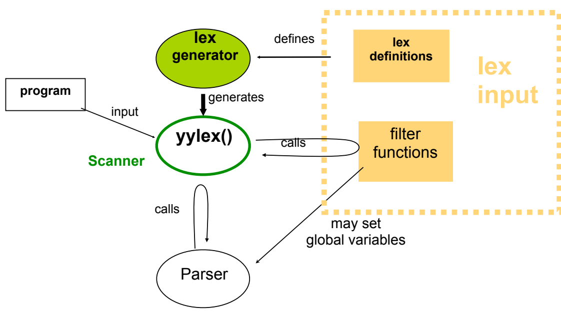
Announcements

- You may optionally work with a partner on the project
  - Must work with the partner for the entire project
  - Let me know by next Thursday who you are working with (if anyone), and under which username you will be submitting
- I'm trying to get the lectures posted online at least a day before class (some days I'm more successful than others!)

From last time: Lex (Flex)

- Commonly used Unix scanner generator (superseded by Flex)
- Has character classes and regular expressions like ScanGen but some key differences:
  - After each token is matched, calls user-defined "filter" function, which processes identified token before returning it to parser
    - Hence, no "Toss" facility (why?)
  - No exception list
    - Instead, supports matching multiple regexps.
      - Matches longest token (i.e., doesn't think `ifa` is `IF ID(a)`)
      - In case of tie, returns earliest-defined regexp
        - To treat `if` as a reserved word instead of an identifier, define token `IF` before defining identifiers.

Lex operation



Example of Lex input on page 67 of textbook

Parsers

Agenda

- Terminology
- LL(1) Parsers
- Overview of LR Parsing

Terminology

- Grammar $G = (V_t, V_n, S, P)$
  - V_t is the set of *terminals*
  - V_n is the set of *non-terminals*
  - S is the *start symbol*
  - P is the set of *productions*
    - Each production takes the form: $V_n \to \lambda \mid (V_n \mid V_t)+$
    - Grammar is *context-free* (why?)
- A simple grammar:

  $G = (\{a, b\}, \{S, A, B\}, \{S \to A\ B\ \$, A \to A\ a, A \to a, B \to B\ b, B \to b\}, S)$

Terminology

- V is the *vocabulary* of a grammar, consisting of terminal (V_t) and non-terminal (V_n) symbols
- For our sample grammar
  - V_n = {S, A, B}
    - Non-terminals are symbols on the LHS of a production
    - Non-terminals are constructs in the language that are recognized during parsing
  - V_t = {a, b}
    - Terminals are the tokens recognized by the scanner
    - They correspond to symbols in the text of the program

Terminology

- Productions (rewrite rules) tell us how to derive strings in the language
  - Apply productions to rewrite strings into other strings
- We will use the standard BNF form
- P = {

  S → A B $

  A → A a

  A → a

  B → B b

  B → b

  }

Generating strings

S → A B $

A → A a

A → a

B → B b

B → b

- Given a start rule, productions tell us how to rewrite a non-terminal into a different set of symbols
- By convention, first production applied has the start symbol on the left, and there is only one such production

To derive the string "a a b b b" we can do the following rewrites:

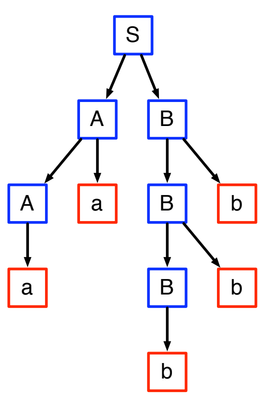
S ⇒ A B $ ⇒ A a B $ ⇒ a a B $ ⇒ a a B b $ ⇒ a a B b b $ ⇒ a a b b b $

Terminology

- *Strings* are composed of symbols
  - A A a a B b b A a is a string
  - We will use Greek letters to represent strings composed of both terminals and non-terminals
- *L(G)* is the language produced by the grammar *G*
  - All strings consisting of only terminals that can be produced by G
  - In our example, L(G) = a+b+$
  - All regular expressions can be expressed as grammars for context-free languages, but not vice-versa
    - Consider: $a^i b^i$ $ (what is the grammar for this?)

Parse trees

- Tree which shows how a string was produced by a language
  - Interior nodes of tree: non-terminals
    - Children: the terminals and non-terminals generated by applying a production rule
  - Leaf nodes: terminals



Leftmost derivation

- Rewriting of a given string starts with the leftmost symbol
- Exercise: do a leftmost derivation of the input program

  F(V + V)

  using the following grammar:

E	→	Prefix (E)
E	→	V Tail
Prefix	→	F
Prefix	→	λ
Tail	→	+ E
Tail	→	λ

- What does the parse tree look like?

Rightmost derivation

- Rewrite using the rightmost non-terminal, instead of the left
- What is the rightmost derivation of this string?

F(V + V)

E	→	Prefix (E)
E	→	V Tail
Prefix	→	F
Prefix	→	λ
Tail	→	+ E
Tail	→	λ

Top-down vs. Bottom-up parsers

- Top-down parsers use left-most derivation
- Bottom-up parsers use right-looking parse
- Notation:
  - LL(1): Leftmost derivation with 1 symbol lookahead
  - LL(k): Leftmost derivation with k symbols lookahead
  - LR(1): Right-looking derivation with 1 symbol lookahead

Micro in standard BNF

1	Program	::=	**BEGIN** Statement-list **END**
2	Statement-list	::=	Statement StatementTail
3	StatementTail	::=	Statement StatementTail
4	StatementTail	::=	λ
5	Statement	::=	**ID :=** Expression **;**
6	Statement	::=	**READ (** Id-list **) ;**
7	Statement	::=	**WRITE (** Expr-list **) ;**
8	Id-list	::=	**ID** IdTail
9	IdTail	::=	**, ID** IdTail
10	IdTail	::=	λ
11	Expr-list	::=	Expression ExprTail
12	ExprTail	::=	**,** Expression ExprTail
13	ExprTail	::=	λ
14	Expression	::=	Primary PrimaryTail
15	PrimaryTail	::=	Add-op Primary PrimaryTail
16	PrimaryTail	::=	λ
17	Primary	::=	**(** Expression **)**
18	Primary	::=	**ID**
19	Primary	::=	**INTLITERAL**
20	Add-op	::=	**PLUSOP**
21	Add-op	::=	**MINUSOP**
22	System-goal	::=	Program **SCANEOF**

Compare this to grammar in lecture 2

Micro in standard BNF

1	Program	::=	**BEGIN** Statement-list **END**
2	Statement-list	::=	Statement StatementTail
3	StatementTail	::=	Statement StatementTail
4	StatementTail	::=	λ
5	Statement	::=	**ID :=** Expression **;**
6	Statement	::=	**READ (** Id-list **) ;**
7	Statement	::=	**WRITE (** Expr-list **) ;**
8	Id-list	::=	**ID** IdTail
9	IdTail	::=	**, ID** IdTail
10	IdTail	::=	λ
11	Expr-list	::=	Expression ExprTail
12	ExprTail	::=	**,** Expression ExprTail
13	ExprTail	::=	λ
14	Expression	::=	Primary PrimaryTail
15	PrimaryTail	::=	Add-op Primary PrimaryTail
16	PrimaryTail	::=	λ
17	Primary	::=	**(** Expression **)**
18	Primary	::=	**ID**
19	Primary	::=	**INTLITERAL**
20	Add-op	::=	**PLUSOP**
21	Add-op	::=	**MINUSOP**
22	System-goal	::=	Program **SCANEOF**

A ::= B | C
↕
A ::= B
A ::= C

Compare this to grammar in lecture 2

Micro in standard BNF

1	Program	::=	**BEGIN** Statement-list **END**
2	Statement-list	::=	Statement StatementTail
3	StatementTail	::=	Statement StatementTail
4	StatementTail	::=	λ
5	Statement	::=	**ID :=** Expression **;**
6	Statement	::=	**READ (** Id-list **) ;**
7	Statement	::=	**WRITE (** Expr-list **) ;**
8	Id-list	::=	**ID** IdTail
9	IdTail	::=	**, ID** IdTail
10	IdTail	::=	λ
11	Expr-list	::=	Expression ExprTail
12	ExprTail	::=	**,** Expression ExprTail
13	ExprTail	::=	λ
14	Expression	::=	Primary PrimaryTail
15	PrimaryTail	::=	Add-op Primary PrimaryTail
16	PrimaryTail	::=	λ
17	Primary	::=	**(** Expression **)**
18	Primary	::=	**ID**
19	Primary	::=	**INTLITERAL**
20	Add-op	::=	**PLUSOP**
21	Add-op	::=	**MINUSOP**
22	System-goal	::=	Program **SCANEOF**

A ::= B | C
↕
A ::= B
A ::= C

A ::= B {C}
↕
A ::= B tail
tail ::= C tail
tail ::= λ

Compare this to grammar in lecture 2

What is parsing

- Parsing is recognizing members in a language specified/defined/generated by a grammar
- When a construct (corresponding to a production in a grammar) is recognized, a typical parser will take some action
  - In a compiler, this action generates an intermediate representation of the program construct
  - In an interpreter, this action might be to perform the action specified by the construct. Thus, if *a+b* is recognized, the value of *a* and *b* would be added and placed in a temporary variable

Another simple grammar

PROGRAM → begin STMTLIST $

STMTLIST → STMT ; STMTLIST

STMTLIST → end

STMT → id

STMT → if (id) STMTLIST

- A sentence in the grammar:

  begin if (id) if (id) id ; end; end; end; $

- What are the terminals and non-terminals of this grammar?

Parsing this grammar

PROGRAM → begin STMTLIST $

STMTLIST → STMT ; STMTLIST

STMTLIST → end

STMT → id

STMT → if (id) STMTLIST

- Note
  - To parse STMT in STMTLIST → STMT; STMTLIST, it is necessary to parse either STMT → id or STMT → if ...
  - Choose the production to parse by finding out if next token is if or id
    - i.e., which production the next input token matches
    - This is the *first* set of the production

Another example

S → A B $

A → x a A

A → y a A

A → λ

B → b

- Consider S ⇒ A B $ ⇒ x a A B $ ⇒ x a B $ ⇒ x a b $
- When parsing x a b $ we know from the goal production we need to match an A. The next token is x, so we apply A → x a A
- The parser matches x, matches a and now needs to parse A again
- How do we know which A to use? We need to use A → λ
  - When matching the right hand side of A → λ, the next token comes from a non-terminal that follows A (i.e., it must be b)
  - Tokens that can follow A are called the *follow* set of A

First and follow sets

- First(α): the set of terminals that begin all strings that can be derived from α
  - First(A) = {x, y}
  - First(xaA) = {x}
  - First (AB) = {x, y, b}
- Follow(A): the set of terminals that can appear immediately after A in some partial derivation
  - Follow(A) = {b}

S → A B $

A → x a A

A → y a A

A → λ

B → b

First and follow sets

- $\text{First}(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$
- $\text{Follow}(A) = \{a \in V_t \mid S \Rightarrow^+ \ldots Aa \ldots\} \cup \{\$ \mid \text{if } S \Rightarrow^+ \ldots A\ \$\}$

S:	start symbol
a:	a terminal symbol
A:	a non-terminal symbol
α,β:	a string composed of terminals and non-terminals (typically, α is the RHS of a production

$\Rightarrow$:	derived in 1 step
$\Rightarrow^*$:	derived in 0 or more steps
$\Rightarrow^+$:	derived in 1 or more steps

Computing first sets

- Terminal: First(a) = {a}
- Non-terminal: First(A)
  - Look at all productions for A

    $A \rightarrow X_1X_2 \ldots X_k$
  - $\text{First}(A) \supseteq (\text{First}(X_1) - \lambda)$
  - If $\lambda \in \text{First}(X_1)$, $\text{First}(A) \supseteq (\text{First}(X_2) - \lambda)$
  - If λ is in $\text{First}(X_i)$ for all i, then $\lambda \in \text{First}(A)$
- Computing First(α): similar procedure to computing First(A)

Exercise

- What are the first sets for all the non-terminals in following grammar:

$S \to A\ B\ \$$

$A \to x\ a\ A$

$A \to y\ a\ A$

$A \to \lambda$

$B \to b$

Computing follow sets

- Follow(S) = {\$}
- To compute Follow(A):
  - Find productions which have A on rhs. Three rules:
    1. $X \to \alpha\ A\ \beta$: Follow(A) $\supseteq$ (First(β) - λ)
    2. $X \to \alpha\ A\ \beta$: If $\lambda \in$ First(β), Follow(A) $\supseteq$ Follow(X)
    3. $X \to \alpha\ A$: Follow(A) $\supseteq$ Follow(X)
- Note: Follow(X) never has λ in it.

Exercise

- What are the follow sets for

$S \to A\ B\ \$$

$A \to x\ a\ A$

$A \to y\ a\ A$

$A \to \lambda$

$B \to b$

Towards parser generators

- Key problem: as we read the source program, we need to decide what productions to use
- Step 1: find the tokens that can tell which production P (of the form $A \to X_1 X_2 \ldots X_m$) applies

$$\text{Predict}(P) =$$

$$\begin{cases} \text{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \text{First}(X_1 \ldots X_m) \\ (\text{First}(X_1 \ldots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

- If next token is in Predict(P), then we should choose this production

Parse tables

- Step 2: build a parse table
  - Given some non-terminal V_n (the non-terminal we are currently processing) and a terminal V_t (the lookahead symbol), the parse table tells us which production P to use (or that we have an error
  - More formally:

    $T: V_n \times V_t \to P \cup \{\text{Error}\}$

Building the parse table

- Start: T[A][t] = //initialize all fields to "error"

  foreach A:

  foreach P with A on its lhs:

  foreach t in Predict(P):

  T[A][t] = P

- Exercise: build parse table for our toy grammar

1. $S \to A\ B\ \$$
2. $A \to x\ a\ A$
3. $A \to y\ a\ A$
4. $A \to \lambda$
5. $B \to b$

Recursive-descent parsers

- Given the parse table, we can create a program which generates recursive descent parsers
  - Remember the recursive descent parser we saw for MICRO
  - If the choice of production is not unique, the parse table tells us which one to take
- However, there is an easier method!

Stack-based parser for LL(1)

- Given the parse table, a stack-based algorithm is much simpler to generate than a recursive descent parser
- Basic algorithm:
  1. Push the RHS of a production onto the stack
  2. Pop a symbol, if it is a terminal, match it
  3. If it is a non-terminal, take its production according to the parse table and go to 1
- Algorithm on page 121
- Note: always start with start state

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3
y a A B $	y a b $	match(y)

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3
y a A B $	y a b $	match(y)
a A B $	a b $	match(a)

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3
y a A B $	y a b $	match(y)
a A B $	a b $	match(a)
A B $	b $	predict 4

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3
y a A B $	y a b $	match(y)
a A B $	a b $	match(a)
A B $	b $	predict 4
B $	b $	predict 5

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3
y a A B $	y a b $	match(y)
a A B $	a b $	match(a)
A B $	b $	predict 4
B $	b $	predict 5
b $	b $	match(b)

An example

1. S → A B $
2. A → x a A
3. A → y a A
4. A → λ
5. B → b

- How would a stack-based parser parse:

  x a y a b

Parse stack	Remaining input	Parser action
S	x a y a b $	predict 1
A B $	x a y a b $	predict 2
x a A B $	x a y a b $	match(x)
a A B $	a y a b $	match(a)
A B $	y a b $	predict 3
y a A B $	y a b $	match(y)
a A B $	a b $	match(a)
A B $	b $	predict 4
B $	b $	predict 5
b $	b $	match(b)
$	$	Done!

LL(k) parsers

- Can use similar techniques for LL(k) parsers
- Use more than one symbol of look-ahead to distinguish productions
- Why might this be bad?

Dealing with semantic actions

- Recall: we can annotate a grammar with *action symbols*
  - Tell the parser to invoke a semantic action routine
- Can simply push action symbols onto stack as well
- When popped, the semantic action routine is called

Non-LL(1) grammars

- Not all grammars are LL(1)!
- Consider

  <stmt> → if <expr> then <stmt list> endif

  <stmt> → if <expr> then <stmt list> else <stmt list> endif
- This is not LL(1) (why?)
- We can turn this in to

  <stmt> → if <expr> then <stmt list> <if suffix>

  <if suffix> → endif

  <if suffix> → else <stmt list> endif

Left recursion

- *Left recursion* is a problem for LL(1) parsers
  - LHS is also the first symbol of the RHS
- Consider:

  E → E + T
- What would happen with the stack-based algorithm?

Removing left recursion

E → E + T
E → T

→

E → E1 Etail
E1 → T
Etail → + T Etail
Etail → λ

Algorithm on page 125

Are all grammars LL(1)?

- No! Consider the if-then-else problem
- `if x then y else z`
- Problem: else is optional
- `if a then if b then c else d`
  - Which `if` does the `else` belong to?
- This is analogous to a "bracket language": $[^i\ ]^j\ (i \geq j)$

S → [S C
S → λ
C →]
C → λ

[[ ] can be parsed: SSλC or SSCλ
(it's ambiguous!)

Solving the if-then-else problem

- The ambiguity exists at the language level. To fix, we need to define the semantics properly
  - "] matches nearest unmatched ["
  - This is the rule C uses for if-then-else
  - What if we try this?

S → [S
S → S1
S1 → [S1]
S1 → λ

This grammar is still not LL(1) (or LL(k) for any k!)

Two possible fixes

- If there is an ambiguity, prioritize one production over another
  - e.g., if C is on the stack, always match "]" before matching "λ"

S → [S C
S → λ
C →]
C → λ

- Another option: change the language!
  - e.g., all if-statements need to be closed with an endif

S → if S E
S → other
E → else S endif
E → endif

Parsing if-then-else

- What if we don't want to change the language?
  - C does not require { } to delimit single-statement blocks
- To parse if-then-else, *we need to be able to look ahead at the entire rhs of a production* before deciding which production to use
  - In other words, we need to determine how many "]" to match before we start matching "["s
- *LR parsers* can do this!

LR Parsers

- Parser which does a Left-to-right, Right-most derivation
  - Rather than parse top-down, like LL parsers do, parse bottom-up, starting from leaves
- Basic idea: put tokens on a stack until an entire production is found
- Issues:
  - Recognizing the endpoint of a production
  - Finding the length of a production (RHS)
  - Finding the corresponding nonterminal (the LHS of the production)

Data structures

- At each state, given the next token,
  - A *goto table* defines the successor state
  - An *action table* defines whether to
    - *shift* – put the next state and token on the stack
    - *reduce* – an RHS is found; process the production
    - *terminate* – parsing is complete

Example

- Consider the simple grammar:

  <program> → begin <stmts> end $

  <stmts> → SimpleStmt ; <stmts>

  <stmts> → begin <stmts> end ; <stmts>

  <stmts> → λ

- Shift-reduce driver algorithm on page 142

Action and goto tables

	begin	end	;	SimpleStmt	$	<program>	<stmts>
0	S / 1						
1	S / 4	R4		S / 5			S / 2
2		S / 3					
3					A		
4	S / 4	R4		S / 5			S / 7
5			S / 6				
6	S / 4	R4		S / 5			S / 10
7		S / 8					
8			S / 9				
9	S / 4	R4		S / 6			S / 11
10		R2					
11		R3					

Example

- Parse: begin SimpleStmt ; SimpleStmt ; end $

Step	**Parse Stack**	**Remaining Input**	**Parser Action**
1	0	begin S ; S ; end $	Shift 1
2	0 1	S ; S ; end $	Shift 5
3	0 1 5	; S ; end $	Shift 6
4	0 1 5 6	S ; end $	Shift 5
5	0 1 5 6 5	; end $	Shift 6
6	0 1 5 6 5 6	end $	Reduce 4 (goto 10)
7	0 1 5 6 5 6 10	end $	Reduce 2 (goto 10)
8	0 1 5 6 10	end $	Reduce 2 (goto 2)
9	0 1 2	end $	Shift 3
10	0 1 2 3	$	Accept

Announcements

- I will be out of town on Tuesday (9/15)
- Class will be covered by Professor Midkiff

LR Parsers

- Basic idea:
  - **shift** tokens onto the stack. At any step, keep the set of productions that could generate the read-in tokens
  - **reduce** the RHS of recognized productions to the corresponding non-terminal on the LHS of the production. Replace the RHS tokens on the stack with the LHS non-terminal.

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead *k* symbols
  - Most powerful class of deterministic bottom-up parsers
  - LR(1) and variants are the most common parsers

Terminology for LR parsers

- Configuration: a production augmented with a "•"

  $A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$

- The "•" marks the point to which the production has been recognized. In this case, we have recognized $X_1 \ldots X_i$
- Configuration set: all the configurations that can apply at a given point during the parse:

  $A \rightarrow B \bullet CD$

  $A \rightarrow B \bullet GH$

  $T \rightarrow B \bullet Z$

- Idea: every configuration in a configuration set is a production that can possibly be matched

Configuration closure set

- Include all the configurations necessary to recognize the next symbol after the •

  closure0(configuration_set) defined on page 146

- Example:

S → E $
E → E + T | T
T → ID | (E)

closure0({S → • E $}) = {
S → • E $
E → • E + T
E → • T
T → • ID
T → • (E)
}

Successor configuration set

- Starting with the initial configuration set

  s0 = closure0({$S \rightarrow \bullet\ \alpha$ $})

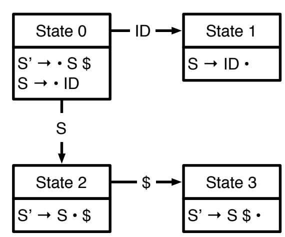
  an LR(0) parser will find the successor given the next symbol X

- X can be either a terminal (the next token from the scanner) or a non-terminal (the result of applying a reduction)
- Determining the successor s' = go_to0(s, X):
  - For each configuration in s of the form $A \rightarrow \beta \bullet X \gamma$ add $A \rightarrow \beta X \bullet \gamma$ to t
  - s' = closure0(t)

CFSM

- CFSM = Characteristic Finite State Machine
- Nodes are configuration sets (starting from s0)
- Arcs are go_to relationships

S' → S $
S → ID



Building the goto table

- We can just read this off from the CFSM

		Symbol		
		ID	$	S
State	0	1		2
	1			
	2		3	
	3			

Building the action table

- Given the configuration set s:
  - We shift if the next token matches a terminal after the • in some configuration

    $A \rightarrow \alpha \bullet a\ \beta \in s$ and $a \in V_t$, else error

  - We reduce production P if the • is at the end of a production

    $B \rightarrow \alpha \bullet \in s$ where production P is $B \rightarrow \alpha$

  - Extra actions:
    - shift if goto table transitions between states on a non-terminal
    - accept if we are about to shift $

Action table

		Symbol		
		ID	$	S
State	0	S		S
	1	R2	R2	R2
	2		A	
	3			

Alternate representation

- Some books represent goto and action tables differently
  - Action table only has columns for terminals, and consists of two kinds of actions:
    - shift + state: shift and move to a state
    - reduce + rule: reduce according to rule
  - Goto table only has columns for non-terminals
    - Specifies which state to go to after reducing

State	Action		Goto
	ID	$	S
0	S1		1
1	R2	R2	
2		A	
3			

Conflicts in action table

- For LR(0) grammars, the action table entries are unique: from each state, can only shift or reduce
- But other grammars may have conflicts
  - Reduce/reduce conflicts: multiple reductions possible from the given configuration
  - Shift/reduce conflicts: we can either shift or reduce from the given configuration

Shift/reduce example

- Consider the following grammar:

  $S \rightarrow A\ y$

  $A \rightarrow \lambda \mid x$

- This leads to the following initial configuration set:

  $S \rightarrow \bullet\, A\ y$

  $A \rightarrow \bullet\, x$

  $A \rightarrow \lambda\, \bullet$

- Can shift or reduce here

Lookahead

- Can resolve reduce/reduce conflicts and shift/reduce conflicts by employing *lookahead*
  - Looking ahead one (or more) tokens allows us to determine whether to shift or reduce
  - (*cf* how we resolved ambiguity in LL(1) parsers by looking ahead one token)
- Note that it is possible to create an LR(0) grammar for any LR(k) grammar (as long as we can determine the end of a program), but it may be very complex!

LR(1) parsing

- Configurations in LR(1) look similar to LR(0), but they are extended to include a lookahead symbol

  $A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j\ , l$ (where $l \in V_t \cup \lambda$)

- If two configurations differ only in their lookahead component, we combine them

  $A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j\ , \{l_1 \ldots l_m\}$

Building configuration sets

- To close a configuration

  B → α • A β, *l*

- Add all configurations of the form A → • γ, *u* where *u* ∈ First(β*l*)
- Intuition: the parse could apply the production for A, and the lookahead after we apply the production should match the next token that would be produced by B

Example

closure1({S → • E $, {λ}}) =

S → E $
E → E + T | T
T → ID | (E)

Example

closure1({S → • E $, {λ}}) =
S → • E $, {λ}

S → E $
E → E + T | T
T → ID | (E)

Example

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}

S → E $
E → E + T | T
T → ID | (E)

Example

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}

S → E $
E → E + T | T
T → ID | (E)

Example

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}
T → • ID, {$}

S → E $
E → E + T | T
T → ID | (E)

Example

S → E $
E → E + T | T
T → ID | (E)

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}
T → • ID, {$}
T → • (E), {$}

Example

S → E $
E → E + T | T
T → ID | (E)

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}
T → • ID, {$}
T → • (E), {$}
E → • E + T, {+}

Example

S → E $
E → E + T | T
T → ID | (E)

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}
T → • ID, {$}
T → • (E), {$}
E → • E + T, {+}
E → • T, {+}

Example

S → E $
E → E + T | T
T → ID | (E)

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}
T → • ID, {$}
T → • (E), {$}
E → • E + T, {+}
E → • T, {+}
T → • ID, {+}

Example

S → E $
E → E + T | T
T → ID | (E)

closure1({S → • E $, {λ}}) =
S → • E $, {λ}
E → • E + T, {$}
E → • T, {$}
T → • ID, {$}
T → • (E), {$}
E → • E + T, {+}
E → • T, {+}
T → • ID, {+}
T → • (E), {+}

Building goto and action tables

- The function goto1(configuration-set, symbol) is analogous to goto0(configuration-set, symbol) for LR(0)
  - Build goto table in the same way as for LR(0)
- Key difference: the action table.

  action[s][x] =

  - reduce when • is at end of configuration *and* x ∈ lookahead set of configuration

    A → α •, {... x ...} ∈ s

  - shift when • is before x

    A → β • x γ ∈ s

Problems with LR(1) parsers

- LR(1) parsers are very powerful ...
  - But the table size is much larger than LR(0) — as much as a factor of $|V_t|$ (why?)
  - Example: Algol 60 (a simple language) includes several thousand states!
- Storage efficient representations of tables are an important issue

Solutions to the size problem

- Different parser schemes
  - SLR (simple LR): build an CFSM for a language, then add lookahead wherever necessary (*i.e.*, add lookahead to resolve shift/reduce conflicts)
    - What should the lookahead symbol be?
    - To decide whether to reduce using production $A \rightarrow \alpha$, use Follow(A)
  - LALR: merge LR states in certain cases (we won't discuss this)

Semantic actions

- Recall: in LL parsers, we could integrate the semantic actions with the parser
  - Why? Because the parser was *predictive*
- Why doesn't that work for LR parsers?
  - Don't know which production is matched until parser reduces
- For LR parsers, we put semantic actions at the end of productions
  - May have to rewrite grammar to support all necessary semantic actions