Parse Table

Motivation

- Want to build a *transducer*
  - Simply accepts or rejects input
  - No other results
- Later, we'll build parse tree

Input

- We have a grammar with terminals and nonterminals
- Several steps needed
- First: Compute nullable, first, follow
- Next, we'll create a table to allow us to parse

Table

- Suppose grammar has t terminals and n nonterminals
- We create table with n rows and t+1 columns
  - Extra column is for $ (end of file metasymbol)
  - Each table cell will either be empty or else will contain a production
- Meaning: table[x][y] says “if you are expanding nonterminal x and terminal y is next one from input, you want this production”

Example

- For convenience, we could use a map-of-maps:
  Dictionary<string, Dictionary<string, Production> > table;

findfirst

- We need to define an operation:
  findfirst(P, e)
  - P = a production (list of symbols)
  - e = a set of terminals
- Result: Any symbol that can lead off any derivation of P
- If all symbols in P are nullable, result also includes symbols in e

Example

- Grammar:
  $S \rightarrow A\ y \mid z\ w$
  $A \rightarrow q \mid \lambda$
- Some example results:
  - findfirst([A y], {a,b,c}) = {q,y}
  - findfirst([z w], {a,b,c}) = {z}
  - findfirst([q] , {a,b,c}) = {q}
  - findfirst([A] , {a,b,c}) = {q,a,b,c}
  - findfirst([] , {a,b,c}) = {a,b,c}

findfirst

```
1  def findfirst(P,e):
2      s = set()
3      for x in P:
4          s = union( s, first[x] )
5          if x not nullable:
6              return s
7      return union( s, e )
```

Table

- Now we can build our LL table:

```
for all nonterminals N:
    for all productions P with lhs of N:
        for s in findfirst(P,follow[N]):
            table[N][s] = P
```

- If we try to set any table entry twice: Grammar is not LL(1)
  - Parser won't know which production to choose

Example

$S \rightarrow stmt\ ;\ S \mid \lambda$
$assign \rightarrow id = e$
$cond \rightarrow if\ (\ e\ )\ stmt \mid if\ (\ e\ )\ stmt\ else\ stmt$
$e \rightarrow e + t \mid t$
$f \rightarrow id \mid num \mid (\ e\ )$
$func\text{-}call \rightarrow id\ (\ e\ )$
$stmt \rightarrow assign \mid cond \mid func\text{-}call$
$t \rightarrow t * f \mid f$

First

(: (
):)
*: *
+: +
;: ;
=: =
S: id , if
assign: id
cond: if
e: (, id , num
else: else
f: (, id , num
func-call: id
id: id
if: if
num: num
stmt: id , if
t: (, id , num

Follow

S: $
assign: ; , else
cond: ; , else
e:) , + , ; , else
f:) , * , + , ; , else
func-call: ; , else
stmt: ; , else
t:) , * , + , ; , else

Table

- There are conflicts
  - Note: Empty columns are not shown here

	$	(	id	if	num
S	λ	•	stmt ; S	stmt ; S	•
assign	•	•	id = e	•	•
cond	•	•	•	if (e) stmt else stmt if (e) stmt	•
e	•	t e + t	t e + t	•	t e + t
f	•	(e)	id	•	num
func-call	•	•	id (e)	•	•
stmt	•	•	func-call assign	cond	•
t	•	t * f f	t * f f	•	t * f f

Fix Grammar

- Need to remove left recursion.

$S \rightarrow \text{stmt} ; S \mid \lambda$
$\text{assign} \rightarrow \text{id} = e$
$\text{cond} \rightarrow \text{if} (e) \text{stmt} \mid \text{if} (e) \text{stmt}$
else stmt
$e \rightarrow t\ e'$
$e' \rightarrow + t\ e' \mid \lambda$

$f \rightarrow \text{id} \mid \text{num} \mid (e)$
$\text{func-call} \rightarrow \text{id} (e)$
$\text{stmt} \rightarrow \text{assign} \mid \text{cond} \mid \text{func-call}$
$t \rightarrow f\ t'$
$t' \rightarrow * f\ t' \mid \lambda$

First

(: (
):)
*: *
+: +
;: ;
=: =
S: id , if
assign: id
cond: if
e: (, id , num
e': +
else: else
f: (, id , num
func-call: id
id: id
if: if
num: num
stmt: id , if
t: (, id , num
t': *

Follow

S: $
assign: ; , else
cond: ; , else
e:) , ; , else
e':) , ; , else
f:) , * , + , ; , else
func-call: ; , else
stmt: ; , else
t:) , + , ; , else
t':) , + , ; , else

Table

- Notice there's another conflict!

	$	(	)	*	+	;	else	id	if	num
S	λ	•	•	•	•	•	•	stmt ; S	stmt ; S	•
assign	•	•	•	•	•	•	•	id = e	•	•
cond	•	•	•	•	•	•	•	•	if (e) stmt if (e) stmt else stmt	•
e	•	t e'	•	•	•	•	•	t e'	•	t e'
e'	•	•	λ	•	+ t e'	λ	λ	•	•	•
f	•	(e)	•	•	•	•	•	id	•	num
func-call	•	•	•	•	•	•	•	id (e)	•	•
stmt	•	•	•	•	•	•	•	assign func-call	cond	•
t	•	f t'	•	•	•	•	•	f t'	•	f t'
t'	•	•	λ	* f t'	λ	λ	λ	•	•	•

Grammar

- Need to left factor as well

$S \rightarrow stmt\ ;\ S \mid \lambda$
$assign \rightarrow id = e$
$cond \rightarrow if\ (\ e\ )\ stmt\ cond'$
$cond' \rightarrow \lambda \mid else\ stmt$
$e \rightarrow t\ e'$
$e' \rightarrow +\ t\ e' \mid \lambda$

$f \rightarrow id \mid num \mid (\ e\ )$
$func\text{-}call \rightarrow id\ (\ e\ )$
$stmt \rightarrow assign \mid cond \mid func\text{-}call$
$t \rightarrow f\ t'$
$t' \rightarrow *\ f\ t' \mid \lambda$

First

(: (
):)
*: *
+: +
;: ;
=: =
S: id , if
assign: id
cond: if
cond': else
e: (, id , num
e': +
else: else
f: (, id , num
func-call: id
id: id
if: if
num: num
stmt: id , if
t: (, id , num
t': *

Follow

S: $
assign: ; , else
cond: ; , else
cond': ; , else
e:) , ; , else
e':) , ; , else

f:) , * , + , ; , else
func-call: ; , else
stmt: ; , else
t:) , + , ; , else
t':) , + , ; , else

Table

- Two remaining conflicts

	$	(	)	*	+	;	else	id	if	num
S	λ	•	•	•	•	•	•	stmt ; S	stmt ; S	•
assign	•	•	•	•	•	•	•	id = e	•	•
cond	•	•	•	•	•	•	•	•	if (e) stmt cond'	•
cond'	•	•	•	•	•	λ	λ else stmt	•	•	•
e	•	t e'	•	•	•	•	•	t e'	•	t e'
e'	•	•	λ	•	+ t e'	λ	λ	•	•	•
f	•	(e)	•	•	•	•	•	id	•	num
func-call	•	•	•	•	•	•	•	id (e)	•	•
stmt	•	•	•	•	•	•	•	func-call assign	cond	•
t	•	f t'	•	•	•	•	•	f t'	•	f t'
t'	•	•	λ	* f t'	λ	λ	λ	•	•	•

Problem

- Problem: Both assign and func-call can lead off with ‘id’
- And stmt $\rightarrow$ assign and stmt $\rightarrow$ func-call
- So we must alter the grammar again:

S $\rightarrow$ stmt ; S | λ
aof $\rightarrow$ id aof’
aof’ $\rightarrow$ = e | (e)
cond $\rightarrow$ if (e) stmt cond’
cond’ $\rightarrow$ λ | else stmt
e $\rightarrow$ t e’

e’ $\rightarrow$ + t e’ | λ
f $\rightarrow$ id | num | (e)
stmt $\rightarrow$ aof | cond
t $\rightarrow$ f t’
t’ $\rightarrow$ * f t’ | λ

First

(: (
):)
*: *
+: +
;: ;
=: =
S: id , if
aof: id
aof': (, =
cond: if
cond': else
e: (, id , num
e': +
else: else
f: (, id , num
id: id
if: if
num: num
stmt: id , if
t: (, id , num
t': *

Follow

S: $
aof: ; , else
aof': ; , else
cond: ; , else
cond': ; , else
e:) , ; , else
e':) , ; , else
f:) , * , + , ; , else
stmt: ; , else
t:) , + , ; , else
t':) , + , ; , else

Table

- It's *still* not LL(1)

	$	(	)	*	+	;	=	else	id	if	num
S	λ	•	•	•	•	•	•	•	stmt ; S	stmt ; S	•
aof	•	•	•	•	•	•	•	•	id aof'	•	•
aof'	•	(e)	•	•	•	•	= e	•	•	•	•
cond	•	•	•	•	•	•	•	•	•	if (e) stmt cond'	•
cond'	•	•	•	•	•	λ	•	else stmt λ	•	•	•
e	•	t e'	•	•	•	•	•	•	t e'	•	t e'
e'	•	•	λ	•	+ t e'	λ	•	λ	•	•	•
f	•	(e)	•	•	•	•	•	•	id	•	num
stmt	•	•	•	•	•	•	•	•	aof	cond	•
t	•	f t'	•	•	•	•	•	•	f t'	•	f t'
t'	•	•	λ	* f t'	λ	λ	•	λ	•	•	•

Problem

- Grammar is ambiguous
  - All grammars that are ambiguous fail to create valid LL(1) parse tables
- Here, if we see cond': We have two choices:
  - Select 'else stmt'
  - If we see anything in follow[cond']: Select λ
  - Problem: 'else' appears in follow[cond']

Problem

- We don’t know if we should parse:
  if(x) if(y) a=1 else a=2
- As:
  if(x){ if(y) a=1 else a=2 }
- Or:
  if(x){ if(y) a=1 } else { a=2 }
- If grammar is ambiguous, table will be too...

Solution

- We could fix by changing the grammar: Force specific indication of where blocks begin and end:

$S \rightarrow \text{stmt} \ ; \ S \mid \lambda$
$\text{aof} \rightarrow \text{id aof'}$
$\text{aof'} \rightarrow = \text{e} \mid (\text{e})$
$\text{cond} \rightarrow \text{if} (\text{e}) \{ \text{stmt} \} \text{ cond'}$
$\text{cond'} \rightarrow \lambda \mid \text{else} \{ \text{stmt} \}$
$\text{e} \rightarrow \text{t e'}$

$\text{e'} \rightarrow + \text{t e'} \mid \lambda$
$\text{f} \rightarrow \text{id} \mid \text{num} \mid (\text{e})$
$\text{stmt} \rightarrow \text{aof} \mid \text{cond}$
$\text{t} \rightarrow \text{f t'}$
$\text{t'} \rightarrow * \text{f t'} \mid \lambda$

First

(: (
):)
*: *
+: +
;: ;
=: =
S: id , if
aof: id
aof': (, =
cond: if
cond': else
e: (, id , num
e': +
else: else
f: (, id , num
id: id
if: if
num: num
stmt: id , if
t: (, id , num
t': *
{: {
}: }

Follow

S: $
aof: ; , }
aof': ; , }
cond: ; , }
cond': ; , }
e:) , ; , }
e':) , ; , }
f:) , * , + , ; , }
stmt: ; , }
t:) , + , ; , }
t':) , + , ; , }

Table

- Finally, we have a working grammar

	$	(	)	*	+	;	=	else	id	if	num	}
S	λ	•	•	•	•	•	•	•	stmt ; S	stmt ; S	•	•
aof	•	•	•	•	•	•	•	•	id aof'	•	•	•
aof'	•	(e)	•	•	•	•	= e	•	•	•	•	•
cond	•	•	•	•	•	•	•	•	•	if (e) { stmt } cond'	•	•
cond'	•	•	•	•	•	λ	•	else { stmt }	•	•	•	λ
e	•	t e'	•	•	•	•	•	•	t e'	•	t e'	•
e'	•	•	λ	•	+ t e'	λ	•	•	•	•	•	λ
f	•	(e)	•	•	•	•	•	•	id	•	num	•
stmt	•	•	•	•	•	•	•	•	aof	cond	•	•
t	•	f t'	•	•	•	•	•	•	f t'	•	f t'	•
t'	•	•	λ	* f t'	λ	λ	•	•	•	•	•	λ

Assignment

- Write a program which computes the LL(1) parse table
- Your program must work with the test harness

Sources

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

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