Left Factoring

Parsing

- How to convert sequence of tokens to parse tree?
- Several methods possible

Hand-Written

- One way: Totally manually
- Typically using a *recursive descent parser*
- Ex: suppose grammar is:
  S → stmt-list
  stmt-list → stmt SEMI stmt-list | stmt SEMI
  stmt → cond | assign | func-call
  func-call → ID LPAREN RPAREN
  expr → expr ADDOP term | term
  term → factor MULOP term | factor
  factor → NUM | ID | LPAREN expr RPAREN
  assign → ID EQUALS expr
  cond → IF LPAREN expr RPAREN LBRACE stmt-list RBRACE | IF LPAREN expr RPAREN LBRACE stmt-list RBRACE ELSE LBRACE stmt-list RBRACE

Recursive Descent

```
def parse_stmt():
    t = peekToken()
    if t.sym == IF:  parse_cond()
    elif t.sym ==ID:
        t2 = peekNextToken()
        if t2 == EQUALS:
            parse_assign()
        else:
            parse_funcCall()
    else:
        error
    t = getToken()
    if t != SEMI:
        error
```

Parse

```
def parse_cond():
    t = getToken()
    if t.sym != IF: error
    t = getToken()
    if t.sym != LPAREN: error
    parse_expr()
    t = getToken()
    if t.sym != RPAREN: error
    t = getToken()
    if t.sym != LBRACE: error
    parse_stmt_list()
    t = getToken()
    if t.sym != RBRACE: error
    t = peekToken()
    if t == ELSE:
        ...
```

Analysis

- Fairly straightforward to code – no advanced ideas
- Very tedious
  - *Every* symbol in *every* grammar production requires a piece of code to handle
- Error prone
  - When handling stmt-list: How do you know when you've come to end?
  - For this grammar, not too difficult – analyze by sight
  - For more complex grammars: Easy to improperly code it
- Change grammar = change potentially large amount of code

Generated Parser

- Can write code that will automate much of the compilation work
- This requires some compiler theory
- Also imposes some restrictions on grammar design
  - The trade-off may or may not be worth it

Problem

- If two choices share a common prefix, this complicates things
  - Observe: assignment vs. func-call
- The manually written parser had to look at the next+1 token
- But special cases are harder to do in automated approach

Transformation

- Assume that we have something like this for “assignment or function call”:
  stmt $\rightarrow$ ID = expr | ID LP num RP
- Find common prefix
  - Here, it’s “ID”
- Break up like so:
  stmt $\rightarrow$ ID stmt’
  stmt’ $\rightarrow$ = expr | LP num RP

Transformation

- In general, common prefix could be more than one symbol
- func-call $\rightarrow$ ID LP RP | ID LP NUM RP | ID LP NUM COMMA NUM RP
- Factor out longest common prefix:
  func-call $\rightarrow$ ID LP func-call'
  func-call' $\rightarrow$ RP | NUM RP | NUM COMMA NUM RP
  - But now we must do factoring again!

Factoring

- Factoring may create λ-productions
- cond $\rightarrow$ IF LP expr RP stmt | IF LP expr RP stmt ELSE stmt
- Factored:
  cond $\rightarrow$ IF LP expr RP stmt cond'
  cond' $\rightarrow$ λ | ELSE stmt

Other Items

- What if other items? Leave alone
- Example: stmt $\rightarrow$ ID EQ expr | ID LP RP | WHILE LP expr RP stmt
- Factored:
  stmt $\rightarrow$ ID stmt' | WHILE LP expr RP stmtid stmt'
  stmt' $\rightarrow$ EQ expr | LP RP

General Method

- Left factoring: general method: Suppose grammar has productions A $\rightarrow p_1|p_2|\ldots|p_n$
- Find longest prefix common to at least two productions
  - Denote this as α
  - If length of α is zero: We've factored A completely
- Create a new unique symbol σ
- For all productions A $\rightarrow p_i$ where p_i begins with α:
  - Delete production A $\rightarrow p_i$ from the grammar
  - Add production $\sigma \rightarrow p_i - \alpha$ to the grammar
    - "$p_i - \alpha$" means "remove initial prefix of α from p_i"
- Add production A $\rightarrow \alpha\,\sigma$ to the grammar
- Repeat above steps until all symbols completely factored.

Left Recursion

- Problem: we had productions like this: expr $\rightarrow$ expr ADDOP term
- How do we code this?

Obvious Way

```
def parse_expr():
    parse_expr( )
    t = getToken()
    if t != ADDOP:
        error
    parse_term()
```

- Do you see the problem?

Left Recursion

- Simple removal: Suppose we have production
  $A \rightarrow A\ \sigma \mid \pi$
  - σ and π = any sequence of one or more symbols
- Change it to:
  $A \rightarrow \pi\ A'$
  $A' \rightarrow \sigma\ A' \mid \lambda$
- It can be proven that new rules are equivalent to the old rules

Example

- Rule:
  expr $\rightarrow$ expr ADDOP term | term
- New rules:
  expr $\rightarrow$ term expr'
  expr' $\rightarrow$ ADDOP term expr' | λ

Multiple Rules

- If several left recursive rules: Best seen with an example:
  expr $\rightarrow$ expr PLUS term | expr MINUS term | term
- Change to:
  expr $\rightarrow$ term expr'
  expr' $\rightarrow$ PLUS term expr' | MINUS term expr' | λ

General Form

- Given: $A \rightarrow A\ \sigma_1 \mid A\ \sigma_2 \mid \ldots \mid A\ \sigma_n \mid \pi_1 \mid \pi_2 \mid \ldots \mid \pi_m$
- Transform to:
  $A \rightarrow \pi_1\ A' \mid \pi_2\ A' \mid \ldots \mid \pi_m\ A'$
  $A' \rightarrow \sigma_1\ A' \mid \sigma_2\ A' \mid \ldots \mid \sigma_n\ A' \mid \lambda$

Full Example

- Consider arithmetic expression grammar:
  expr $\rightarrow$ expr ADDOP term | expr SUBOP term | term
  term $\rightarrow$ term MULOP factor | term DIVOP factor | factor
  factor $\rightarrow$ ID | NUM | LP expr RP
- Rewrite: Do in class

λ productions

- Some automated parse techniques don't work well with λ productions
- We can transform any grammar with λ productions to equivalent one without
  - Except if grammar accepts λ
  - We can handle this as special case if needed

Idea

- First, compute all *nullable* symbols
- What's a nullable symbol?
  - One that goes (directly or indirectly) to λ

Nullable

```
let nullable = empty set
do
    for each nonterminal N:
        if N not in nullable:
            for all productions P with lhs of N:
                if all symbols in P are nullable:
                    if N is not in nullable:
                        nullable = union( nullable , N )
until nullable stabilizes
```

Example

- Consider:
  $S \rightarrow A \mid B \mid A\ A$
  $A \rightarrow x \mid y\ S \mid \lambda$
  $B \rightarrow A\ w \mid z$
- Compute nullable = { A,S }

Example

- Grammar:

$$S \rightarrow A\ B \mid A\ C\ z \mid x\ C\ y \mid x\ y \mid A\ x\ S\ y \mid C\ C$$
$$A \rightarrow x \mid y\ S \mid \lambda$$
$$B \rightarrow A\ w \mid z$$
$$C \rightarrow \lambda$$

- Nullable: {S,A,C}

Removing

- Now: To remove λ productions
- Scan grammar. Remove all λ productions
- Drop any nonterminals that now have no productions
  - Delete them from any rhs that had them

Example

- Consider previous grammar
- Remove λ productions

  $S \rightarrow A\ B \mid A\ C\ z \mid x\ C\ y \mid x\ y \mid A\ x\ S\ y$

  $A \rightarrow x \mid y\ S$

  $B \rightarrow A\ w \mid z$

  $C \rightarrow$

Example

- Remove symbols with no productions left (here, “C”)
  $S \rightarrow A\ B \mid A\ z \mid x\ y \mid x\ y \mid A\ x\ S\ y$
  $A \rightarrow x \mid y\ S$
  $B \rightarrow A\ w \mid z$
- Notice we have a duplicated rhs (in S)
  - We’ll deal with this later

Next

- Consider each production
- Compute all combinations of that production with and without its nullable symbols
- Example: $S \rightarrow A\ x\ S\ y$ (Suppose S and A are both nullable)
  - A x S y
  - x S y
  - A x y
  - x y
- How can we do this programmatically?

Option

- Suppose we are considering production P
- First, determine how many nullable symbols there are. Call this n.
- We will have 2^n combinations
  - If n == 0: No need to do anything with P

Nullable

- Cycle a counter c from $0 \ldots 2^n - 1$ inclusive
- If bit i of c is 1: Retain nullable symbol i. Else, discard it

Example

- Suppose we have S $\rightarrow$ A x S y
- Two nullable symbols, so c=0,1,2,3
- c=0: Retain neither nullable: x y
- c=1: Retain first nullable: A x y
- c=2: Retain second nullable: x S y
- c=3: Retain first and second nullables: A x S y
- How can we code this?

Code

- Scan production, noting wherever we have nullable symbol:

```
#nullable indices
#key = position in production P
#value = which nullable it is (1,2,3,...)
ni = {}
ncount=0
for i in range(len(P)):
    if P[i] is nullable:
        ni[i]=ncount
        ncount += 1
n = len(ni.keys())
```

Code

- Next, cycle the counter:

```
newProductions=[]
for ctr in range( 1 << ncount ):
    P'=[]
    for j in range(len(P)):
        if j in ni:
            if ctr & (1<<ni[j]):
                P'.append(P[j])
        else:
            P'.append(P[j])
    newProductions.append(P')
nonterminals[lhs] = newProductions
```

- P' represents one new production
- We'll generate 2^n such productions

Assignment

- Write code to compute and return the nullable set for a grammar.
- Your code must work with the test harness
- If you're having trouble getting started, here's a stub implementation that, while incorrect, at least compiles: ComputeNullable.cs

Sources

- Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques, & Tools (2nd ed). Addison-Wesley Publishing.
- K. Louden. Compiler Construction: Principles and Practice. PWS Publishing.

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