we *recognize* sentences in a language?
  - Simulate a derivation using a pushdown automaton
    - top-down parser simulates a *leftmost derivation*
- Top-down parser operation
  - Input *w* is read from left to right
  - Parse stack initialized with start symbol *S*
  - Repeat until parse stack is empty or input is exhausted
    - if top of parse stack is terminal b
      - if b matches current input symbol then pop b from stack and advance to next input symbol
      - otherwise parse error
    - if top of parse stack is a nonterminal A
      - "predict" correct rule A ::= α from grammar G
      - pop A and push α
  - $w \in L(G)$ iff stack empty and input exhausted



CFG G_0

S ::= A $

A ::= (A)

A ::= x

Operation of top-down parser

CFG G_0

S ::= A $

A ::= (A)

A ::= x

Example

- CFG G_0
- input string: (x)$

Input seen	Stack	Input left	Action
	S	(x)$	predict S ::= A$
	A$	(x)$	see “(“, predict A ::= (A)
	(A)$	(x)$	match terminal
(	A)$	x)$	see “x”, predict A ::= x
(	x)$	x)$	match terminal
(x	)$	)$	match terminal
(x)	$	$	match terminal
(x)$			stack empty, no input left – sentence recognized

Leftmost derivation

Key idea for top-down parser

- Resolve choices in grammar rules by looking at next symbol of input

  A ::= (A)

  A ::= x

  Two choices of rule for A. Which terminals can appear at the start of each choice?

  – starters of (A) = { (}
  – starters of x = { x }

  Since these two sets are disjoint, we can always resolve choice for A by looking at the next input symbol

  What if the grammar were changed as follows?

  S ::= A $

  A ::= (A)

  A ::= ε (empty sequence)

Top-down parsing and the LL(1) condition

- LL(1) condition
  - guarantees parser can always "predict" the correct rule to apply based on the next (1) input symbol reading Left to right following the Leftmost derivation

- CFG grammars
  - grammars meeting the LL(1) condition can be efficiently parsed using a top-down parser
  - however, many grammars do not meet the LL(1) condition

- Example 1
  - N = { S A }
  - T = { (,) x $}
  - rules

```
S  ::=   ( A ) $
A  ::=   x , A
A  ::=   x
```

Example 2
  - same N, T
  - new rules

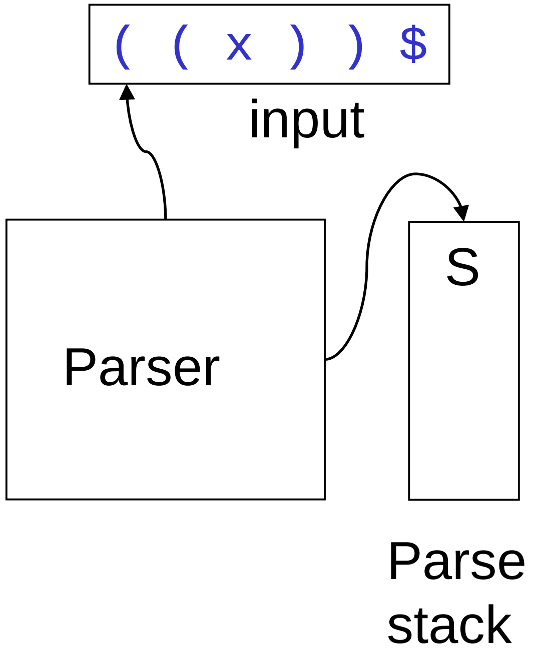
```
S  ::=   ( A ) $
A  ::=   A , x
A  ::=   x
```

- We may need to modify grammars to achieve the LL(1) condition
  - not always possible: some CFLs do not have an LL(1) grammar (!)

Top-down parsing

- How can we *recognize* sentences in a language?
  - Simulate a derivation using a pushdown automaton
    - top-down parser simulates a *leftmost derivation*

- Top-down parser operation
  - Input *w* is read from left to right
  - Parse stack initialized with start symbol *S*
  - Repeat until parse stack is empty or input is exhausted
    - if top of parse stack is terminal b
      - if b matches current input symbol then pop b from stack and advance to next input symbol
      - otherwise parse error
    - if top of parse stack is a nonterminal A
      - "predict" correct rule A ::= α from grammar G
      - pop A and push α
  - $w \in L(G)$ iff stack empty and input exhausted

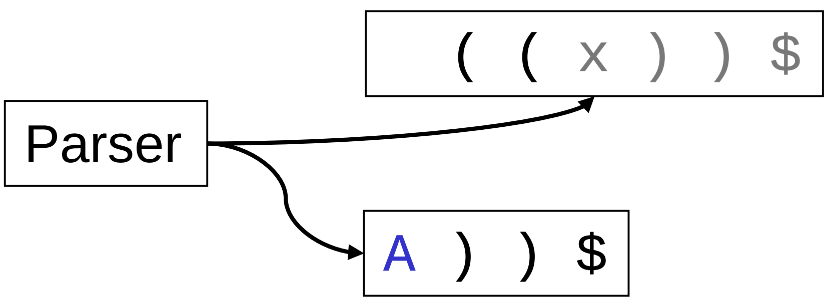


CFG G_0

S	::=	A $
A	::=	(A)
A	::=	x

Recursive Descent Parsing

- Implementation of a top-down parser using recursive procedures
  - uses a set of mutually recursive procedures
    - one procedure parseN() for each nonterminal N in the grammar
  - parseN() parses the right-hand side(s) of rule(s) for N
    - maintains some local state recording progress
  - the parse stack is implicitly maintained in the procedure call stack



CFG G_0

S ::= A $

A ::= (A)

A ::= x

```
parseS() {
   parseA();
   accept( '$' );
}

parseA() {
   if ( currChar == '(' ) {
      accept( '(' );
      parseA();
      accept( ')' );
   }
   else
      accept( 'x' );
}
```

EBNF grammars

- An *Extended BNF* grammar is a CFG with
  - rules of the form A ::= α where A $\in$ N
    and α is an *extended regular expression* that may contain

BNF

- sequences of terminals and nonterminals
  - IfStmt ::= `if` Exp `then` Stmt ElsePart
  - SimpleStmt ::= `skip`
- empty sequence ε
  - Empty ::= ε

Extended

- choice |
  - ElsePart ::= `else` Stmt | Empty
- repetition *
  - Stmt ::= SimpleStmt*
- grouping ()
  - Prog ::= (`let` Decl (`;` Decl)* `in` Stmt) | IfStmt

EBNF language

- A *sentence* *w* is generated by a EBNF grammar G if
  - $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ ... \Rightarrow \alpha_n = w$ where
    - S is the start symbol
    - w consists exclusively of terminal symbols
    - $\alpha_i \Rightarrow \alpha_{i+1}$ if
      - $\alpha_i = \beta W \gamma$ and $\alpha_{i+1} = \beta \mu \gamma$ where
        - » W ::= ω is a rule in G and
        - » regular expression ω can generate μ

- An EBNF grammar G generates a language L(G)
  - L(G) is a *context free language*

EBNF grammars

- EBNF is simply a convenience
  - Every EBNF grammar can be rewritten as a simple context free grammar (CFG)
  - Ex: eliminate EBNF extensions in this rule

    Prog ::= let Decl (; Decl)* in Stmt | IfStmt

- EBNF benefits
  - simpler expression of grammars
  - better target for grammar transformations
  - we can conveniently extend recursive descent parsers to directly parse an EBNF grammar

Grammar Transformations

- Transform grammar to a form suitable or more convenient for parsing
  - Substitution of nonterminal symbols

    C ::= A b D => C ::= (c | d) b D

    A ::= c | d

  - Left-factorization

    IfStmt ::= `if` Exp `then` Stmt
    | `if` Exp `then` Stmt `else` Stmt

    →

    IfStmt ::= `if` Exp `then` Stmt (ε | `else` Stmt)

  - Elimination of Left Recursion

    N ::= X | NY

    →

    N ::= X (Y)*

Elimination of left-recursion

- Why is the left recursion elimination transformation correct?
  – General case can be reduced to simple case

    $N ::= \alpha_1 \mid \ldots \mid \alpha_m \mid N\beta_1 \mid \ldots \mid N\beta_n$

    $\rightarrow$

    $N ::= \underbrace{(\alpha_1 \mid \ldots \mid \alpha_m)}_{X} \mid N\underbrace{(\beta_1 \mid \ldots \mid \beta_n)}_{Y}$

  – Correctness of $N ::= X \mid NY \quad \rightarrow \quad N ::= X\,(Y)^*$
    - examine derivations of both sides

Simplify a grammar for parsing

- A simple grammar for a subset of arithmetic expressions

  E ::= T | E Op T

  T ::= (E) | num

  Op ::= + | ×

- Add new start symbol S and terminal $ representing end-of-input

  S ::= E $

- Remove left recursion

  E ::= T (Op T)*

  T ::= (E) | num

  Op ::= + | ×

- Substitute for Op

  E ::= T ((+ | ×) T)*

  T ::= (E) | num

Other versions of arithmetic expression grammars

- Simplify these for parsing. Do they meet the LL(1) condition?

- Right recursive arithmetic expressions

  E ::= T | T Op E
  T ::= (E) | num
  Op ::= + | ×

- Left and right recursive arithmetic expressions

  E ::= T | E Op E
  T ::= (E) | num
  Op ::= + | ×

Recursive descent parsers for EBNF

- How can we implement recursive descent parsers for EBNF?
  - Choice $\alpha \mid \beta$
    - Conditional or case statement based on next input symbol
  - Repetition α^*
    - While statement that repeats based on next input symbol
  - Example

    S ::= E $

    E ::= T ((+ | ×) T)*

    T ::= (E) | num

```
void parseS() {
   parseE();
   accept('$');
}

void parseE() {
   parseT();
   while (currChar == '+'
            || currChar == '×') {
         acceptIt();
         parseT();
   }
}

void parseT() {
   switch (currChar) {
      case '0',.., case'9':
            acceptIt();
            return;

      case '(':
            acceptIt();
            parseE();
            accept( ')' );
            return;
}}
```

Informal definitions of grammar properties

- Given an EBNF grammar
  - nonterminal set N, start symbol S, Terminal set T
  - assume one rule per nonterminal
    - multiple rules with same NT at left can be combined

      $A ::= \alpha_1 \quad \ldots \quad A ::= \alpha_m \quad \rightarrow \quad A ::= \alpha_1 \mid \ldots \mid \alpha_m$

- Define
  - *Nullable*(α)
    - Property that is True iff α can derive the empty string
  - *Starters*[α]
    - Set of terminals that may start derivations from α
    - Includes ε if Nullable(α)
  - *Followers*[A] where $A \in N$
    - Set of terminals that may follow A in a derivation
    - For augmented grammars, only Followers[S] includes ε

Informal LL(1) condition for EBNF grammars

- Idea
  - For each choice of the form $A ::= \beta\ (\alpha_1 \mid \ldots \mid \alpha_m)\ \gamma$
    - Starters[α_i] and Starters[α_j] must be disjoint for all $1 \le i,j \le m$
  - For each repetition of the form $A ::= \beta\ (\alpha)^*\ \gamma$
    - Starters [α] and Starters[γ] are disjoint
    - Nullable(α) is False

- Example
  - Is this EBNF grammar LL(1)?

```
S ::= A $
A ::= x z | x E (y E)* z
E ::= a | b
```

Parsing a grammar that does not meet LL(1)

- Consider conditional statements
  - with optional "else" part

- Example G_1
  - N = {Stmt, Exp, ElsePart},
  - T = {`if`, `then`, `skip`, `else`, `true`, `false`}
  - start symbol Stmt
  - rules

```
Stmt     ::= if Exp then Stmt ElsePart
Exp      ::= true
Exp      ::= false
Stmt     ::= skip
ElsePart ::= else Stmt
ElsePart ::= ε
```

- What is $L(G_1)$? Why does this grammar not meet the LL(1) condition? Can we parse it anyway?