Programming Languages & Translators

PARSING

Baishakhi Ray

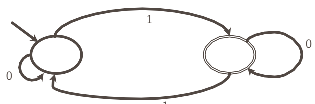
Fall 2018

These slides are motivated from Prof. Alex Aiken: Compilers (Stanford)

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular Languages
  - Weakest formal languages that are widely used
  - Many applications
- Many Languages are not regular

Automata that accepts odd numbers of 1



How many 1s it has accepted?

- Only solution is duplicate state

Automata does not have any memory

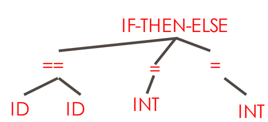
Intro to Parsing

- Regular Languages
  - Weakest formal languages that are widely used
  - Many applications
- Consider the language $\{(^i\,)^i \mid i \geq 0\}$
  - (), (( )), ((( )))
  - ((1 + 2) * 3)
- Nesting structures
  - if .. if.. else.. else..

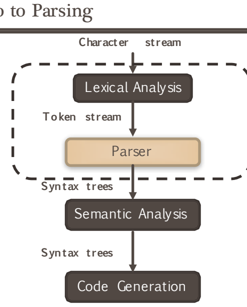
Regular languages cannot handle well

Intro to Parsing

- Input: if(x==y) 1 else 2;
- Parser Input (Lexical Input):
  KEY(IF) '(' ID(x) OP('==') ')' INT(1) KEY(ELSE) INT(2) ';'
- Parser Output



Intro to Parsing



- Nor every strings of tokens are valid
- Parser must distinguish between valid and invalid token strings.
- We need
  - A Language: to describe valid string
  - A method: to distinguish valid from invalid.

Context Free Grammar

- A CFG consists of
  - A set of terminal T
  - A set of non-terminal N
  - A start symbol S (S ϵ N)
  - A set of production rules
    - X -> $Y_1 \ldots Y_N$
    - X ϵ N
    - Y_i ϵ {N, T, ε}
- Ex: S -> (S) | ε
  - N = {S}
  - T = { (,) , ε}

Context Free Grammar

1. Begin with a string with only the start symbol S
2. Replace a non-terminal X with in the string by the RHS of some production rule: X->$Y_1 \ldots Y_n$
3. Repeat 2 again and again until there are no non-terminals

$X_1 \ldots X_i$ X X_{i+1} .... X_n -> $X_1 \ldots X_i$ $Y_1 \ldots Y_k$ X_{i+1} .... X_n
For the production rule X -> $Y_1 \ldots Y_k$

$$\propto_0 \rightarrow \propto_1 \rightarrow \ldots \rightarrow \propto_n$$

$$\propto_0 \xrightarrow{*} \propto_n, n \geq 0$$

Context Free Grammar

- Let G be a CFG with start symbol S. Then the language L(G) of G is:

$$\{a_1 \ldots \ldots an | \forall_i\, ai \in T \wedge S \xrightarrow{*} a_1 a_2 \ldots \ldots an\}$$

Context Free Grammar

- There are no rules to replace terminals.
- Once generated, terminals are permanent
- Terminals ought to be tokens of programming languages
- Context-free grammars are a natural notation for this recursive structure

CFG: Simple Arithmetic expression

E → E + E
  | E * E
  | (E)
  | id

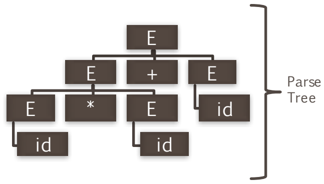
Languages can be generated: id, (id), (id + id) * id, …

Derivation

- A derivation is a sequence of production
  - S -> … -> … ->
- A derivation can be drawn as a tree
  - Start symbol is tree's root
  - For a production X -> $Y_1 \ldots Y_n$, add children $Y_1 \ldots Y_n$ to node X

- Grammar
  - E -> E + E | E * E | (E) | id
- String
  - id * id + id
- Derivation

E -> E + E
  -> E * E + E
  -> id * E + E
  -> id * id + E
  -> id * id + id



Parse Tree

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

Parse Tree

- Left-most derivation
  - At each step, replace the left-most non-terminal

E -> E + E
  -> E * E + E
  -> id * E + E
  -> id * id + E
  -> id * id + id

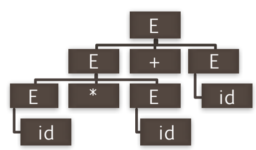
- Right-most derivation
  - At each step, replace the right-most non-terminal

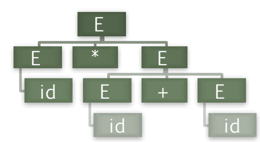
E -> E + E
  -> E + id
  -> E * E + id
  -> E * id + id
  -> id * id + id

Note that, right-most and left-most derivations have the same parse tree

Ambiguity

- Grammar
  - E -> E + E | E * E | (E) | id
- String
  - id * id + id





Ambiguity

- A grammar is ambiguous if it has more than one parse tree for a string
  - There are more than one right-most or left-most derivation for some string
- Ambiguity is bad
  - Leaves meaning for some programs ill-defined

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g., in C)

Error Kind	Example	Detected by
Lexical	... $ ...	Lexer
Syntax	... x*%...	Parser
Semantic	... int x; y = x(3);...	Type Checker
Correctness	your program	tester/user

Error Handling

- Error Handler should
  - Recover errors accurately and quickly
  - Recover from an error quickly
  - Not slow down compilation of valid code
- Types of Error Handling
  - Panic mode
  - Error productions
  - Automatic local or global correction

Panic Mode Error Handling

- Panic mode is simplest and most popular method
- When an error is detected
  - Discard tokens until one with a clear role is found
  - Continue from there
- Typically looks for "synchronizing" tokens
  - Typically the statement of expression terminators

Panic Mode Error Handling

- Example:
  - (1 + + 2) + 3
- Panic-mode recovery:
  - Skip ahead to the next integer and then continue
- Bison: use the special terminal error to describe how much input to skip
  - $E \rightarrow \underbrace{\text{int} \mid E + E \mid (E)}_{\text{Normal mode}} \mid \underbrace{\text{error int} \mid (\text{ error })}_{\text{Error mode}}$

Error Productions

- Specify known common mistakes in the grammar
- Example:
  - Write 5x instead of 5 * x
  - Add production rule E -> .. | E E
- Disadvantages
  - complicates the grammar

Error Corrections

- Idea: find a correct "nearby" program
  - Try token insertions and deletions (goal: minimize edit distance)
  - Exhaustive search
- Disadvantages
  - Hard to implement
  - Slows down parsing of correct programs
  - "Nearby" is not necessarily "the intended" program

Error Corrections

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in once cycle as possible
- Disadvantages
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling

Abstract Syntax Trees

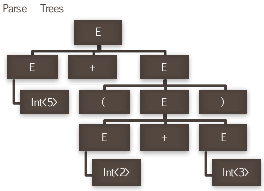
- A parser traces the derivation of a sequence of tokens
- But the rest of the compiler needs a structural representation of the program
- Abstract Syntax Trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

Abstract Syntax Trees

- Grammar
  - E -> int | (E) | E + E
- String
  - 5 + (2 + 3)
- After lexical analysis
  - Int<5> '+' '(' Int<2> '+' Int<3> ')'

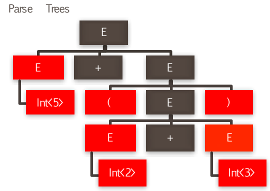
Abstract Syntax Trees: 5 + (2 + 3)

Parse Trees



Abstract Syntax Trees: 5 + (2 + 3)

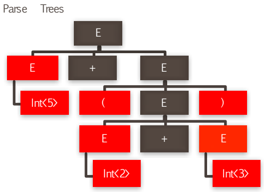
Parse Trees



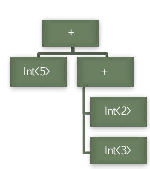
- Have too much information
  - Parentheses
  - Single-successor nodes

Abstract Syntax Trees: 5 + (2 + 3)

Parse Trees



AST



- Have too much information
  - Parentheses
  - Single-successor nodes

- ASTs capture the nesting structure
- But abstracts from the concrete syntax
  - More compact and easier to use

Disadvantages of ASTs

- AST has many similar forms
  - E.g., for, while, repeat...until
  - E.g., if, ?:, switch
- Expressions in AST may be complex, nested
  - (x * y) + (z > 5 ? 12 * z : z + 20)
- Want simpler representation for analysis
  - ...at least, for dataflow analysis

30

Parsing algorithm: Recursive Descent Parsing

- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream

Parsing algorithm: Recursive Descent Parsing

- Grammar:
  - E -> T | T + E
  - T -> int | int * T | (E)
- Token Stream: (int<5>)
- Start with top level non-terminal E
  - Try the rules for E in order

Recursive Descent Parsing Example

E -> **T** | T + E

T -> **int** | int * T | (E)

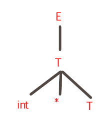
mismatch: int does not match arrowhead (
backtrack

(int<5>)

Recursive Descent Parsing Example

E -> **T** | T + E

T -> int | **int * T** | (E)



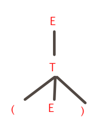
backtrack

(int<5>)

Recursive Descent Parsing Example

E -> **T** | T + E

T -> int | int * T | **(E)**



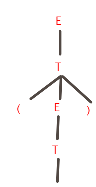
Match! Advance input

(int<5>)

Recursive Descent Parsing Example

E -> **T** | T + E

T -> **int** | int * T | (E)



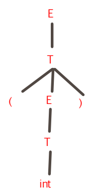
Match! Advance input

(int<5>)

Recursive Descent Parsing Example

E -> T | T + E

T -> int | int * T | (E)



Match! Advance input

(int<5>)

A Recursive Descent Parser. Preliminaries

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES .
- Let the global **next** point to the next token

A (Limited) Recursive Descent Parser

- Define boolean functions that check the token string for a match of
  - A given token terminal

    ```
    bool term (TOKEN tok) { return *next++ == tok; }
    ```
  - The n^{th} production of S:

    ```
    bool Sn() { … }
    ```
  - Try all productions of S:

    ```
    bool S() { … }
    ```

A (Limited) Recursive Descent Parser

- For production E → T

  ```
  bool E1() { return T(); }
  ```
- For production E → T + E

  ```
  bool E2() { return T() && term(PLUS) && E(); }
  ```
- For all productions of E (with backtracking)

  ```
  bool E() {
    TOKEN *save = next;
    return (next = save, E1( )) || (next = save, E2( ));
  }
  ```

A (Limited) Recursive Descent Parser (4)

- Functions for non-terminal T

```
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() {
    TOKEN *save = next;
    return (next = save, T1())
        || (next = save, T2())
        || (next = save, T3());
}
```

Recursive Descent Parsing

- To start the parser
  - Initialize next to point to first token
  - Invoke E() . Notice how this simulates the example parse .

Example

Grammar:

E → T |T + E
T → int | int * T | (E)

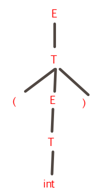
Input: (int)

Code:

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {TOKEN *save = next;
          return (next = save, E1()) || (next = save, E2()); }

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next;
     return (next = save, T1())
          || (next = save, T2())
          || (next = save, T3()); }
```



When Recursive Descent Does Not Work

Grammar:

E → T |T + E
T → int | int * T | (E)

Input: int * int

Code:

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {TOKEN *save = next;
          return (next = save, E1()) || (next = save, E2()); }

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next;
     return (next = save, T1())
          || (next = save, T2())
          || (next = save, T3()); }
```

Recursive Descent Parsing: Limitation

- If production for non-terminal X succeeds
  - Cannot backtrack to try different production for X later
- General recursive descent algorithms support such full backtracking
  - Can implement any grammar
- Presented RDA is not general
  - But easy to implement
- Sufficient for grammars where for any non-terminal at most one production can succeed
- The grammar can be rewritten to work with the presented algorithm
  - By left factoring

Left Factoring

A -> $\alpha\beta_1$ | $\alpha\beta_2$

- The input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.
- We can defer the decision by expanding A to αA'.
- Then, after seeing the input derived from α, we expand A' to β_1 or β_2 (left-factored)
- The original productions become:

  A -> $\alpha A'$, A'-> β_1 | β_2

When Recursive Descent Does Not Work

- Consider a production S → S a
  bool S1() { return S() && term(a); }
  bool S() { return S1(); }
- S() goes into an infinite loop
- A left-recursive grammar has a non-terminal S

  S $\rightarrow^{+}$ Sα for some α
- Recursive descent does not work for left recursive grammar

Elimination of Left Recursion

- Consider the left-recursive grammar

  S → S α | β
- S generates all strings starting with a β and followed by a number of α
- Can rewrite using right-recursion

  S → β S'

  S' → α S' | ε

More Elimination of Left-Recursion

- In general
  $S \to S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as
  $S \to \beta_1\ S' \mid \ldots \mid \beta_m\ S'$
  $S' \to \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$

General Left Recursion

- The grammar
  $S \to A\ \alpha \mid \delta$
  $A \to S\ \beta$
  is also left-recursive because
  $S \to^+ S\ \beta\ \alpha$
- This left-recursion can also be eliminated

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
  - In practice, LL(1) is used

LL(1) vs. Recursive Descent

- In recursive-descent
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In LL(1)
  - At each step, only one choice of production
  - That is
    - When a non-terminal A is leftmost in a derivation
    - The next input symbol is t
    - There is a unique production $A \to \alpha$ to use
      - Or no production to use (an error state)
- LL(1) is a recursive descent variant without backtracking

Predictive Parsing and Left Factoring

- Recall the grammar
  $E \to T + E \mid T$
  $T \to \text{int} \mid \text{int} * T \mid (E)$
- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- We need to left-factor the grammar

Left-Factoring Example

- Grammar
  E → T + E | T
  T → int | int * T | (E)
- Factor out common prefixes of productions
  E → T X
  X → + E | ε
  T → (E) | int Y
  Y → * T | ε

LL(1) Parsing Table Example

- Left-factored grammar
  E → T X
  X → + E | ε
  T → (E) | int Y
  Y → * T | ε
- The LL(1) parsing table:

Left-most non-terminals	next input tokens					
	int	*	+	(	)	$
E	TX			TX		
X			+E		ε	ε
T	int Y			(E)		
Y		*T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production E → T X"
  - This can generate an int in the first position
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only if Y → ε

LL(1) Parsing Tables. Errors

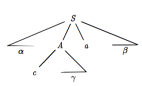
- Blank entries indicate error situations
- Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

Using Parsing Tables

- Method similar to recursive descent, except
  - For the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]
- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to match against the input
  - Top of stack = leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input & empty stack

First & Follow

- During top down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- FIRST(α), α is any string of grammar symbols
  - A set of **terminals** that begin strings derived from α.
  - If $\alpha \xrightarrow{*} \epsilon$, then ϵ is in FIRST(α).
  - if $\alpha \xrightarrow{*} cY$, the c is in FIRST($\alpha$).



- FOLLOW(A), A is a **nonterminal**
  - the set of **terminals** that can appear immediately to the right of A
  - A set of terminals "a" such that S $\xrightarrow{*} \alpha A a \beta$ for some α and β.

Constructing Parsing Tables: The Intuition

- Consider non-terminal A, production $A \to \alpha$, & token t
- T[A,t] = α in two cases:
- If $\alpha \to^* t\ \beta$
  - α can derive a t in the first position
  - We say that $t \in \mathrm{First}(\alpha)$
- If $A \to \alpha$ and $\alpha \to^* \varepsilon$ and $S \to^* \beta\ A\ t\ \delta$
  - Useful if stack has A, input is t, and A cannot derive t
  - In this case only option is to get rid of A (by deriving ε)
  - We say $t \in \mathrm{Follow}(A)$

Computing First Sets

- Definition

  $\mathrm{First}(X) = \{ t \mid X \to^* t\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$, X can be single terminal, single non-terminal, or string including both

- Algorithm sketch:

1. $\mathrm{First}(t) = \{ t \}$, t is terminal
2. $\varepsilon \in \mathrm{First}(X)$
   - if $X \to \varepsilon$
   - if $X \to A_1 \dots A_n$ and $\varepsilon \in \mathrm{First}(A_i)$ for $1 \le i \le n$
3. $\mathrm{First}(\alpha) \subseteq \mathrm{First}(X)$ if $X \to A_1 \dots A_n\ \alpha$
   - $\varepsilon \in \mathrm{First}(A_i)$ for $1 \le i \le n$

First Sets. Example

- grammar

  E → T X
  X → + E | ε
  T → (E) | int Y
  Y → * T | ε

- First sets

  First(( ) = { (}
  First( )) = {) }
  First(int) = { int }
  First(+) = { + }
  First(*) = { * }

  First(E) ⊇ = First(T) = {int, (}
  First(X) = {+, ε }
  First(Y) = {*, ε }

Computing Follow Sets

- Definition:

  $\mathrm{Follow}(X) = \{ t \mid S \to^* \beta\ X\ t\ \delta \}$

- Intuition:
  - If $X \to A\ B$ then $\mathrm{First}(B) \subseteq \mathrm{Follow}(A)$ and $\mathrm{Follow}(X) \subseteq \mathrm{Follow}(B)$
  - If $B \to^* \varepsilon$ then $\mathrm{Follow}(X) \subseteq \mathrm{Follow}(A)$
  - If S is the start symbol then $\$ \in \mathrm{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch:

1. $\$ \in \mathrm{Follow}(S)$
2. $\mathrm{First}(\beta) - \{\varepsilon\} \subseteq \mathrm{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$
3. $\mathrm{Follow}(A) \subseteq \mathrm{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$ where $\varepsilon \in \mathrm{First}(\beta)$

Follow Sets. Example

- Recall the grammar

  E → T X   X → + E | ε
  T → (E) | int Y   Y → * T | ε

- Follow sets

```
Follow( + ) = { int, ( }
Follow( ( ) = { int, ( }          Follow( E ) = {), $}
Follow( * ) = { int, ( }          Follow( T ) = {+, ) , $}
Follow( ) ) = {+, ) , $}          Follow( Y ) = {+, ) , $}
Follow( int) = {*, +, ) , $}.     Follow( X ) = {$, ) }
```

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production A → α in G do:
  - For each terminal t ∈ First(α) do
    - T[A, t] = α
  - If ε ∈ First(α), for each t ∈ Follow(A) do
    - T[A, t] = α
  - If ε ∈ First(α) and $ ∈ Follow(A) do
    - T[A, $] = α

LL(1) Parsing Table Example

- Left-factored grammar
  - E → T X
  - X → + E | ε
  - T → (E) | int Y
  - Y → * T | ε
- The LL(1) parsing table:

Rules:

For each production A → α in G do:
For each terminal t ∈ First(α) do
T[A, t] = α
If ε ∈ First(α), for each t ∈ Follow(A) do
T[A, t] = α
If ε ∈ First(α) and $ ∈ Follow(A) do
T[A, $] = α

		next input tokens					
		int	*	+	(	)	$
Left-most non-terminals	E	TX			TX		
	X			+E		ε	ε
	T	int Y			(E)		
	Y		*T	ε		ε	ε

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1) [Eg: S->Sa|b]
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - other: e.g., LL(2)
- Most programming language CFGs are not LL(1)
  - too weak
  - However they build on these basic ideas

Bottom-Up Parsing

- Bottom-up parsing is more general than (deterministic) top-down parsing
  - just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up parsers don't need left-factored grammars
- Revert to the "natural" grammar for our example:
  - E → T + E | T
  - T → int * T | int | (E) ·
- Consider the string: int * int + int

Bottom-Up Parsing

- Revert to the "natural" grammar for our example:
  - E → T + E | T
  - T → int * T | int | (E) ·
- Consider the string: int * int + int
- Bottom-up parsing reduces a string to the start symbol by inverting productions:

```
int * int + int      T → int
int * T + int        T → int * T
T + int              T → int
T + T                E → T
T + E                E → T + E
E
```

Observation

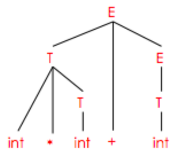
- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

```
int * int + int      T → int
int * T + int        T → int * T
T + int              T → int
T + T                E → T
T + E                E → T + E
E
```

Bottom-Up Parsing

- A bottom-up parser traces a rightmost derivation in reverse

```
int * int + int        T → int
int * T  + int         T → int * T
T + int                T → int
T + T                  E → T
T + E                  E → T + E
E
```

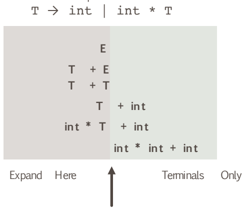


A trivial Bottom-Up Parsing Algorithm

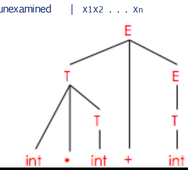
```
Let I = input string
  repeat
    pick a non-empty substring β of I
          where X→ β is a production
    if no such β, backtrack
    replace one β by X in I
  until I = "S" (the start symbol) or all possibilities are exhausted
```

Bottom-Up Parsing

```
E → T | T + E
T → int | int * T
```



- Split string into two substrings
  - Right substring is not examined yet by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a |
  - The | is not part of the string
- Initially, all input is unexamined | $x_1x_2 \ldots x_n$



Where Do Reductions Happen?

- Right-most derivation has an interesting consequence:
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \to \beta$
  - Then ω is a string of terminals
- Why? Because $\alpha X\omega \to \alpha\beta\omega$ is a step in a rightmost derivation

Shift-Reduce Parsing

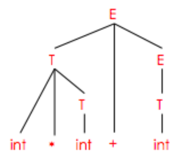
- Bottom-up parsing uses only two kinds of actions:
  - Shift
  - Reduce
- Shift: Move | one place to the right
  - Shifts a terminal to the left string ABC|xyz ⇒ ABCx|yz
- Reduce: Apply an inverse production at the right end of the left string
  - If A → xy is a production, then Cbxy|ijk ⇒ CbA|ijk

The Example with Reductions Only

```
int * int | + int      reduce T → int
int * T | + int        reduce T → int * T
T + int |              reduce T → int
T + T |                reduce E → T
T + E |                reduce E → T + E
E |
```

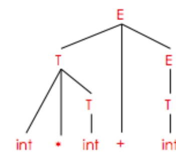
The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T | + int	shift
T + | int	shift
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	



An Example with Shift-Reduce Parsing

stack	input	
	int * int + int	shift
int	* int + int	shift
int *	int + int	shift
int * int	+ int	reduce
int * T	+ int	reduce
T	+ int	shift
T +	int	shift
T + int		reduce
T + T		reduce
T + E		reduce
E		



The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce
  - pops 0 or more symbols off of the stack (production rhs)
  - pushes a nonterminal on the stack (production lhs)

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse
- If it is legal to shift or reduce, there is a shift-reduce conflict
- If it is legal to reduce by two different productions, there is a reduce-reduce conflict.

Key Issue

- How do we decide when to shift or reduce?
- Example grammar:
  E → T + E | T
  T → int * T | int | (E)
- Consider step int | * int + int
  - We could reduce by T → int giving T | * int + int
  - A fatal mistake!
    - No way to reduce to the start symbol E

Handles

- Intuition: Want to reduce only if the result can still be reduced to the start symbol.
- Assume a rightmost derivation
  $S \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\omega$
- Then $X \rightarrow \beta$ in the position after α is a handle of $\alpha\beta\omega$

Handles

- A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot).
- We only want to reduce at handles
- In shift-reduce parsing, handles appear only at the top of the stack, never inside
- Informal induction on # of reduce moves:
- True initially, stack is empty
- Immediately after reducing a handle
  - right-most non-terminal on top of the stack
  - next handle must be to right of right-most nonterminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

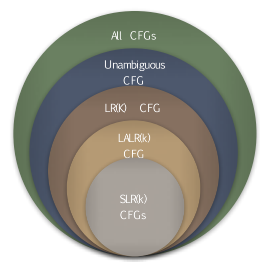
Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack
- Handles are never to the left of the rightmost non-terminal
  - Therefore, shift-reduce moves are sufficient; the | need never move left
- Bottom-up parsing algorithms are based on recognizing handles

Recognizing Handles

- There are no known efficient algorithms to recognize handles
- Solution: use heuristics to guess which stacks are handles
- On some CFGs, the heuristics always guess correctly
  - For the heuristics we use here, these are the SLR grammars
  - Other heuristics work for other grammars

Grammars



Viable Prefixes

- α is a viable prefix if there is an ω such that α|ω is a state of a shift-reduce parser
  - α is stack
  - ω is rest of the inputs
- A viable prefix does not extend past the right end of the handle
- It's a viable prefix because it is a prefix of the handle
- As long as a parser has viable prefixes on the stack **no** parsing error has been detected
- For any grammar, the set of variable prefixes is a regular language
  - we can compute an automata that accepts variable prefixes

Viable Prefixes

```
E → T | T + E
T → int | int * T
```

Items

- An item is a production with a "." somewhere on the rhs
- The items for T → (E) are
  - T → .(E)
  - T → (.E)
  - T → (E.)
  - T → (E).
- The only item for X → ε is X → .
- Items are often called "LR(0) items"

Intuition

- The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions
- If it had a complete rhs, we could reduce
- These bits and pieces are always prefixes of rhs of productions

Example

- Consider the input (int)
  - Then (E|) is a state of a shift-reduce parse
  - (E is a prefix of the rhs of T → (E) ·
    - Will be reduced after the next shift
  - Item T → (E.) says that so far we have seen (E of this production and hope to see)

Generalization

- The stack may have many prefixes of rhs's
  - $Prefix_1$ $Prefix_2$ . . . $Prefix_{n-1}$ $Prefix_n$
- Let $Prefix_i$ be a prefix of rhs of $X_i \rightarrow \alpha_i$
  - $Prefix_i$ will eventually reduce to X_i
  - The missing part of α_{i-1} starts with X_i
  - i.e. there is a $X_{i-1} \rightarrow Prefix_{i-1}\ X_i\ \beta$ for some β
- Recursively, $Prefix_{k+1} \ldots Prefix_n$ eventually reduces to the missing part of α_k

An Example

- Consider the string (int * int):
  - (int *|int) is a state of a shift-reduce parse
  - "(" is a prefix of the rhs of T → (E)
  - "ε" is a prefix of the rhs of E → T
  - "int *" is a prefix of the rhs of T → int * T
- The "stack of items"
  - T → (.E)
  - E → .T
  - T → int * .T
- Says
  - We've seen "(" of T → (E)
  - We've seen ε of E → T
  - We've seen int * of T → int * T

Recognizing Viable Prefixes

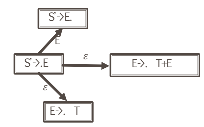
- Idea: To recognize viable prefixes, we must
  - Recognize a sequence of partial rhs's of productions, where
  - Each sequence can eventually reduce to part of the missing suffix of its predecessor

An NFA Recognizing Viable Prefixes

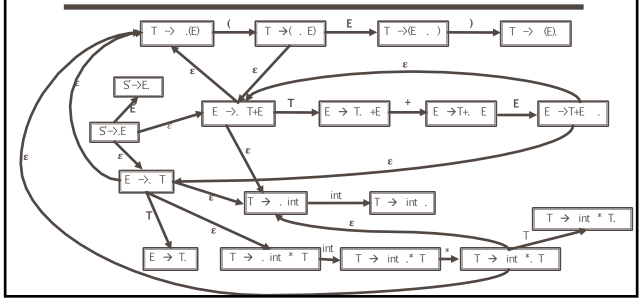
1. Add a dummy production S' → S to G
2. The NFA states are the items of G
   - Including the extra production
   - NFA(stack) → accept|reject
3. For item E → α.Xβ add transition
   E → α.Xβ →^{X} E → αX.β
4. For item E → α.Xβ and production X → γ add
   E → α.Xβ →^{ε} X → .γ
5. Every state is an accepting state
6. Start state is S' → .S

Recognizing VP

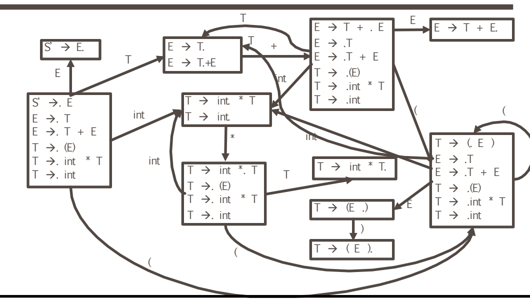
S'->E
E ->T+E|T
T->int*T|int| (E)



NFA of Viable Prefixes



DFA of Viable Prefixes



DFA of Viable Prefixes

- The states of the DFA are
  "canonical collections of items"
  or
  "canonical collections of LR(0) items"

Valid Items

- Item X → β.γ is valid for a viable prefix αβ if
  S' →* αXω → αβγω by a right-most derivation
- After parsing αβ, the valid items are the possible tops of the stack of items
- An item I is valid for a viable prefix α if the DFA recognizing viable prefixes terminates on input α in a state s containing I
- The items in s describe what the top of the item stack might be after reading input α
- An item is often valid for many prefixes
  - Example: The item T → (.E) is valid for prefixes ((((((((((

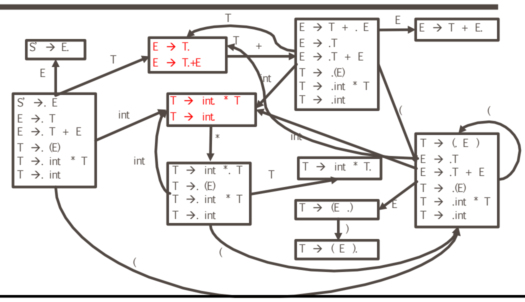
LR(o) Parsing

- Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \rightarrow \beta$ if
  - s contains item $X \rightarrow \beta.$
- Shift if
  - s contains item $X \rightarrow \beta.t\omega$
  - equivalent to saying s has a transition labeled t

LR(o) Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.$
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.t\delta$

LR(o) Conflicts: Two shift-reduce conflicts



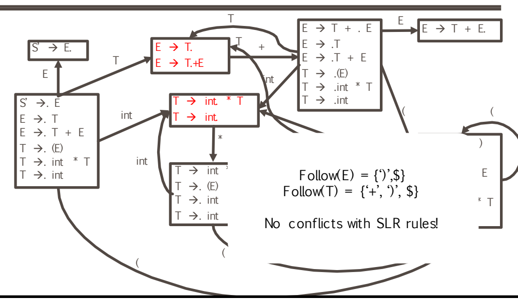
SLR

- LR = "Left-to-right scan"
- SLR = "Simple LR"
- SLR improves on LR(0) shift/reduce heuristics
  - Fewer states have conflicts

SLR Parsing

- Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \rightarrow \beta$ if
  - s contains item $X \rightarrow \beta.$
  - $t \in$ Follow(X)
- Shift if
  - s contains item $X \rightarrow \beta.t\omega$
- If there are conflicts under these rules, the grammar is not SLR
- The rules amount to a heuristic for detecting handles
  - The SLR grammars are those where the heuristics detect exactly the handles

SLR Conflicts



Follow(E) = {')',$}
Follow(T) = {'+', ')', $}

No conflicts with SLR rules!

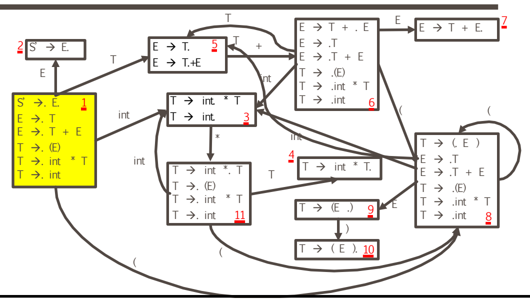
Precedence Declarations

- Lots of grammars aren't SLR
  - including all ambiguous grammars
- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts
- Consider our favorite ambiguous grammar: – E → E + E | E * E | (E) | int · The DFA for this grammar contains a state with the following items: – E → E * E . E → E . + E – shift/reduce conflict! · Declaring "* has higher precedence than +" resolves this conflict in favor of reducing

Naïve SLR Parsing Algorithm

1. Let M be DFA for viable prefixes of G
2. Let |$x_1 \ldots x_n$\$ be initial configuration
3. Repeat until configuration is S|\$
   - Let α|ω be current configuration
   - Run M on current stack α
   - If M rejects α, report parsing error
     - Stack α is not a viable prefix
   - If M accepts α with items I, let a be next input
     - Shift if X → β. a γ ∈ I
     - Reduce if X → β. ∈ I and a ∈ Follow(X)
     - Report parsing error if neither applies

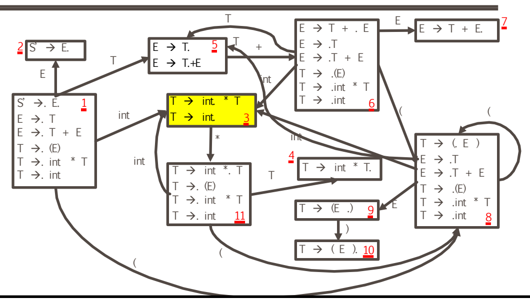
Parsing |int * int $



Parsing |int * int $

<table>
<tr><th>Configuration</th><th>DFA Halt State</th><th>Action</th></tr>
<tr><td>|int * int$</td><td>1</td><td>shift</td></tr>
<tr><td>int | * int$</td><td></td><td></td></tr>
</table>

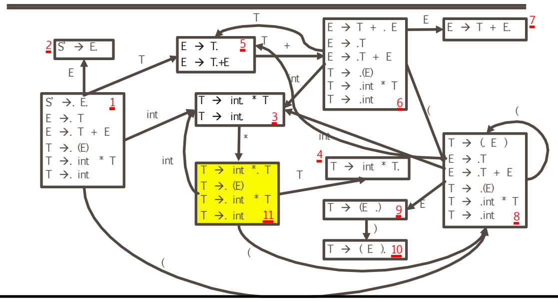
Parsing int| * int $



Parsing |int * int $

<table>
<tr><th>Configuration</th><th>DFA Halt State</th><th>Action</th></tr>
<tr><td>|int * int$</td><td>1</td><td>shift</td></tr>
<tr><td>int | * int$</td><td>3 (* not in Follow(T))</td><td>shift</td></tr>
<tr><td>int * | int$</td><td></td><td></td></tr>
</table>

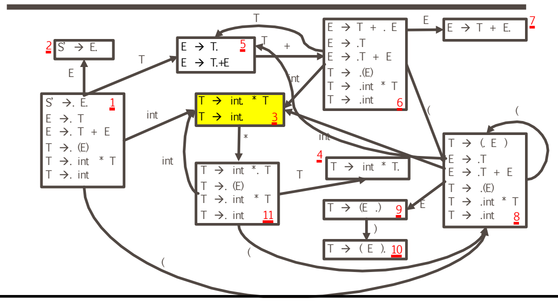
Parsing int * |int $



Parsing |int * int $

<table>
<tr><th>Configuration</th><th>DFA Halt State</th><th>Action</th></tr>
<tr><td>|int * int$</td><td>1</td><td>shift</td></tr>
<tr><td>int | * int$</td><td>3 (* not in Follow(T))</td><td>shift</td></tr>
<tr><td>int * | int$</td><td>11</td><td>shift</td></tr>
<tr><td>int * int | $</td><td></td><td></td></tr>
</table>

Parsing int| * int $



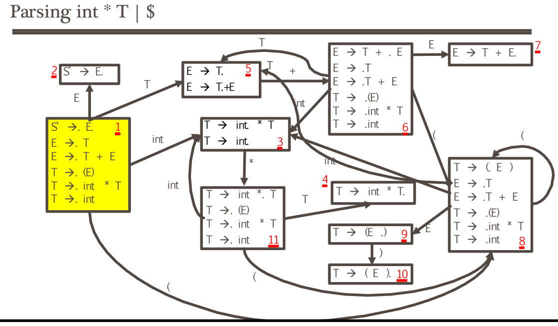
Parsing |int * int $

<table>
<tr><th>Configuration</th><th>DFA Halt State</th><th>Action</th></tr>
<tr><td>|int * int$</td><td>1</td><td>shift</td></tr>
<tr><td>int | * int$</td><td>3 (* not in Follow(T))</td><td>shift</td></tr>
<tr><td>int * | int$</td><td>11</td><td>shift</td></tr>
<tr><td>int * int | $</td><td>3 ($ ϵ Follow (T))</td><td>reduce T->int</td></tr>
</table>

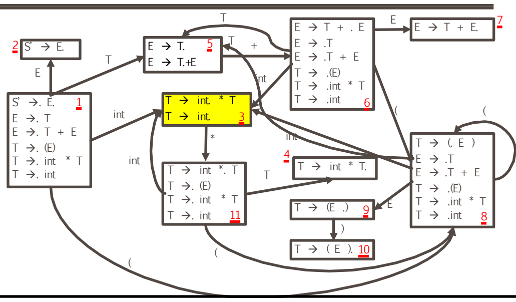
Parsing |int * int $

<table>
<tr><th>Configuration</th><th>DFA Halt State</th><th>Action</th></tr>
<tr><td>|int * int$</td><td>1</td><td>shift</td></tr>
<tr><td>int | * int$</td><td>3 (* not in Follow(T))</td><td>shift</td></tr>
<tr><td>int * | int$</td><td>11</td><td>shift</td></tr>
<tr><td>int * int | $</td><td>3 ($ ϵ Follow (T))</td><td>reduce T->int</td></tr>
<tr><td>int * T | $</td><td></td><td></td></tr>
</table>

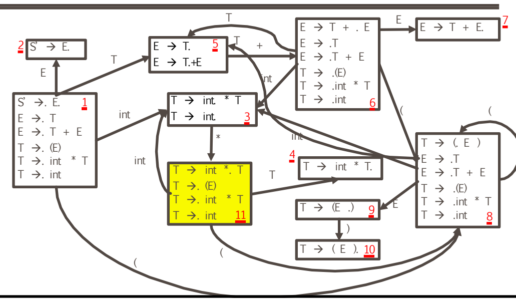
Parsing int * T | $



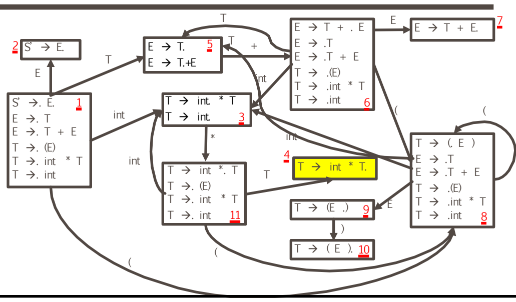
Parsing int * T | $



Parsing int * T | $



Parsing int * T | $



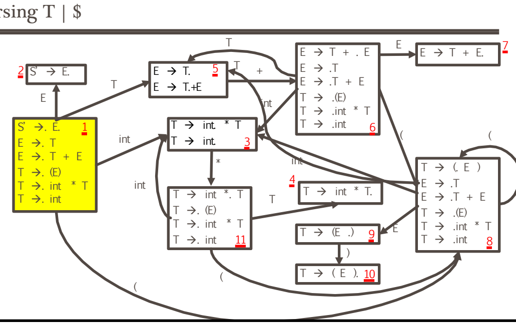
Parsing |int * int $

Configuration	DFA Halt State	Action
|int * int$	1	shift
int | * int$	3 (* not in Follow(T))	shift
int * | int$	11	shift
int * int | $	3 ($ ϵ Follow (T))	reduce T->int
int * T | $	4 ($ ϵ Follow (T))	reduce T-> int * T

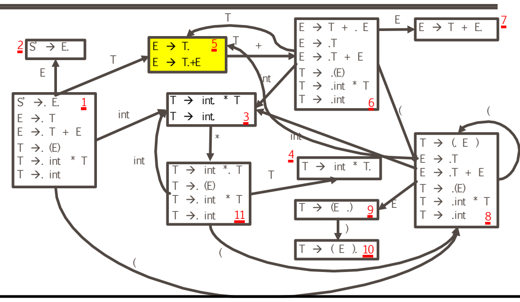
Parsing |int * int $

Configuration	DFA Halt State	Action
|int * int$	1	shift
int | * int$	3 (* not in Follow(T))	shift
int * | int$	11	shift
int * int | $	3 ($ ϵ Follow (T))	reduce T->int
int * T | $	4 ($ ϵ Follow (T))	reduce T-> int * T
T | $		

Parsing T | $



Parsing T | $



Parsing |int * int $

Configuration	DFA Halt State	Action
|int * int$	1	shift
int | * int$	3 (* not in Follow(T))	shift
int * | int$	11	shift
int * int | $	3 ($ ϵ Follow (T))	reduce T->int
int * T | $	4 ($ ϵ Follow (T))	reduce T-> int * T
T | $	5 ($ ϵ Follow (E))	reduce E-> T

Parsing |int * int $

Configuration	DFA Halt State	Action
|int * int$	1	shift
int | * int$	3 (* not in Follow(T))	shift
int * | int$	11	shift
int * int | $	3 ($ ϵ Follow (T))	reduce T->int
int * T | $	4 ($ ϵ Follow (T))	reduce T-> int * T
T | $	5 ($ ϵ Follow (E))	reduce E-> T
E | $		accept

An Improvement

- Rerunning the automaton at each step is wasteful
  - Most of the work is repeated
- Change stack to contain pairs 〈 Symbol, DFA State 〉
  - DFA State is the state of the automaton on each prefix of the stack
- For a stack 〈 sym_1, $state_1$ 〉 . . . 〈 sym_n, $state_n$ 〉
  - $state_n$ is the final state of the DFA on sym_1 … sym_n
- The bottom of the stack is 〈any, start〉 where
  - any is any dummy symbol
  - start is the start state of the DFA

Goto Table

- Define goto[i,A] = j if $state_i \to^A state_j$
- goto is the transition function of the DFA

Refined Parser Moves

- Shift x
  - Push 〈a, x〉 on the stack
  - a is current input
  - x is a DFA state
- Reduce X → α
  - As before
- Accept
- Error

Action Table

- For each state s_i and terminal a
  - If s_i has item X → α.aβ and goto[i,a] = j then action[i,a] = shift j
  - If s_i has item X → α. and a ∈ Follow(X) and X ≠ S' then action[i,a] = reduce X → α
  - If s_i has item S' → S. then action[i,$] = accept
  - Otherwise, action[i,a] = error

SLR Parsing Algorithm

```
Let I = w$ be initial input
Let j = 0
Let DFA state 1 have item S' → .S
Let stack = 〈 dummy, 1 〉
    repeat
        case action[top_state(stack),I[j]] of
            shift k: push 〈 I[j++], k 〉
            reduce X → A:
                pop |A| pairs,
                push 〈X, goto[top_state(stack),X]〉
            accept: halt normally
            error: halt and report error
```

Notes on SLR Parsing Algorithm

- Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used! ·
  - However, we still need the symbols for semantic actions

L, R, and all that

- LR parser: "Bottom-up parser"
- L = Left-to-right scan, R = Rightmost derivation
- RR parser: R = Right-to-left scan (from end)
  - nobody uses these
- LL parser: "Top-down parser":
- L = Left-to-right scan: L = Leftmost derivation
- LR(1): LR parser that considers next token (lookahead of 1)
- LR(0): Only considers stack to decide shift/reduce
- SLR(1): Simple LR: lookahead from first/follow rules Derived from LR(0) automaton
- LALR(1): Lookahead LR(1): fancier lookahead analysis Uses same LR(0) automaton as SLR(1)