CSC488/2107 Winter 2019 — Compilers & Interpreters

https://www.cs.toronto.edu/~csc488h/

Peter McCormick
pdm@cs.toronto.edu

Agenda

- Implementing *LL(1)* parsers
- Recursive descent
- Ambiguous grammars
- PLY tutorial

Top Down Parsing

- Starting from the start symbol *S*, find the correct sequence of production expansions that will transform *S* into the input token stream (if possible)
  - Called a *derivation*
- Build the parse tree from the root (*S*) to the leaves (terminals)
- Top down techniques:
  - *LL(k)*: **L**eft-to-right, **L**eft-most derivation, ***k*** tokens worth of input lookahead
  - Recursive Descent

Implementing *LL(1)* parsers

Arithmetic Expressions: Ambiguous

$$S \longrightarrow E$$

$$E \longrightarrow \mathbf{id}$$

$$E \longrightarrow \mathbf{num}$$

$$E \longrightarrow E \; \mathbf{*} \; E$$

$$E \longrightarrow E \; \mathbf{/} \; E$$

$$E \longrightarrow E \; \mathbf{+} \; E$$

$$E \longrightarrow E \; \mathbf{-} \; E$$

$$E \longrightarrow \mathbf{(} \; E \; \mathbf{)}$$

Unambiguous, but not *LL(1)*

$S \longrightarrow E$

$E \longrightarrow T$

$E \longrightarrow E + T$

$E \longrightarrow E - T$

$T \longrightarrow T * F$

$T \longrightarrow T / F$

$T \longrightarrow F$

$F \longrightarrow \mathrm{id}$

$F \longrightarrow \mathrm{num}$

$F \longrightarrow (E)$

Left recursion in productions for E and T

Removing the recursion yields an *LL(1)* grammar

S ⟶ E

E ⟶ T

E ⟶ E + T

E ⟶ E - T

T ⟶ T * F

T ⟶ T / F

T ⟶ F

F ⟶ id

F ⟶ num

F ⟶ (E)

S ⟶ E

E ⟶ T E’

E’ ⟶ + T E’

E’ ⟶ - T E’

E’ ⟶

T ⟶ F T’

T’ ⟶ * F T’

T’ ⟶ / F T’

T’ ⟶

F ⟶ id

F ⟶ num

F ⟶ (E)

Nullable, First & Follow Sets

S ⟶ E $

E ⟶ T E'
E' ⟶ + T E'
E' ⟶ - T E'
E' ⟶

T ⟶ F T'
T' ⟶ * F T'
T' ⟶ / F T'
T' ⟶

F ⟶ id
F ⟶ num
F ⟶ (E)

	Nullable?	First	Follow
S		id num (	
E		id num (	) $
E'	Yes	+ -	) $
T		id num (	) + - $
T'	Yes	* /	) + - $
F		id num (	) + - * / $

$$\begin{aligned} Predict\,(X \to \gamma) &= First\,(\gamma) && \text{(if } \gamma \text{ is } \textit{not} \text{ nullable)} \\ &= First\,(\gamma) \cup Follow\,(X) && \text{(if } \gamma \textit{ is} \text{ nullable)} \end{aligned}$$

Predictive Table

1. *Advance input on matching token terminal*
2. *Given current left-most non-terminal X, lookup row X at input column y for production rewrite rule to apply*

Pop S
Push $ E

Terminals along X axis
Non-terminals along Y axis

	id	num	(	)	+
S	$S \longrightarrow E\ \$$				
E	$E \longrightarrow T\ E'$				
E'				$E' \longrightarrow$	$E' \longrightarrow + T\ E$

Pop **E'**
(nothing to push)

Nullable, First & Follow Sets

S ⟶ E $

E ⟶ T E’
E’ ⟶ + T E’
E’ ⟶ - T E’
E’ ⟶

T ⟶ F T’
T’ ⟶ * F T’
T’ ⟶ / F T’
T’ ⟶

F ⟶ id
F ⟶ num
F ⟶ (E)

	Nullable?	First	Follow
S		id num (	
E		id num (	) $
E’	Yes	+ -	) $
T		id num (	) + - $
T’	Yes	* /	) + - $
F		id num (	) + - * / $

$$\begin{aligned} Predict\,(X \to \gamma) &= First\,(\gamma) && \text{(if } \gamma \text{ is } \textit{not} \text{ nullable)} \\ &= First\,(\gamma) \cup Follow\,(X) && \text{(if } \gamma \text{ } \textit{is} \text{ nullable)} \end{aligned}$$

Predictive Table (*k=1*)

S → E $

E → T E’

E’ → + T E’

E’ → - T E’

E’ →

T → F T’

T’ → * F T’

T’ → / F T’

T’ →

F → id

F → num

F → (E)

	id	num	(	)	+	-	*	/	$
S	S → E $								
E	E → T E’								
E’				E’ →	E’ → + T E’	E’ → − T E’			E’ →
T	T → F T’								
T’				T’ →			T’ → * F T’	T’ → / F T’	T’ →
F	F → id	F → num	F → (E)						

Table-driven derivation

`1 + 2 * 3 ⟶ num + num * num`

	id	num	(	)	+	-	*	/	$
S	E $								
E	T E'								
E'				λ	(+ | -) T E'				λ
T	F T'								
T'				λ			(* | /) F T'		λ
F	id	num	(E)						

S	1 + 2 * 3 $
E $	1 + 2 * 3 $
T E' $	1 + 2 * 3 $
F T' E' $	1 + 2 * 3 $
num T' E' $	1 + 2 * 3 $
num T' E' $	+ 2 * 3 $
num E' $	+ 2 * 3 $
num + T E' $	+ 2 * 3 $
num + T E' $	2 * 3 $
num + F T' E' $	2 * 3 $
num + num T' E' $	2 * 3 $
num + num T' E' $	* 3 $
num + num * F T' E' $	* 3 $
*num + num ** F T' E' $	3 $
*num + num ** num T' E' $	3 $
*num + num * num* T' E' $	$
*num + num * num* E' $	$
*num + num * num* $	$
*num + num * num*	

Recursive Descent

Recursive descent

- Using the predictive table, we can manually write a top-down parser using recursive functions

Support interface

```
cur(): return current token

advance(): go to next token

eat(typ): advance if current token
is of type `typ`, otherwise error

error(): report unexpected input
error
```

```
def S():
  if cur() in [ ID,NUM,LPAREN ]:
    E()

  else: error()

def E():
  if cur() in [ ID,NUM,LPAREN ]:
    T()
    E1()

  else: error()

def T():
  if cur() in [ ID,NUM,LPAREN ]:
    F()
    T1()

  else: error()
```

```
def F():
  if cur() == ID:
    eat(ID)

  elif cur() == NUM:
    eat(NUM)

  elif cur() == LPAREN:
    eat(LPAREN)
    E()
    eat(RPAREN)

  else: error()
```

```
def E1():
  if cur() == PLUS:
    eat(PLUS)
    T()
    E1()

  elif cur() == MINUS:
    eat(MINUS)
    T()
    E1()

  elif cur() in [ RPAREN, EOF ]:
    pass

  else: error()
```

```
def T1():
  if cur() in [ PLUS, MINUS,
                RPAREN, EOF ]:
    pass

  elif cur() == STAR:
    eat(STAR)
    F()
    T1()

  elif cur() == SLASH:
    eat(SLASH)
    F()
    T1()

  else: error()
```

```
1 + (2 + 3) * (4 - 5) + 6
```

```
S:
  E:
    T:
      F:
        num(1)
      T1:
    E1:
      plus
      T:
        F:
          lparen
          E:
            T:
              F:
                num(2)
              T1:
            E1:
              plus
              T:
                F:
                  num(3)
                T1:
              E1:
          rparen
        T1:
          star
          F:
```

See *recdescent.py*

Ambiguous grammars

Ambiguous grammars

Example: C++

See <u>Willink (p147)</u>

```
int x;
int y;
```

```
int x, y;
```

```
int (x), (y);
```

```
int* x, y, z;
```

```
p = (q++, q*2);
```

```
p,  q*2;
```

```
int *m = new int;
*m = 488;
delete m;
```

```
int(x), y, *const z;
```

```
int(x), y, *new int;
```

```
int(x), y, *const z;

int(x), y, *new int;
```

```
int(x), y, a, b, c, …,*const z;

int(x), y, a, b, c, …, new int;
```

C++ is *inherently* ambiguous

Requires custom approaches to parse

Ambiguous grammars

Example:
CSC488 Source Language

Previously...

```
statements
    : …
    | “return” expression
    | “return”
    | procedureName “(“ “)”

expression
    : …
    | functionName “(“ “)”
```

```
return foo()
```

— vs —

```
return
foo()
```

```
return foo()
```

```
return
foo()
```

From Piazza:

"Could we build a parser that considers whether or not the return statement is inside a function or a procedure?"

```
func p() {
  return
  p()
}
```

Two statements

```
func f() integer {
  return f()
}
```

One statement

```
func p() {
  return p()
}
```

One statement w/ error

```
func f() integer {
  return
  f()
}
```

Two statements w/ 1 error (or maybe 2?)

```
def decl_func():
  eat(FUNC, ID, LPAREN)
  parameters()
  eat(RPAREN)

  if cur() in [ INTEGER, BOOLEAN ]:
    eat(cur())
    pushRoutine(isFunc=True)
    …
  else:
    pushRoutine(isProc=True)
    …

  scope()
  popRoutine()
```

```
def stmt_return():
  eat(RETURN)

  if curRoutine().isProc:
    if peek(NEWLINE):
      eat(NEWLINE)
      # bare “return"
    else:
      # "return" with value
      error('proc cannot
             return value’)
```

```
func p() {
  return
  p()
}
```

```
func p() {
  return p()
}
```

```
return bools[0] == true
```

— **vs** —

```
return
bools[0] = true
```

```
return xs[a+b+c+f(x+y+z+g(…))] == true
```

Python is *LL(1)*

https://github.com/python/cpython/blob/master/Grammar/Grammar

Python is *LL(1)*

- Semicolons as statement separators
- Matrix multiply @ operator
- try…else ?
- Productions for *term* and *factor*
- Indentation handling in *suite* production

Bottom Up Parsing

- Given a stream of input tokens, find the correct sequence of production contractions that take the input back to the start symbol
  - Called a *reduction* or *reverse derivation*
- Build the parse tree from the leaves (terminals) to the root (S)
- Bottom up techniques: *LR(k)*, *SLR(k)*, *LALR(k)*

PLY: Python Lex-Yacc

https://www.dabeaz.com/ply/ply.html

PLY: Python Lex-Yacc

- Specifying lexical scanners
- Specifying grammars
- Adding actions to productions
- Building values during a parse

Next Week

- *LR* grammars