Principles of Programming Languages

CS 314

Recitation 3

RUTGERS

Topics Today

LL(1) Parsing

Recursive Descent Parsing

LL(1) Parsing

Recursive Descent Parsing

LL(1) Parsing – Example 1

Consider the following grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

Is this grammar LL(1)?

Make the FIRST sets (and if necessary, the FOLLOW sets) for this grammar, then construct the LL(1) parse table.

LL(1) Parsing – Example 1

The grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

has the following FIRST sets:

FIRST(<ID> <decl_tail>) = {“a”, “b”, “c”}

FIRST(, <decl>) = {“,”}

FIRST(= <ID> ;) = {“=”}

FIRST(a) = {“a”}

FIRST(b) = {“b”}

FIRST(c) = {“c”}

Recall that PREDICT(A ::= δ) = (FIRST(δ) - {ε}) ∪ FOLLOW(A) if ε is an element of FIRST(δ).
Otherwise, PREDICT(A ::= δ) = FIRST(δ).

LL(1) Parsing – Example 1

The grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

There is no ε in any of the FIRST sets, so PREDICT(A ::= δ) = FIRST(δ). We have the following:

PREDICT(<decl> ::= <ID> <decl_tail>) = FIRST(<ID> <decl_tail>) = {“a”, “b”, “c”}

PREDICT(<decl_tail> ::= , <decl>) = FIRST(, <decl>) = {“,”}

PREDICT(<decl_tail> ::= =<ID> ;) = FIRST(= <ID> ;) = {“=”}

PREDICT(<ID> ::= a) = FIRST(a) = {“a”}

PREDICT(<ID> ::= b) = FIRST(b) = {“b”}

PREDICT(<ID> ::= c) = FIRST(c) = {“c”}

Since no two production rules for the same non-terminal have overlap in their PREDICT sets, the grammar is LL(1). Now, for the parsing table.

LL(1) Parsing – Example 1

Grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

Since FIRST(<ID> <decl_tail>) = {“a”, “b”, “c”}

	,	=	a	b	c
<decl>			<ID> <decl_tail>	<ID> <decl_tail>	<ID> <decl_tail>
<decl_tail>					
<ID>					

LL(1) Parsing – Example 1

Grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

If the derivation of <decl_tail> starts with ",", the grammar follows the rule: <decl_tail> ::= , <decl>

	,	=	a	b	c
<decl>			<ID> <decl_tail>	<ID> <decl_tail>	<ID> <decl_tail>
<decl_tail>	, <decl>				
<ID>					

LL(1) Parsing – Example 1

Grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

If the derivation of <decl_tail> starts with "=" , the grammar follows the rule <decl_tail> ::= = <ID> ;

	,	=	a	b	c
<decl>			<ID> <decl_tail>	<ID> <decl_tail>	<ID> <decl_tail>
<decl_tail>	, <decl>	= <ID> ;			
<ID>					

LL(1) Parsing – Example 1

Grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

Following the grammar, the derivation of <ID> can become either a, b, or c.

	,	=	a	b	c
<decl>			<ID> <decl_tail>	<ID> <decl_tail>	<ID> <decl_tail>
<decl_tail>	, <decl>	= <ID> ;			
<ID>			a	b	c

LL(1) Parsing – Example 1

Grammar:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

Note that the LL(1) parse table has no conflicts (i.e., each cell in the table only has one entry).

	,	=	a	b	c
<decl>			<ID> <decl_tail>	<ID> <decl_tail>	<ID> <decl_tail>
<decl_tail>	, <decl>	= <ID> ;			
<ID>			a	b	c

LL(1) Parsing – Example 2

Consider the following modified grammar:

<decl> ::= <ID> <mid> <tail>
<mid> ::= <mid> , <ID> | ε
<tail> ::= : <ID> ;
<id> ::= a | b | c

Where the rules for <mid> are: <mid> ::= <mid> , <ID> and <mid> ::= ε

Is this grammar still LL(1)?

Make the FIRST sets (and if necessary, FOLLOW sets) for this grammar, then construct the LL(1) parse table.

LL(1) Parsing – Example 2

The grammar:

<decl> ::= <ID> <mid> <tail>
<mid> ::= <mid> , <ID> | ε
<tail> ::= : <ID> ;
<ID> ::= a | b | c

has the following FIRST sets:

FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(: <ID> ;) = {:}
FIRST(<ID> <mid> <tail>) = {a, b, c}

But what about FIRST sets for <mid>'s rules?

LL(1) Parsing – Example 2

Since <mid> ::= ε is a rule,

FIRST(<mid>) = {ε, ","}

FIRST(ε) = {ε}

Recall that PREDICT(A ::= δ) = (FIRST(δ) - {ε}) ∪ FOLLOW(A) if ε is an element of FIRST(δ).
Otherwise, PREDICT(A ::= δ) = FIRST(δ).

Since ε is an element of FIRST(ε), PREDICT(<mid> ::= ε) = (FIRST(ε) - {ε}) ∪ FOLLOW(<mid>).

So, for the rule <mid> ::= ε:

We'll need to know what FOLLOW(<mid>) is in order to compute this rule's PREDICT set.

LL(1) Parsing – Example 2

For the grammar:

<decl> ::= <ID> <mid> <tail>

<mid> ::= <mid> , <ID> | ε

<tail> ::= : <ID> ;

<ID> ::= a | b | c

<mid> appears in the right side of two rules:

- <decl> ::= <ID> <mid> <tail>
- <mid> ::= <mid> , <ID>

So, the FOLLOW set for <mid> would be:

FOLLOW(<mid>) = FIRST(<tail>) ∪ {“,”}

= {: , “,”}

Now we can find the PREDICT set:

PREDICT(<mid>::=ε) = (FIRST(ε) – {ε}) ∪ {:, “,”} = {:, “,”}

LL(1) Parsing – Example 2

For the grammar:

<decl> ::= <ID> <mid> <tail>

<mid> ::= <mid> , <ID> | ε

<tail> ::= : <ID> ;

<ID> ::= a | b | c

Since FIRST(<mid> , <ID>) = {“,”}, ε is not in FIRST(<mid> , <ID>), so

PREDICT(<mid> ::= <mid> , <ID>) = FIRST(<mid> , <ID>) = {“,”}

Thus, the grammar is not LL(1), since there is overlap between the PREDICT sets of the rules <mid>::=ε and <mid> ::= <mid> , <ID>

PREDICT(<mid>::=ε) = (FIRST(ε) – {ε}) ∪ {:, “,”} = {:, “,”}

PREDICT(<mid> ::= <mid> , <ID>) = FIRST(<mid> , <ID>) = {“,”}

The overlap is the element “,”.

LL(1) Parsing – Example 2

The modified Grammar:

<decl> ::= <ID> <mid> <tail>
<mid> ::= <mid> , <ID> | ε
<tail> ::= : <ID> ;
<ID> ::= a | b | c

Since FIRST(<ID> <mid> <tail>) = {a, b, c} :

	,	:	a	b	c
<decl>			<ID> <mid> <tail>	<ID> <mid> <tail>	<ID> <mid> <tail>
<mid>					
<tail>					
<ID>					

LL(1) Parsing – Example 2

The modified Grammar:

<decl> ::= <ID> <mid> <tail>
<mid> ::= <mid> , <ID> | ε
<tail> ::= : <ID> ;
<ID> ::= a | b | c

Since FIRST(<mid> , <ID>) = {“,”} and <mid> ::= <mid> , <ID>

	,	:	a	b	c
<decl>			<ID> <mid> <tail>	<ID> <mid> <tail>	<ID> <mid> <tail>
<mid>	<mid> , <ID>				
<tail>					
<ID>					

LL(1) Parsing – Example 2

The modified Grammar:

<decl> ::= <ID> <mid> <tail>
<mid> ::= <mid> , <ID> | ε
<tail> ::= : <ID> ;
<ID> ::= a | b | c

Since FIRST(ε) = {ε}, PREDICT(<mid>::=ε) = {:, “,”} and <mid> ::= ε

	,	:	a	b	c
<decl>			<ID> <mid> <tail>	<ID> <mid> <tail>	<ID> <mid> <tail>
<mid>	<mid> , <ID> ε				
<tail>					
<ID>					

LL(1) Parsing – Example 2

The modified Grammar:

<decl> ::= <ID> <mid> <tail>
<mid> ::= <mid> , <ID> | ε
<tail> ::= : <ID> ;
<ID> ::= a | b | c

Since PREDICT(<mid>::=ε) = {":", ","}, if the symbol read is ":", then <mid> ::= ε applies.

	,	:	a	b	c
<decl>			<ID> <mid> <tail>	<ID> <mid> <tail>	<ID> <mid> <tail>
<mid>	<mid> , <ID> ε	ε			
<tail>					
<ID>					

LL(1) Parsing – Example 2

The modified Grammar:

<decl> ::= <ID> <mid> <tail>
<mid> ::= <mid> , <ID> | ε
<tail> ::= : <ID> ;
<ID> ::= a | b | c

Since FIRST(a) = {a}, FIRST(b) = {b}, and FIRST(c) = {c},

	,	:	a	b	c
<decl>			<ID> <mid> <tail>	<ID> <mid> <tail>	<ID> <mid> <tail>
<mid>	<mid> , <ID> ε	ε			
<tail>		: <ID> ;			
<ID>			a	b	c

LL(1) Parsing – Example 2

Notice that the parse table has a cell with two entries.

There's two choices that can be made in the circled area: "<mid>, <ID>" and ε.

So, 1 symbol of look-ahead is not enough to decide between these two rules:

<mid> ::= <mid> , <ID>

<mid> ::= ε

	,	:	a	b	c
<decl>			<ID> <mid> <tail>	<ID> <mid> <tail>	<ID> <mid> <tail>
<mid>	<mid> , <ID> ε	ε			
<tail>		: <ID> ;			
<ID>			a	b	c

Recursive Descent Parsing

Recursive Descent Parsing – Example 1

Consider the following grammar from Example 1 again:

<decl> ::= <ID> <decl_tail>

<decl_tail> ::= , <decl> | = <ID> ;

<ID> ::= a | b | c

Recall that this grammar had the following LL(1) parse table:

	,	=	a	b	c
<decl>			<ID> <decl_tail>	<ID> <decl_tail>	<ID> <decl_tail>
<decl_tail>	, <decl>	= <ID> ;			
<ID>			a	b	c

Now we construct a recursive descent parser for this grammar, with routines for each non-terminal. To handle the look-ahead symbol, the global variable ‘token’ and function ‘next_token()’ will be used.

Recursive Descent Parsing – Example 1

	,	=	a	b	c
<decl>			<ID> <decl_tail>		
<decl_tail>	, <decl>	= <ID> ;			
<ID>			a	b	c

Since there's three non-terminals:

- <decl>
- <decl_tail>
- <ID>

Each of these non-terminals will have its own method, reading the next token until parsing ends or it reaches an error, and returns true only if parsing was successful.

The main program:

```
void program() {
    token = next_token() //initialize starting token
    if (decl() && token == eof())
        //if parsing <decl> returns true and we reached the end of string
        print("Parsed successfully.")
    else
        print("Error!")
    }
```

Recursive Descent Parsing – Example 1

	,	=	a	b	c
<decl>			<ID> <decl_tail>		
<decl_tail>	, <decl>	= <ID> ;			
<ID>			a	b	c

```
bool decl() {
    if(token != “a” && token != “b” && token != “c”)
        //If starting token is not a, b, or c, it does not start with <ID>, so return failure
        return false
    if(!ID())
        //If parsing <ID> has failed, then return failure
        return false
    if(!decl_tail())
        //Once <ID> is done, the method parses <decl_tail>. If it failed, then return failure
        return false
    return true
}
```

Recursive Descent Parsing – Example 1

	,	=	a	b	c
<decl>			<ID> <decl_tail>		
<decl_tail>	, <decl>	= <ID> ;			
<ID>			a	b	c

```
bool decl_tail() {
    if(token == ",")
        //parsing should be successful for ", <decl>"
        token = next_token()
        if(!decl()) //if subsequent parsing for <decl> fails, return failure
            return false
    else if(token == "=") //subsequent parsing should be successful for "= <ID> ;"
        token = next_token()
        if(!ID()) //if subsequent parsing for <ID> fails, return failure
            return false
        if(token != ";") //parsing this part should end with ;
            return false
        token = next_token()
    else //neither ", <decl>" nor "= <ID> ;" was parsed, return failure
        return false
    return true
}
```

Recursive Descent Parsing – Example 1

	,	=	a	b	c
<decl>			<ID> <decl_tail>		
<decl_tail>	, <decl>	= <ID> ;			
<ID>			a	b	c

```
bool ID() {
    if(token == “a” || token == “b” || token == “c”)
        //if the current is either a, b, or c, parsing was successful
        token = next_token()
    else
        //if the current token is not a, b, or c, it does not follow the ID rule, return failure
        return false
    return true
}
```