

PARSING

Baishakhi Ray

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

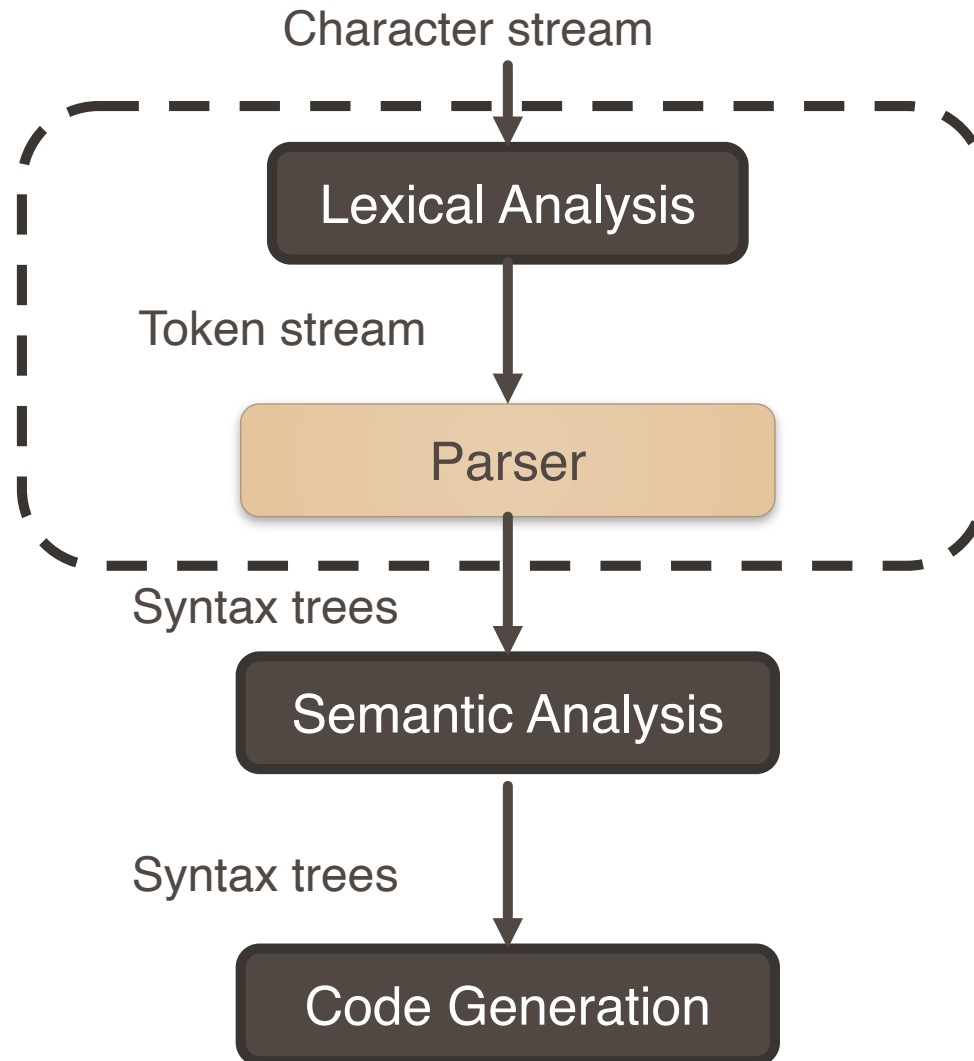


- `<id, x> <op, *> <op, %>`

- Is it a valid token stream in C language?

- Is it a valid statement in C language?

Intro to Parsing



- Not every strings of tokens are valid
- Parser must distinguish between valid and invalid token strings.
- We need
 - A Language: to describe what is valid string?
 - A method: to determine membership of inputs in this language.

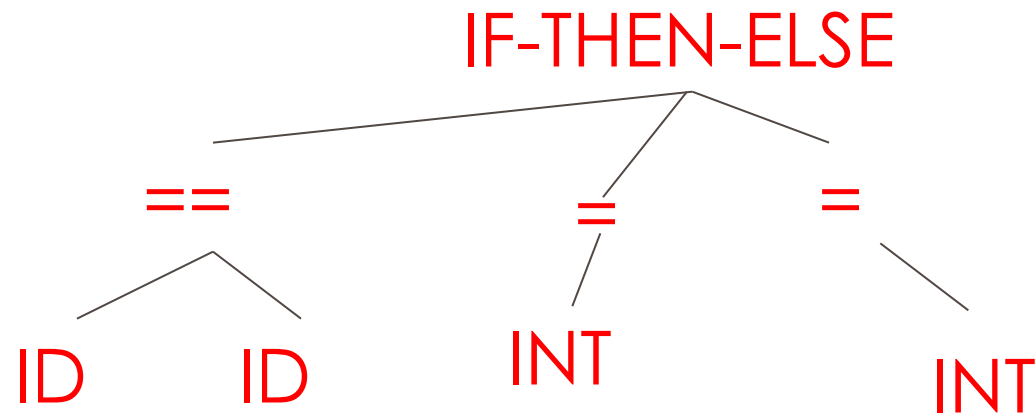
Intro to Parsing

- Input: if(x==y) 1 else 2;

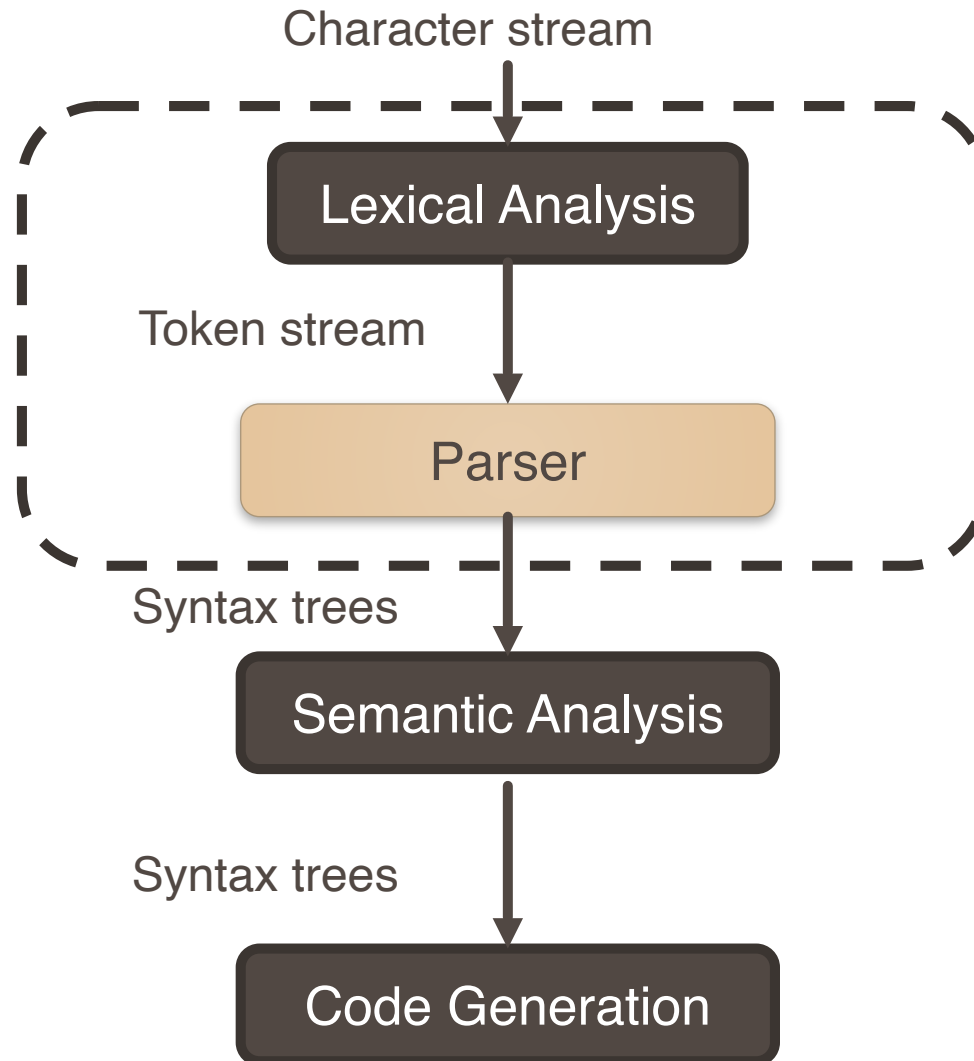
- Parser Input (Lexical Input):

KEY(IF) '(' ID(x) OP('==') ')' INT(1) KEY(ELSE) INT(2) ';' ;

- Parser Output



Intro to Parsing



- Not every strings of tokens are valid
- Parser must distinguish between valid and invalid token strings.
- We need
 - A Language: to describe what is valid string?
 - Context Free Grammar
 - Capture Language Syntax
 - A method: to determine membership of inputs in this language.

Context Free Grammar

- A CFG consists of
 - A set of terminal T
 - A set of non-terminal N
 - A start symbol S ($S \in N$)
 - A set of production rules
 - $X \rightarrow Y_1 \dots Y_N$
 - $X \in N$
 - $Y_i \in \{N, T, \varepsilon\}$
- Ex: $S \rightarrow (S) \mid \varepsilon$
 - $N = \{S\}$
 - $T = \{ (,) , \varepsilon \}$

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:

$$X \rightarrow Y_1 \dots Y_n$$

3. Repeat 2 again and again until there are no non-terminals

$$X_1 \dots X_i \text{ ~~X~~ } X_{i+1} \dots X_n \rightarrow X_1 \dots X_i Y_1 \dots Y_k X_{i+1} \dots X_n$$

For the production rule $X \rightarrow Y_1 \dots Y_k$

$$\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \dots \rightarrow \alpha_n$$

$$\alpha_0 \xrightarrow{*} \alpha_n, n \geq 0$$

Context Free Grammar

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

$$\{a_1 \dots a_i \dots a_n \mid \forall i a_i \in T \wedge S \xrightarrow{*} a_1 \dots a_i \dots a_n\}$$

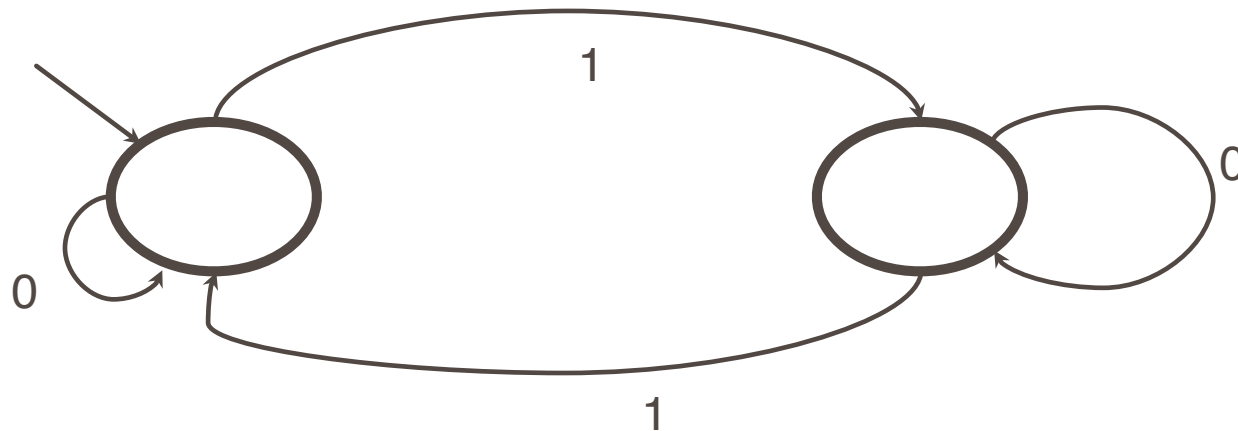
Context Free Grammar

- 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

Languages and Automata

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

Automata that accept odd numbers of 1



How many 1s it has accepted?

- Only solution is duplicate state

Automata do not have any memory

Intro to Parsing

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



Regular languages
cannot handle well

CFG: Simple Arithmetic expression

$E \rightarrow E + E$

$| E * E$

$| (E)$

$| \text{id}$

Languages can be generated: id , (id) , $(\text{id} + \text{id}) * \text{id}$, ...

CFG: Exercise

$$S \rightarrow aXa$$

$$X \rightarrow \varepsilon \mid bY$$

$$Y \rightarrow \varepsilon \mid cXc$$

Some Valid Strings are: aba, abcca, ...

Derivation

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

- Grammar

- $E \rightarrow E + E \mid E * E \mid (E) \mid id$

- String

- $id * id + id$

- Derivation

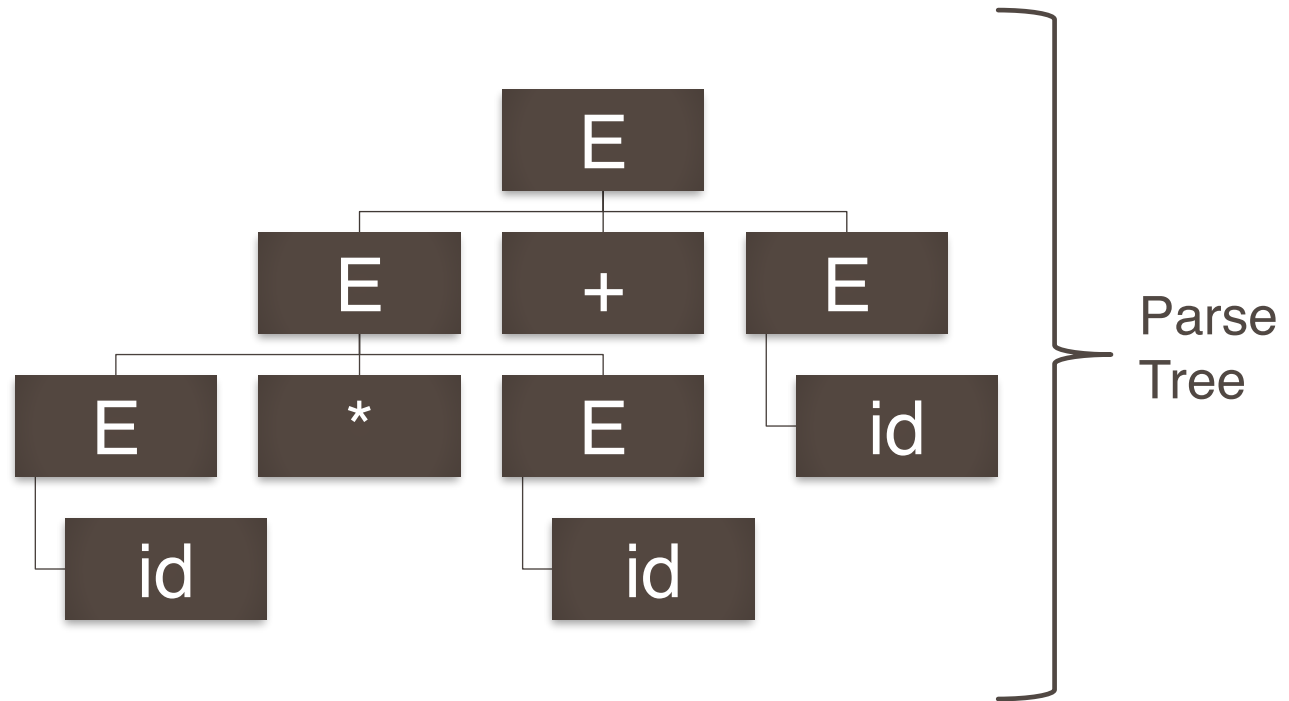
$E \rightarrow E + E$

$\rightarrow E * E + E$

$\rightarrow id * E + E$

$\rightarrow id * id + E$

$\rightarrow id * id + id$



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

$E \rightarrow E + E$

$\rightarrow E * E + E$

$\rightarrow id * E + E$

$\rightarrow id * id + E$

$\rightarrow id * id + id$

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

$E \rightarrow E + E$

$\rightarrow E + id$

$\rightarrow E * E + id$

$\rightarrow E * id + id$

$\rightarrow id * id + id$

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

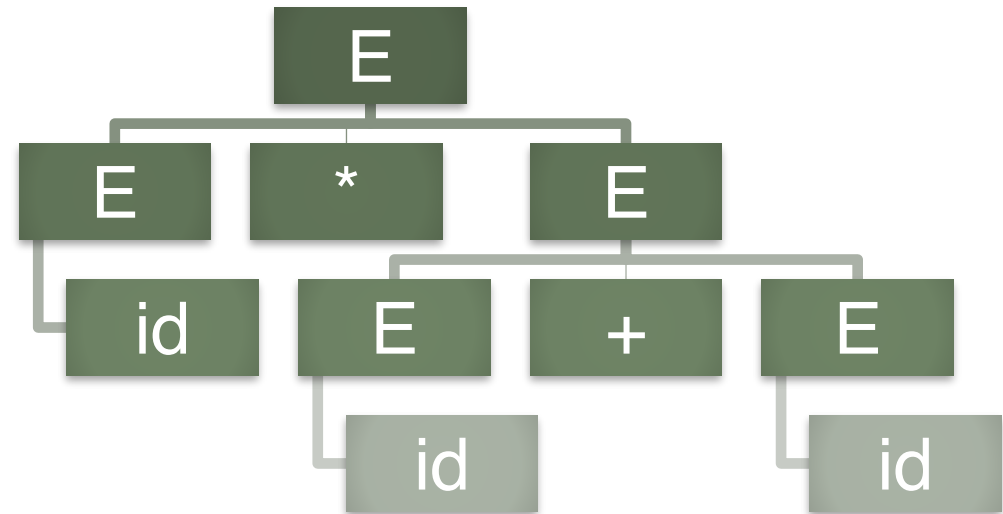
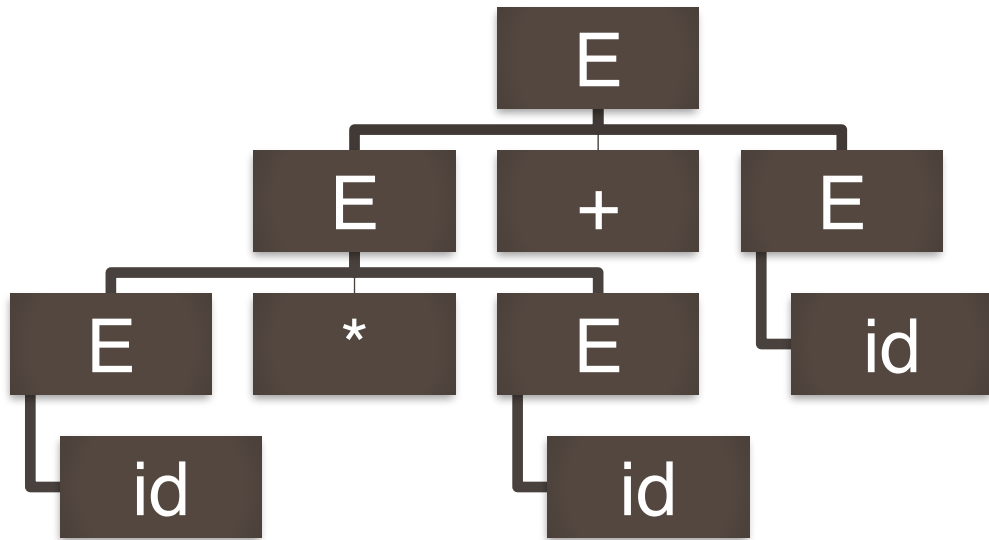
Ambiguity

- Grammar

- $E \rightarrow E + E \mid E * E \mid (E) \mid id$

- 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

Example of Ambiguous Grammar

- $S \rightarrow SS \mid a \mid b$

Resolving Ambiguity

- Most direct way to rewrite the grammar unambiguously

$$id * id + id$$

$$E = E' + E \mid E'$$

$$E' = id * E' \mid id \mid (E) * E' \mid (E)$$

Resolving Ambiguity

- Impossible to convert ambiguous to unambiguous grammar automatically
- Instead of rewriting
 - Use ambiguous grammar
 - Along with disambiguating rules
 - Eg, precedence and associativity rules
 - Enforces precedence of * over +
 - associativity: %left +

Abstract Syntax Trees

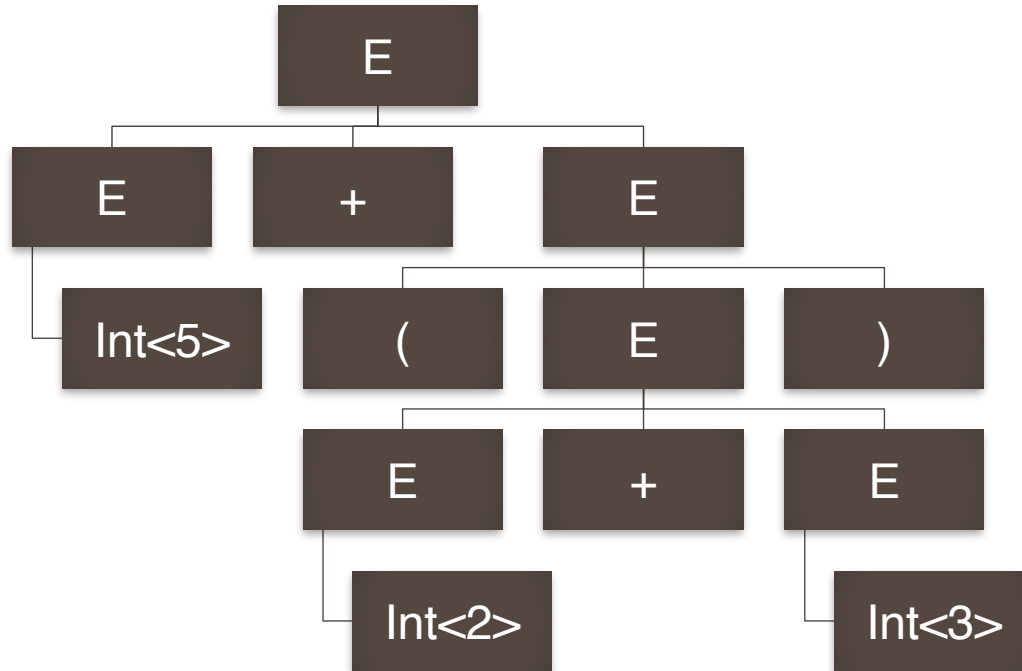
- A parser traces the derivation of a sequence of tokens
- But the rest of the compiler needs a structural representation of the program
- Abstract Syntax Trees
 - Like parse trees but ignore some details
 - Abbreviated as AST

Abstract Syntax Trees

- Grammar
 - $E \rightarrow \text{int} \mid (E) \mid E + E$
- String
 - $5 + (2 + 3)$
- After lexical analysis
 - $\text{Int}\langle 5 \rangle \text{'+' '(' Int}\langle 2 \rangle \text{'+' Int}\langle 3 \rangle \text{'})'}$

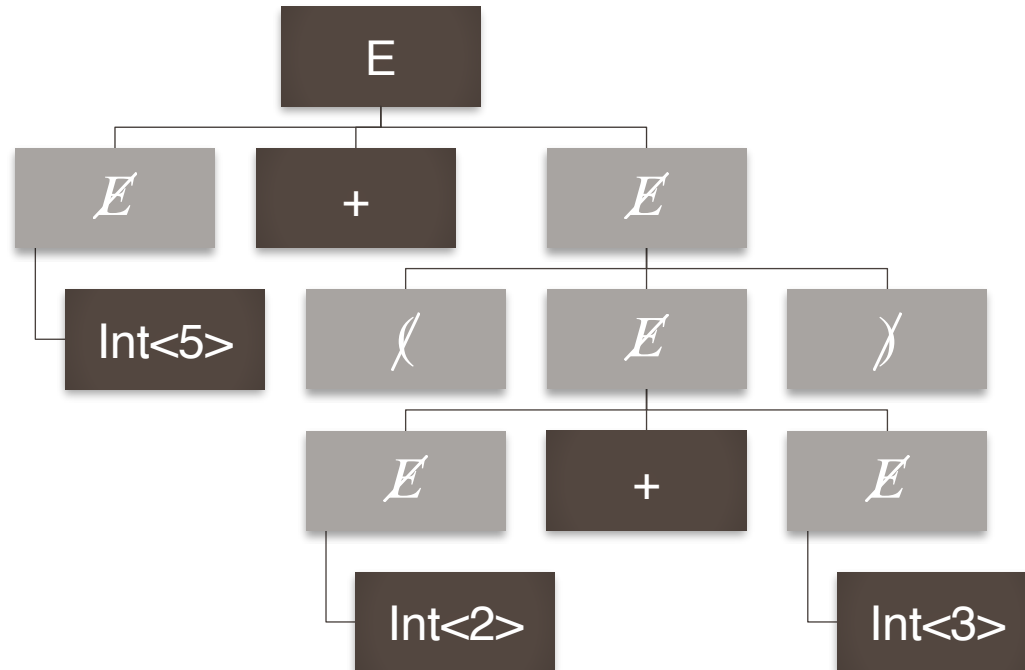
Abstract Syntax Trees: $5 + (2 + 3)$

Parse Trees



Abstract Syntax Trees: $5 + (2 + 3)$

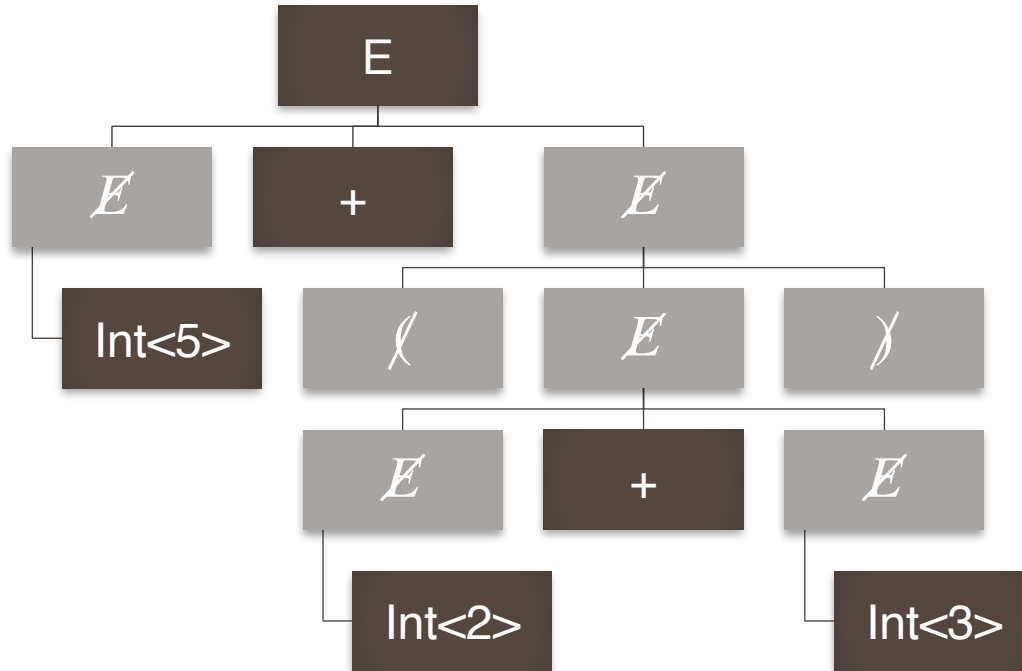
Parse Trees



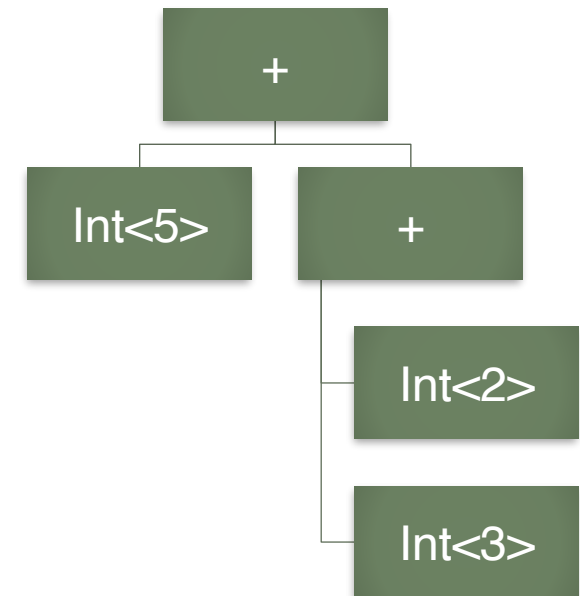
- Have too much information
 - Parentheses
 - Single-successor nodes

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

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 | Misspelling of identifiers, keywords, or operators. | ... \$... | Lexer |
| Syntax | Misplaced operators, semicolons, braces, switch-case statements, etc. | ... x*%... | Parser |
| Semantic | Type mismatches between operators and operands | ... int x; y = x(3);... | Type Checker |
| Correctness | Incorrect reasoning | Using = instead of == | tester/user |

Error Handling

- Error Handler should
 - Discover 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
 - Typically looks for “synchronizing” tokens
 - Typically the statement of expression terminators
 - Example: delimiters (; }, etc.)
 - Continue from there

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
 - $E \rightarrow \text{int} \mid E + E \mid (E) \mid \text{error int} \mid (\text{error})$



Error Productions

- Specify known common mistakes in the grammar
- Example:
 - Write $5x$ instead of $5 * x$
 - Add production rule $E \rightarrow .. \mid E E$
- 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

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:
 - $E \rightarrow T \mid T + E$
 - $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$
- Token Stream: (int<5>)
- Start with top level non-terminal E
 - Try the rules for E in order

Recursive Descent Parsing Example

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

E
|
T
|
int

mismatch: int does not match arrowhead (
backtrack

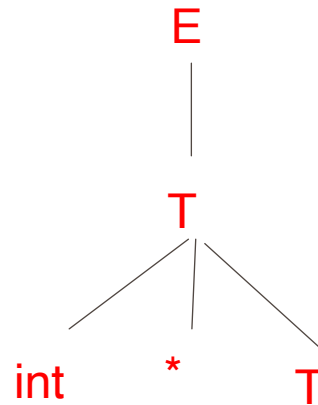
(int<5>)



Recursive Descent Parsing Example

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



backtrack

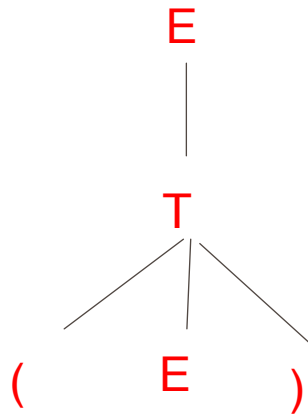
(int<5>)



Recursive Descent Parsing Example

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



Match! Advance input

(int<5>)

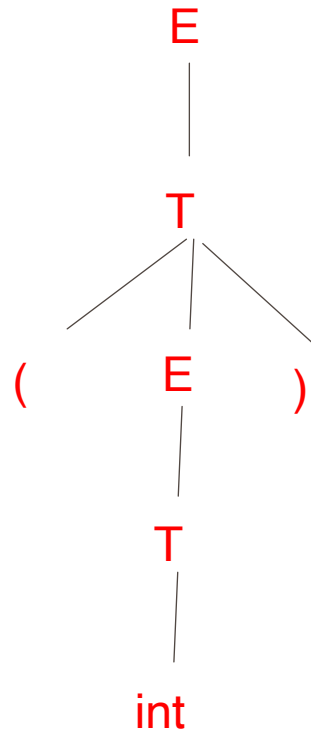


Recursive Descent Parsing Example

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

(int<5>)



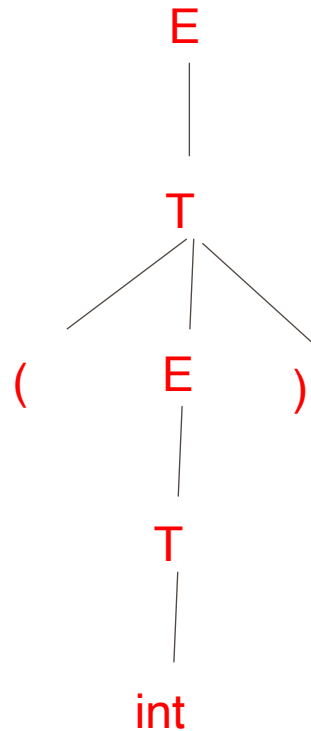
Match! Advance input

Recursive Descent Parsing Example

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

(int<5>)
↑



Match! Advance input

$$E \rightarrow E' \mid E' + E$$
$$E' \rightarrow -E' \mid id \mid (E)$$

Input: id + id

A Recursive Descent Parser. Preliminaries

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

A Top Down Parsing Algorithm

```
void A() {  
    Choose an A-production:  $A \rightarrow S_1 S_2 \dots S_k$ ;  
    for (i=1 or k) {  
        if ( $S_i$  is a nonterminal)  
            Call  $S_i()$ ;  
        else if ( $X_i ==$  current input TOKEN tok). /*terminal*/  
            next++;  
        else  
            /* Error */  
    }  
}
```

Recursion without
backtracking

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 *nextt++ == tok; }
```

- The n^{th} production of S:

```
bool Sn() { ... }
```

- Try all productions of S:

```
bool S() { ... }
```

A (Limited) Recursive Descent Parser

- For production $E \rightarrow T$

```
bool E1() { return T(); }
```

- For production $E \rightarrow T + E$

```
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( ));  
}
```

Grammar:

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

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());  
}
```

Grammar:

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

Recursive Descent Parsing

- To start the parser
 - Initialize next to point to first token
 - Invoke $E()$ (start symbol)

Example

Grammar:

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

Input: (int)

Code:

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

```
bool E1() { return T(); }
```

```
bool E2() { return T() && term(PLUS) && E(); }
```

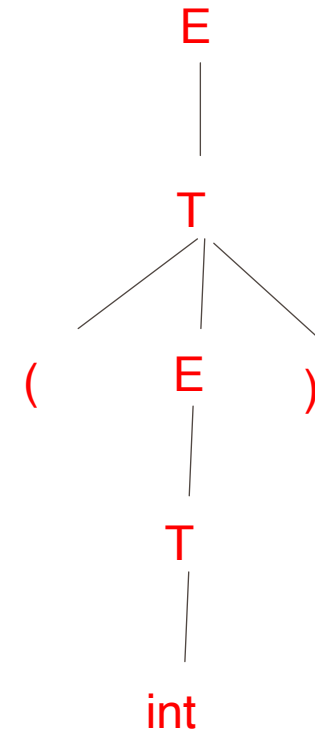
```
bool E() { TOKEN *save = next;  
          return (next = save, E1()) || (next = save, E2()); }
```

```
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()); }
```



Example

Grammar:

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

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:

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

Input: $\text{int} * \text{int}$

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

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

- 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 $\alpha A'$.
- Then, after seeing the input derived from α , we expand A' to β_1 or β_2 (left-factored)
- The original productions become:

$$A \rightarrow \alpha A', A' \rightarrow \beta_1 \mid \beta_2$$

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S a$
`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
 $S \rightarrow^+ S\alpha$ for some α
- Recursive descent does not work for left recursive grammar

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S \alpha \mid \beta$$

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

$$S \rightarrow \beta S'$$

$$S' \rightarrow \alpha S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

$$S \rightarrow S \alpha_1 \mid \dots \mid S \alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

- All strings derived from S start with one of β_1, \dots, β_m and continue with several instances of $\alpha_1, \dots, \alpha_n$

- Rewrite as

$$S \rightarrow \beta_1 S' \mid \dots \mid \beta_m S'$$

$$S' \rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon$$

General Left Recursion

- The grammar

$$S \rightarrow A \alpha \mid \delta$$

$$A \rightarrow S \beta$$

is also left-recursive because

$$S \rightarrow^+ S \beta \alpha$$

- This left-recursion can also be eliminated

Announcement

- Midterm Next Wednesday (October 21)
- Lexical and Syntactic Analysis

Break Out Session

- $S \rightarrow Aa \mid b$
- $A \rightarrow Ac \mid Sd \mid \epsilon$ $S \rightarrow Aa = Ac a = Acca$
- Remove Recursion.

-
- $S \rightarrow Aa \mid b.$
 - $A \rightarrow bdA' \mid A'$
 - $A' \rightarrow cA' \mid adA' \mid a \mid \epsilon$

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 $A \rightarrow \alpha$ 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

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

- 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

$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Factor out common prefixes of productions

$$E \rightarrow T X$$
$$X \rightarrow + E \mid \varepsilon$$
$$T \rightarrow (E) \mid \text{int} Y$$
$$Y \rightarrow * T \mid \varepsilon$$

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T X$

$X \rightarrow + E \mid \varepsilon$

$T \rightarrow (E) \mid \text{int } Y$

$Y \rightarrow * T \mid \varepsilon$

- 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 $[E, \text{int}]$ entry
 - “When current non-terminal is E and next input is int , use production $E \rightarrow T X$ ”
 - 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 $Y \rightarrow \epsilon$

LL(1) Parsing Tables. Errors

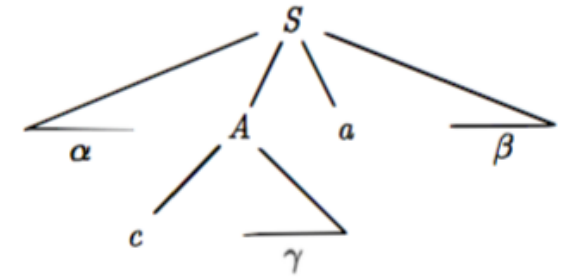
- Blank entries indicate error situations
- Consider the $[E, *]$ entry
 - “There is no way to derive a string starting with $*$ from non-terminal E ”

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.
- $\text{FIRST}(\alpha)$, 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 $\text{FIRST}(\alpha)$.
 - if $\alpha \xrightarrow{*} cY$, the c is in $\text{FIRST}(\alpha)$.
- $\text{FOLLOW}(A)$, 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 $A \rightarrow \alpha$, & token t
- $T[A,t] = \alpha$ in two cases:
- If $\alpha \rightarrow^* t \beta$
 - α can derive a t in the first position
 - We say that $t \in \text{First}(\alpha)$
- If $A \rightarrow \alpha$ and $\alpha \rightarrow^* \varepsilon$ and $S \rightarrow^* \beta A t \delta$
 - 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 $t \in \text{Follow}(A)$

First Sets. Example

- grammar

$E \rightarrow T X$

$X \rightarrow + E \mid \varepsilon$

$T \rightarrow (E) \mid \text{int } Y$

$Y \rightarrow * T \mid \varepsilon$

- First sets : Breakout room

$\text{First}(()) = \{ (\}$

$\text{First}()) = \{) \}$

$\text{First}(\text{int}) = \{ \text{int} \}$

$\text{First}(+) = \{ + \}$

$\text{First}(*) = \{ * \}$

$\text{First}(E) = ?$

$\text{First}(T) = ?$

$\text{First}(X) = ?$

$\text{First}(Y) = ?$

Computing Follow Sets

- Definition:

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

- Intuition:

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

Follow Sets. Example

- Recall the grammar

$E \rightarrow T X$

$X \rightarrow + E \mid \varepsilon$

$T \rightarrow (E) \mid \text{int } Y$

$Y \rightarrow * T \mid \varepsilon$

- Follow sets

$\text{Follow}(+) = \{ \text{int}, (\}$

$\text{Follow}(() = \{ \text{int}, (\}$

$\text{Follow}(*) = \{ \text{int}, (\}$

$\text{Follow}()) = \{ +,) , \$ \}$

$\text{Follow}(\text{int}) = \{ *, +,) , \$ \}.$

$\text{Follow}(E) = \{), \$ \}$

$\text{Follow}(T) = \{ +,) , \$ \}$

$\text{Follow}(Y) = \{ +,) , \$ \}$

$\text{Follow}(X) = \{ \$,) \}$

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production $A \rightarrow \alpha$ in G do:
 - For each terminal $t \in \text{First}(\alpha)$ do
 - $T[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
 - $T[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
 - $T[A, \$] = \alpha$

LL(1) Parsing Table Example

- Left-factored grammar

$E \rightarrow T X$

$X \rightarrow + E \mid \varepsilon$

$T \rightarrow (E) \mid \text{int } Y$

$Y \rightarrow * T \mid \varepsilon$

- The LL(1) parsing table:

Rules:

For each production $A \rightarrow \alpha$ in G do:

For each terminal $t \in \text{First}(\alpha)$ do

$T[A, t] = \alpha$

If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do

$T[A, t] = \alpha$

If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do

$T[A, \$] = \alpha$

| Left-most non- terminals | | next input tokens | | | | | |
|--------------------------------|---|-------------------|----|---------------|-------|---------------|---------------|
| | | int | * | + | (|) | \$ |
| | 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 multiple defined then G is not LL(1) [Eg: $S \rightarrow Sa|b$]
 - 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:
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E) \cdot$$
- Consider the string: `int * int + int`

Bottom-Up Parsing

- Revert to the “natural” grammar for our example:

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} \mid (E) \cdot$

- Consider the string: $\text{int} * \text{int} + \text{int}$
- Bottom-up parsing **reduces** a string to the start symbol by **inverting** productions:

$\text{int} * \text{int} + \text{int}$

$T \rightarrow \text{int}$

$\text{int} * T + \text{int}$

$T \rightarrow \text{int} * T$

$T + \text{int}$

$T \rightarrow \text{int}$

$T + T$

$E \rightarrow T$

$T + E$

$E \rightarrow T + E$

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`

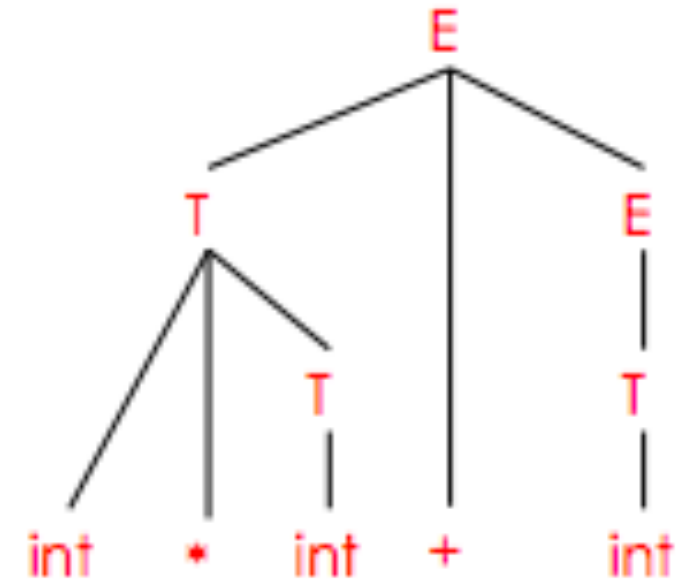
`E`

Bottom-Up Parsing

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

int * int + int
int * T + int
T + int
T + T
T + E
E

$T \rightarrow \text{int}$
 $T \rightarrow \text{int} * T$
 $T \rightarrow \text{int}$
 $E \rightarrow T$
 $E \rightarrow T + E$



A trivial Bottom-Up Parsing Algorithm

Let I = input string

repeat

 pick a non-empty substring β of I

 where $X \rightarrow \beta$ is a production

 if no such β , backtrack

 replace one β by X in I

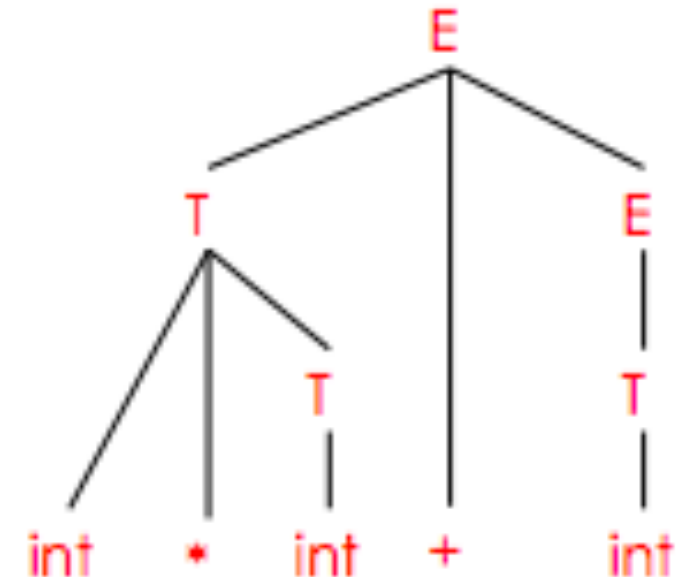
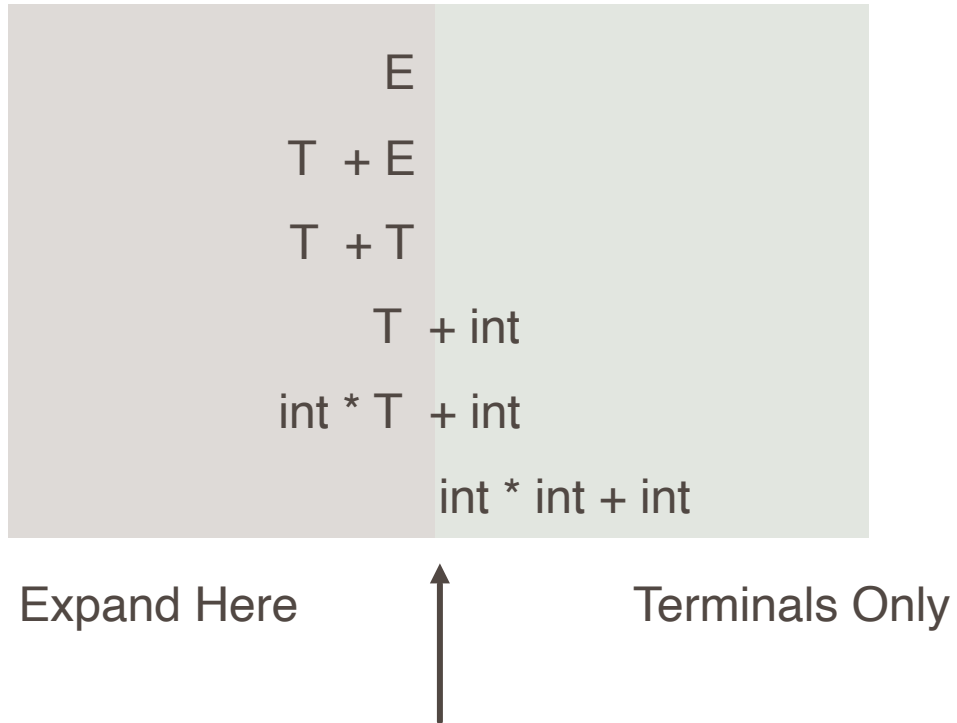
until $I = \text{"S"}$ (the start symbol) or all possibilities are exhausted

Bottom-Up Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T$

- 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 $| x_1 x_2 \dots x_n$



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 $X \rightarrow \beta$
 - Then ω is a string of terminals
- Why? Because $\alpha X\omega \rightarrow \alpha\beta\omega$ 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 $ABClxyz \Rightarrow ABCxlyz$
- Reduce: Apply an inverse production at the right end of the left string
 - If $A \rightarrow xy$ is a production, then $Cbxylijk \Rightarrow CbAlijk$

The Example with Reductions Only

int * int | + int

reduce $T \rightarrow \text{int}$

int * T | + int

reduce $T \rightarrow \text{int} * T$

T + int |

reduce $T \rightarrow \text{int}$

T + T |

reduce $E \rightarrow T$

T + E |

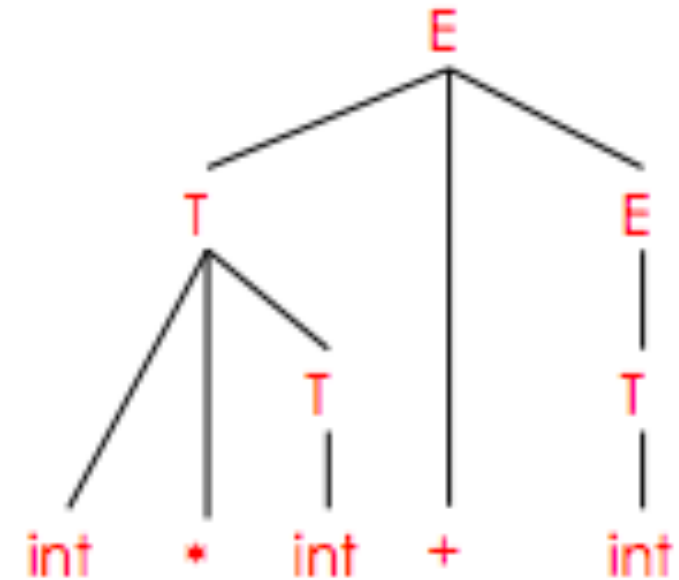
reduce $E \rightarrow T + E$

E |

An Example with Shift-Reduce Parsing

| | | |
|-----------|-----------------|--------|
| | 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 | | |

stack input



The Stack

- Left string can be implemented by a stack
 - Top of the stack is the \mid
- 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:
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$
- Consider step $\text{int} \mid * \text{int} + \text{int}$
 - We could reduce by $T \rightarrow \text{int}$ giving $T \mid * \text{int} + \text{int}$
 - 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 $\alpha \beta \omega$
 - $\alpha \beta$ 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

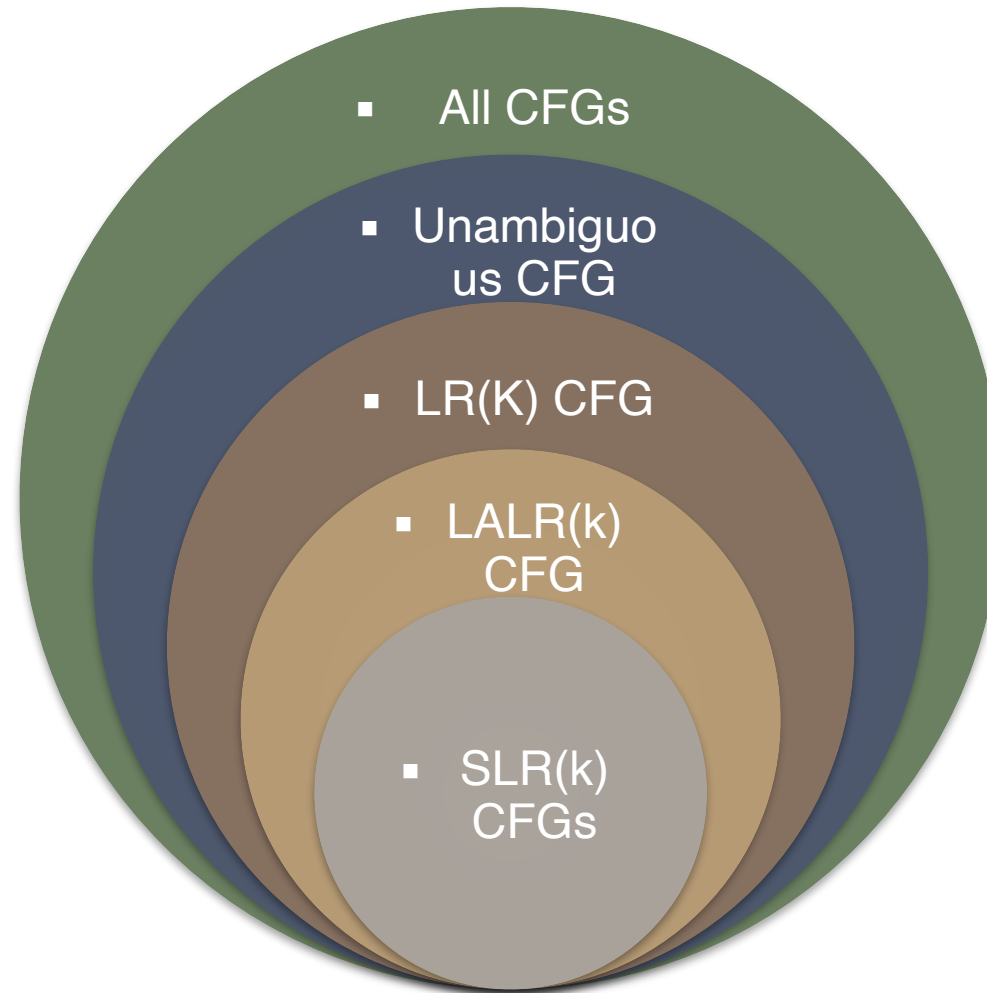
Summary of Handles

- 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 \mid 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 $\alpha\omega$ 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

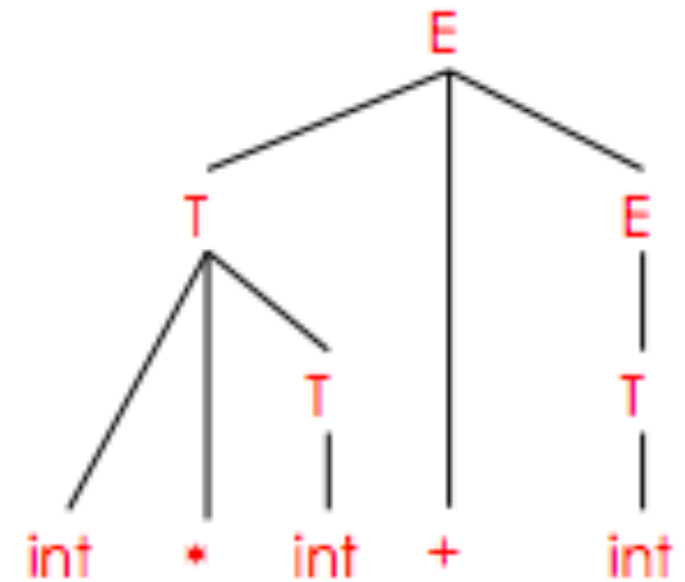
$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T$

| | |
|---------|-----------------|
| | E |
| T + E | |
| T + T | |
| | T + int |
| int * T | + int |
| | int * int + int |

viable prefixes

Terminals



Items

- An item is a production with a “.” somewhere on the rhs
- The items for $T \rightarrow (E)$ are
 - $T \rightarrow \cdot(E)$
 - $T \rightarrow (\cdot E)$
 - $T \rightarrow (E \cdot)$
 - $T \rightarrow (E) \cdot$
- The only item for $X \rightarrow \varepsilon$ is $X \rightarrow \cdot$
- Items are often called “LR(0) items”

Intuition

- The problem of 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 $T \rightarrow (E) \cdot$
 - Will be reduced after the next shift
 - Item $T \rightarrow (E.)$ 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
 - $\text{Prefix}_1 \text{Prefix}_2 \dots \text{Prefix}_{n-1} \text{Prefix}_n$
- 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 $X_{i-1} \rightarrow \text{Prefix}_{i-1} X_i \beta$ for some β
- Recursively, $\text{Prefix}_{k+1} \dots \text{Prefix}_n$ eventually reduces to the missing part of α_k

An Example

- Consider the string $(\text{int} * \text{int})$:
 - $(\text{int} * \text{int})$ is a state of a shift-reduce parse
 - “(” is a prefix of the rhs of $T \rightarrow (E)$
 - “ ϵ ” is a prefix of the rhs of $E \rightarrow T$
 - “ $\text{int} *$ ” is a prefix of the rhs of $T \rightarrow \text{int} * T$
- The “stack of items”
 - $T \rightarrow (.E)$
 - $E \rightarrow .T$
 - $T \rightarrow \text{int} * .T$
- Says
 - We’ve seen “(” of $T \rightarrow (E)$
 - We’ve seen ϵ of $E \rightarrow T$
 - We’ve seen $\text{int} *$ of $T \rightarrow \text{int} * T$

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

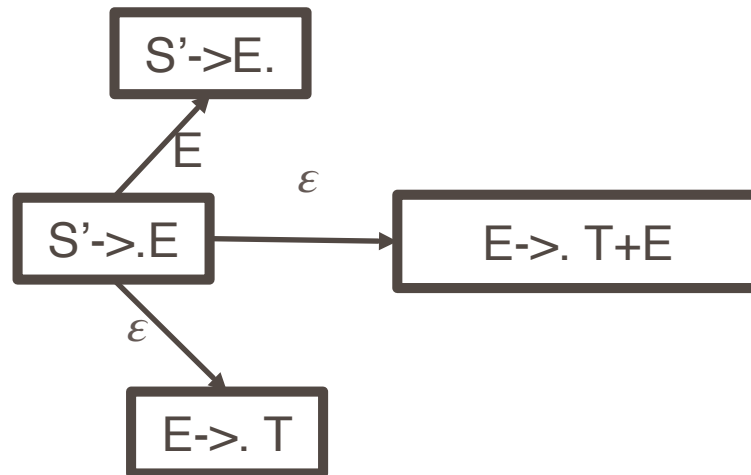
1. Add a dummy production $S' \rightarrow S$ to G
2. The NFA states are the items of G
 - Including the extra production
 - NFA takes the stack as input
 - $\text{NFA}(\text{stack}) \rightarrow \text{accept/reject}$
3. For item $E \rightarrow \alpha.X\beta$ add transition
 $E \rightarrow \alpha.X\beta \xrightarrow{x} E \rightarrow \alpha X.\beta$
4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add
 $E \rightarrow \alpha.X\beta \xrightarrow{\varepsilon} X \rightarrow \gamma$
5. Every state is an accepting state
6. Start state is $S' \rightarrow .S$

Recognizing VP

$S' \rightarrow E$

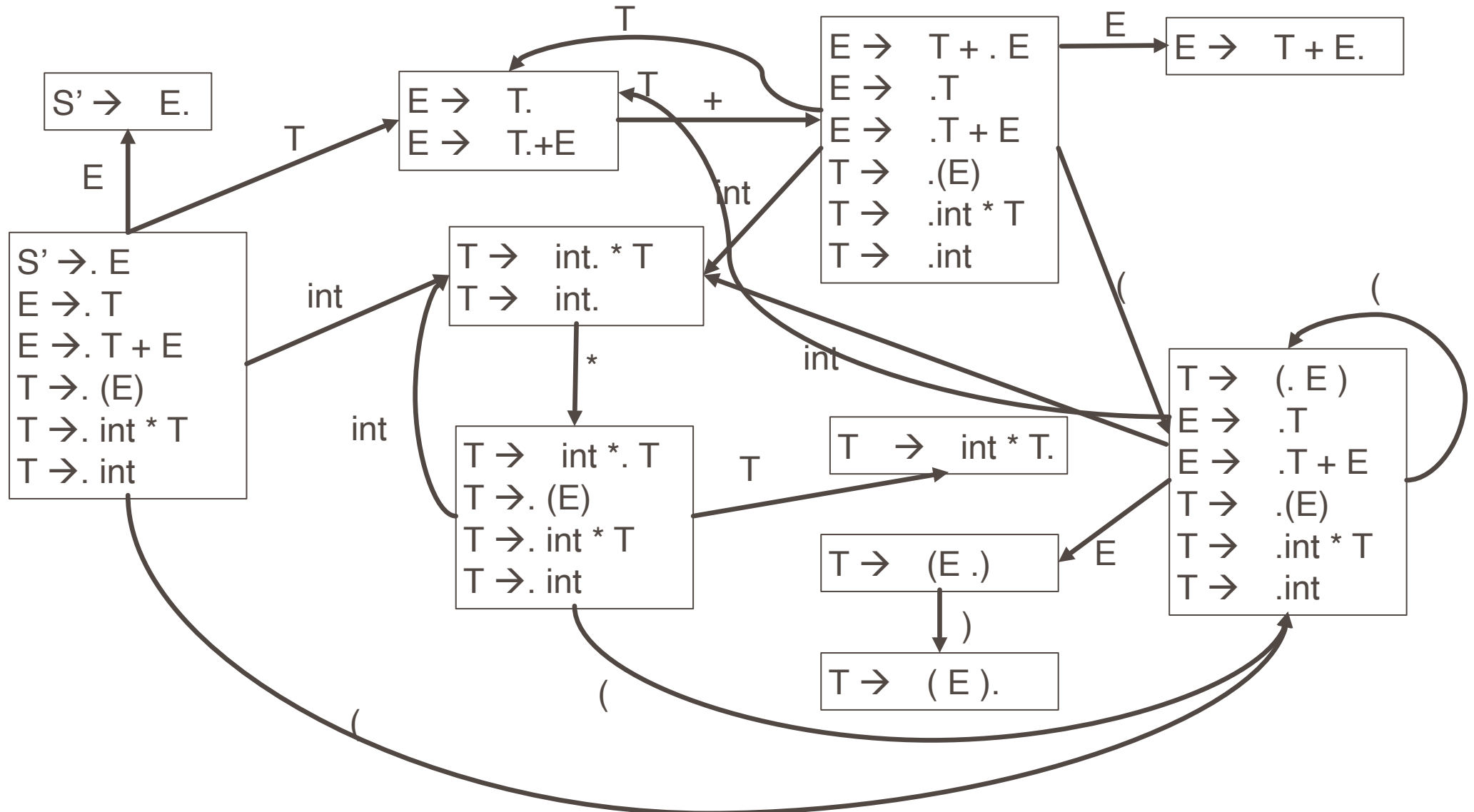
$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$



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 $X \rightarrow \beta.\gamma$ is valid for a viable prefix $\alpha\beta$ if
$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \gamma \omega \text{ by a right-most derivation}$$
- After parsing $\alpha\beta$, 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 $T \rightarrow (.E)$ is valid for prefixes $(\ ((((((($

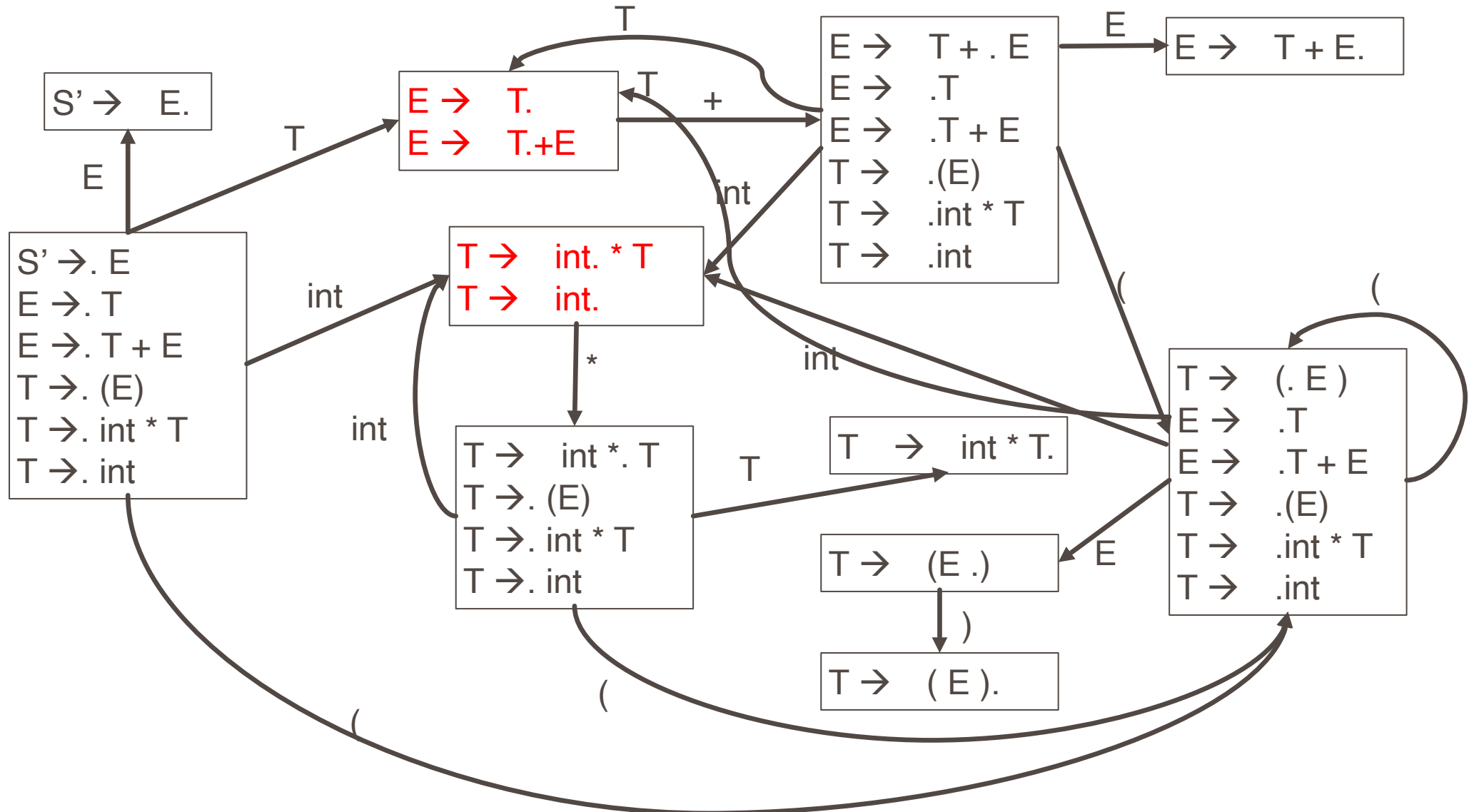
LR(o) Parsing

- 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



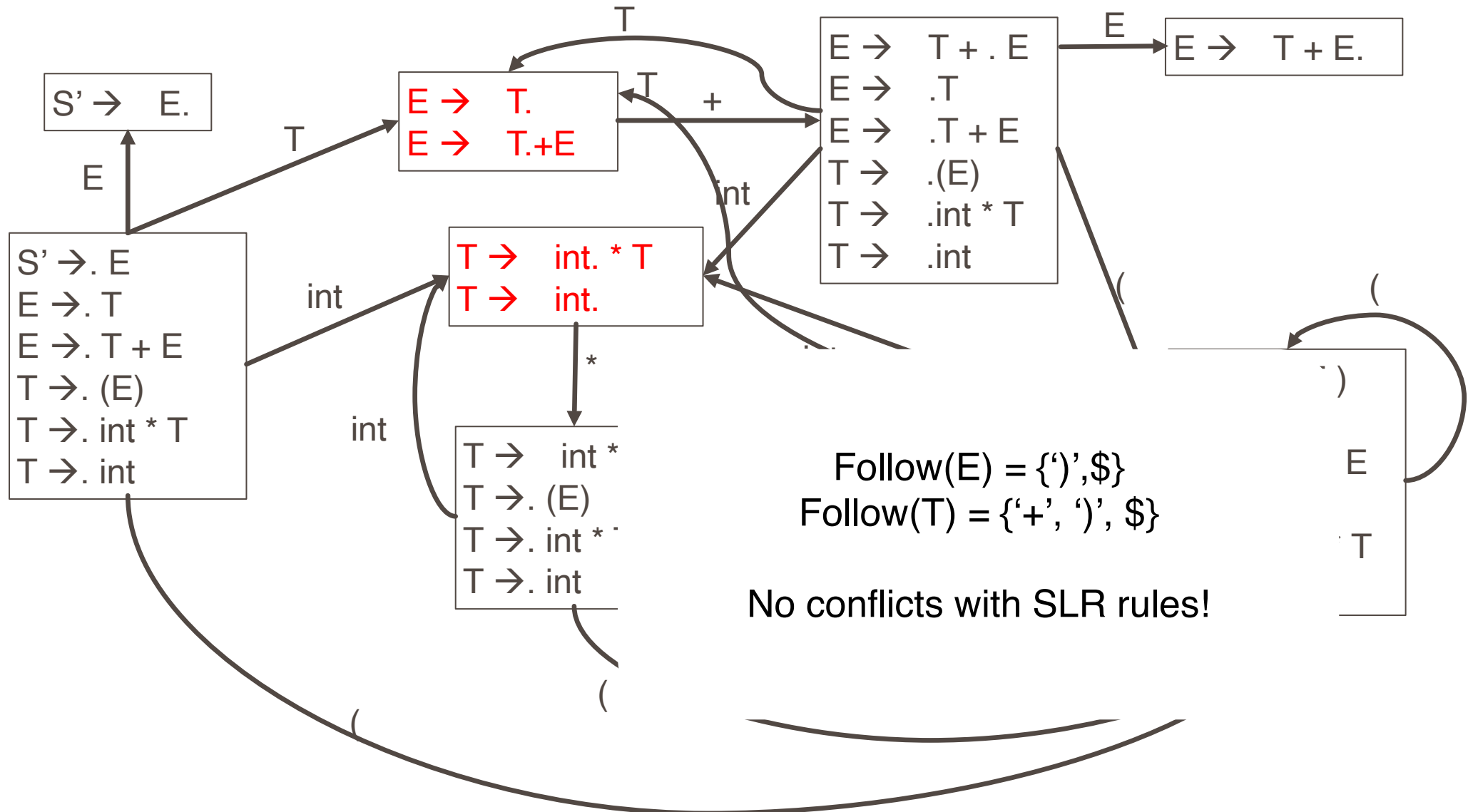
SLR

- 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 $X \rightarrow \beta$.
 - $t \in \text{Follow}(X)$
- 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

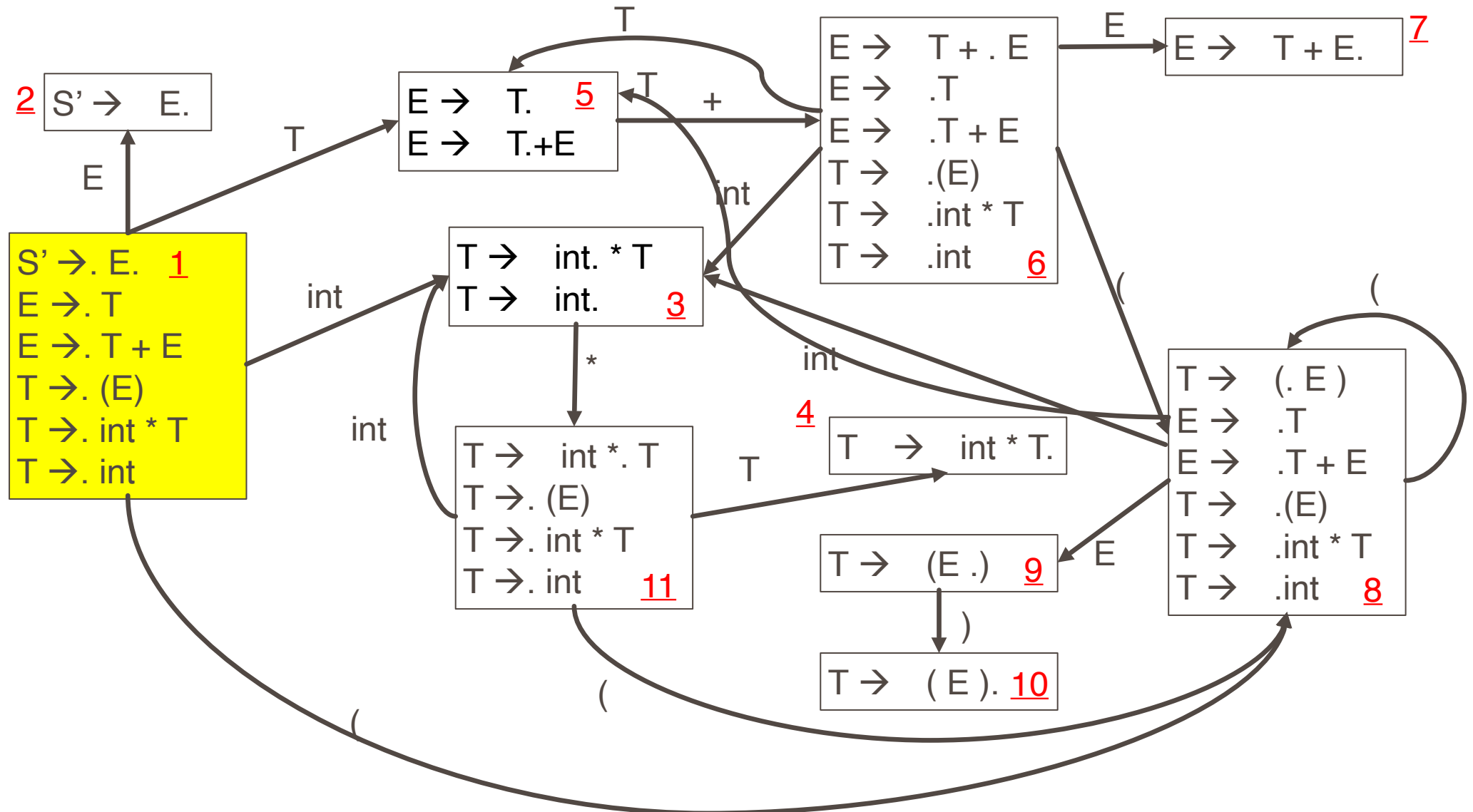
SLR Conflicts



Naïve SLR Parsing Algorithm

1. Let M be DFA for viable prefixes of G
2. Let $lx_1 \dots x_n \$$ be initial configuration
3. Repeat until configuration is $SI\$$
 - Let $\alpha\omega$ 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 $X \rightarrow \beta. a \gamma \in I$
 - Reduce if $X \rightarrow \beta. \in I$ and $a \in \text{Follow}(X)$
 - Report parsing error if neither applies

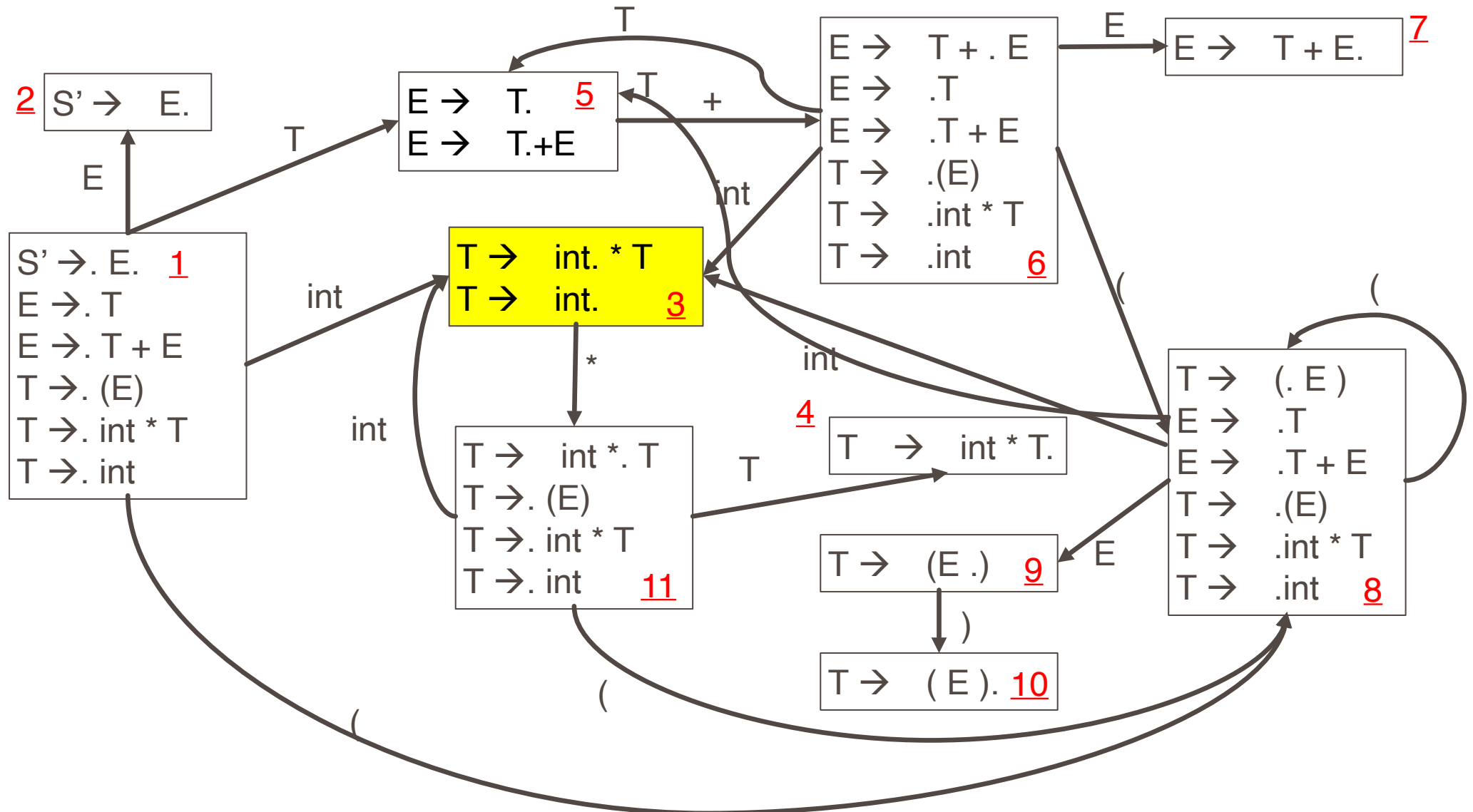
Parsing `|int * int $`



Parsing $\text{int} * \text{int} \$$

| Configuration | DFA Halt State | Action |
|--------------------------------|----------------|--------|
| $\text{int} * \text{int} \$$ | 1 | shift |
| $\text{int} * \text{int} \$$ | | |

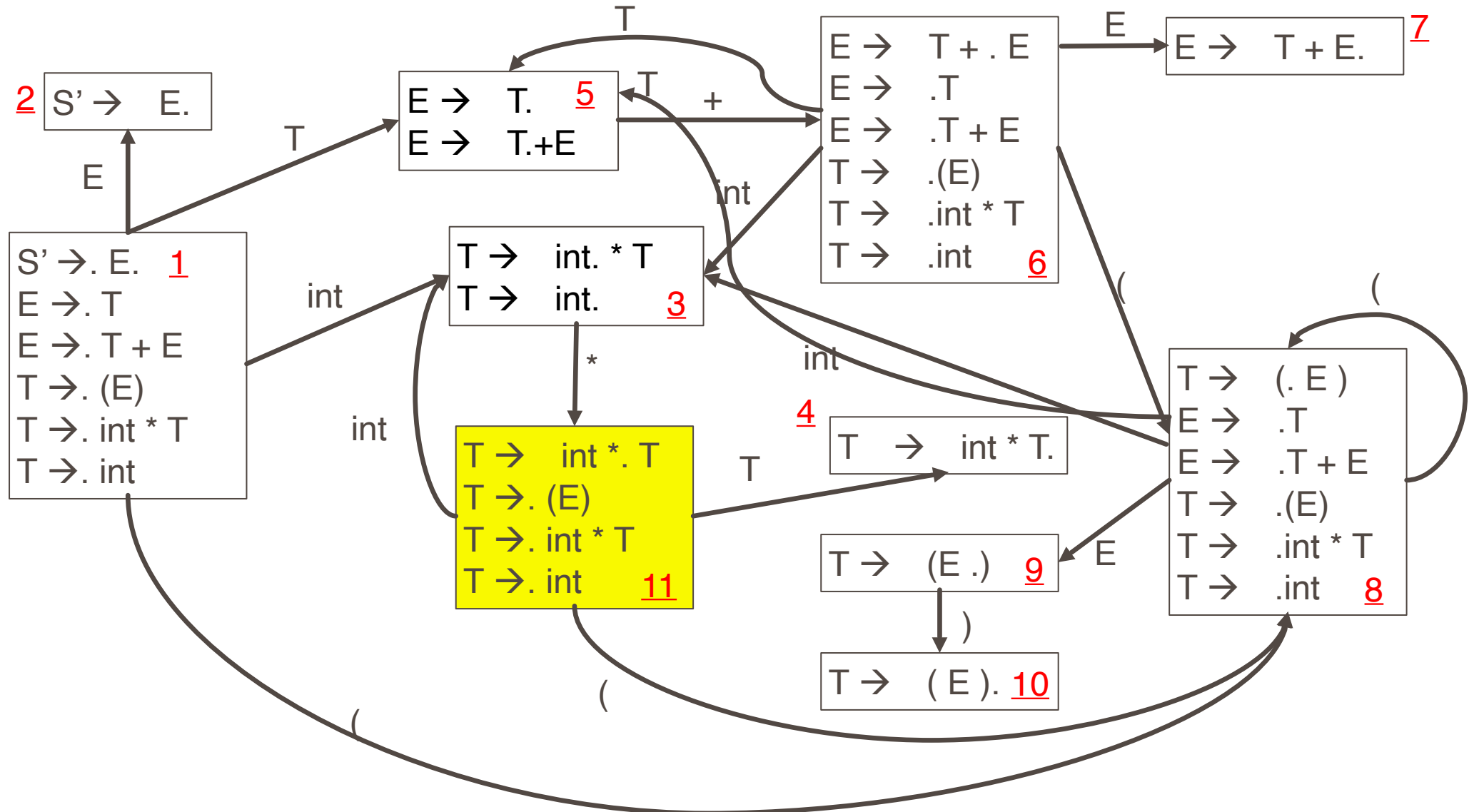
Parsing int| * int \$



Parsing |int * int \$

| Configuration | DFA Halt State | Action |
|---------------|-------------------------|--------|
| lint * int\$ | 1 | shift |
| int * int\$ | 3 (* not in Follow(T)) | shift |
| int * int\$ | | |

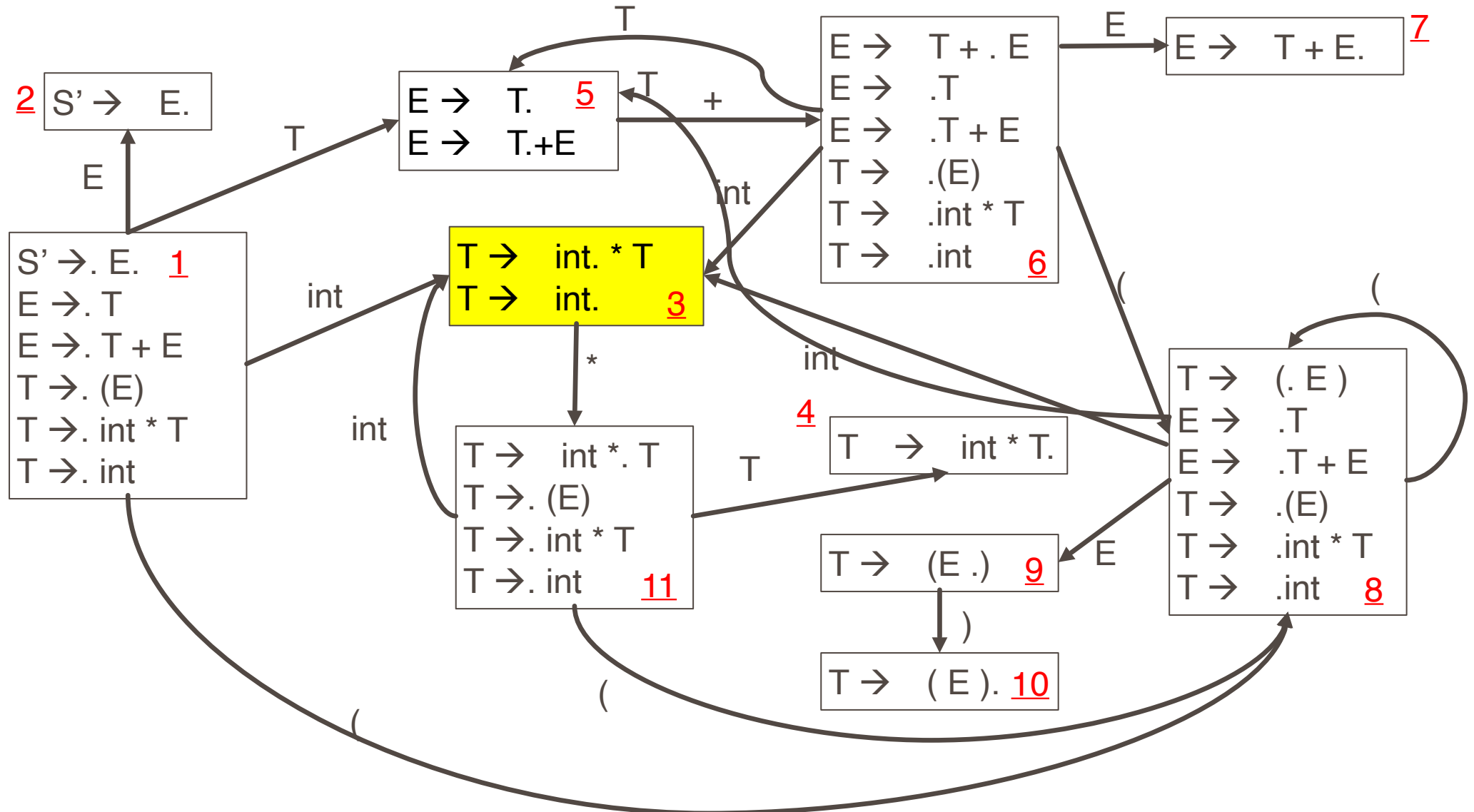
Parsing int * |int \$



Parsing $\text{int} * \text{int} \$$

| Configuration | DFA Halt State | Action |
|--------------------------------|-------------------------|--------|
| $\text{int} * \text{int} \$$ | 1 | shift |
| $\text{int} * \text{int} \$$ | 3 (* not in Follow(T)) | shift |
| $\text{int} * \text{int} \$$ | 11 | shift |
| $\text{int} * \text{int} \$$ | | |

Parsing int| * int \$



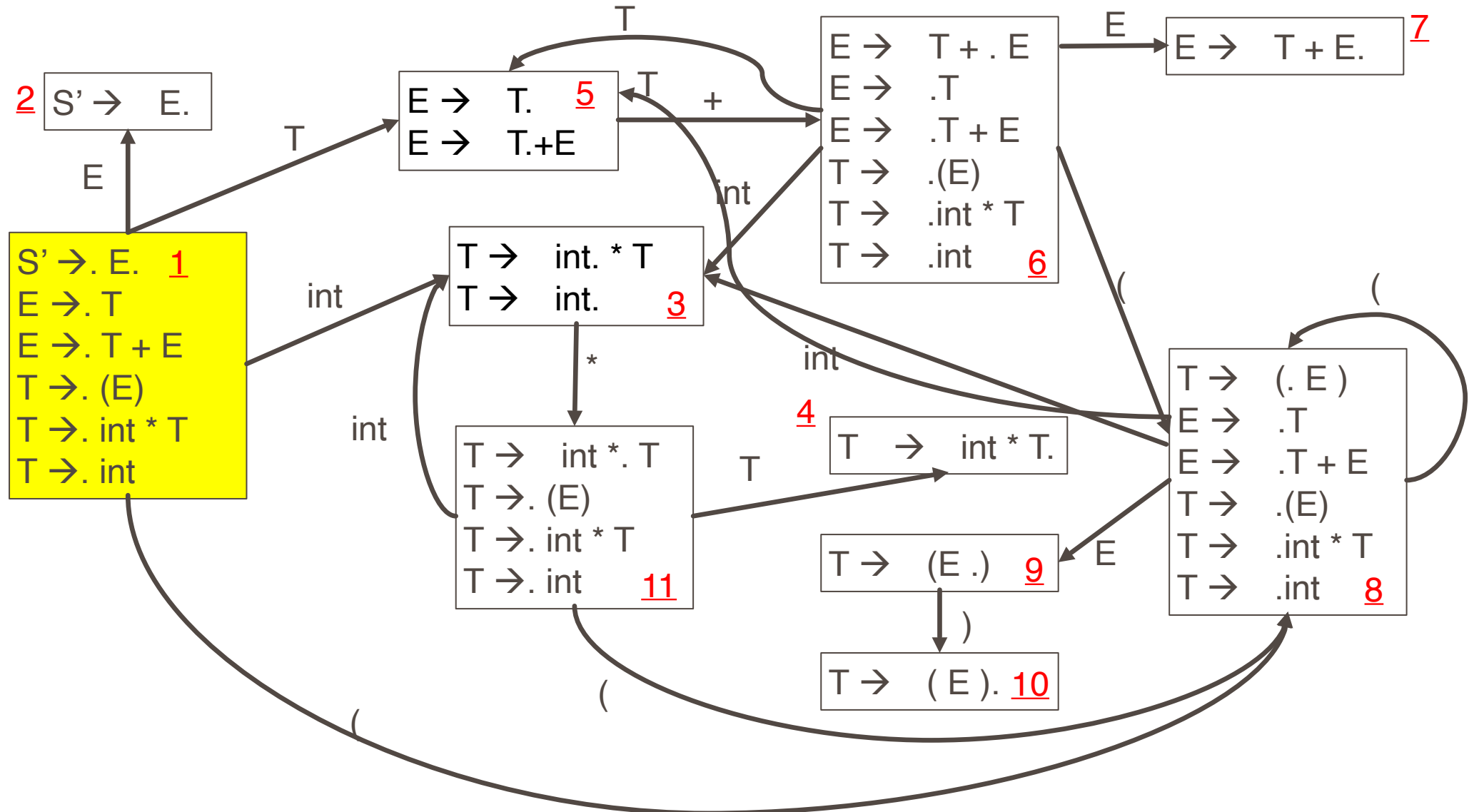
Parsing $\text{int} * \text{int} \$$

| Configuration | DFA Halt State | Action |
|--------------------------------|-------------------------|-----------------------------------|
| $\text{int} * \text{int} \$$ | 1 | shift |
| $\text{int} * \text{int} \$$ | 3 (* not in Follow(T)) | shift |
| $\text{int} * \text{int} \$$ | 11 | shift |
| $\text{int} * \text{int} \$$ | 3 (\$ Follow (T)) | reduce $T \rightarrow \text{int}$ |

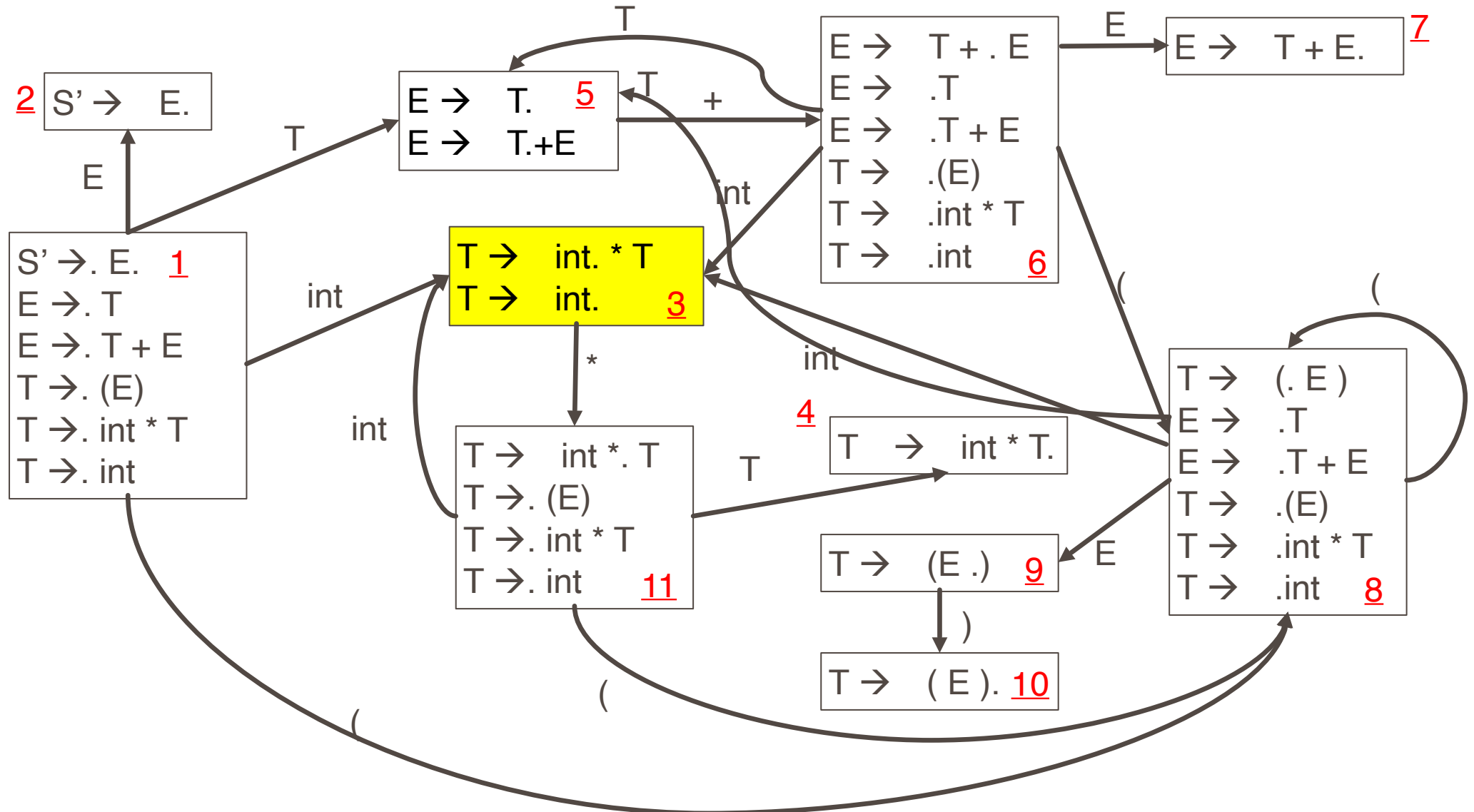
Parsing $\text{int} * \text{int} \$$

| Configuration | DFA Halt State | Action |
|--------------------------------|-------------------------|-----------------------------------|
| $\text{int} * \text{int} \$$ | 1 | shift |
| $\text{int} * \text{int} \$$ | 3 (* not in Follow(T)) | shift |
| $\text{int} * \text{int} \$$ | 11 | shift |
| $\text{int} * \text{int} \$$ | 3 (\$ Follow (T)) | reduce $T \rightarrow \text{int}$ |
| $\text{int} * T \$$ | | |

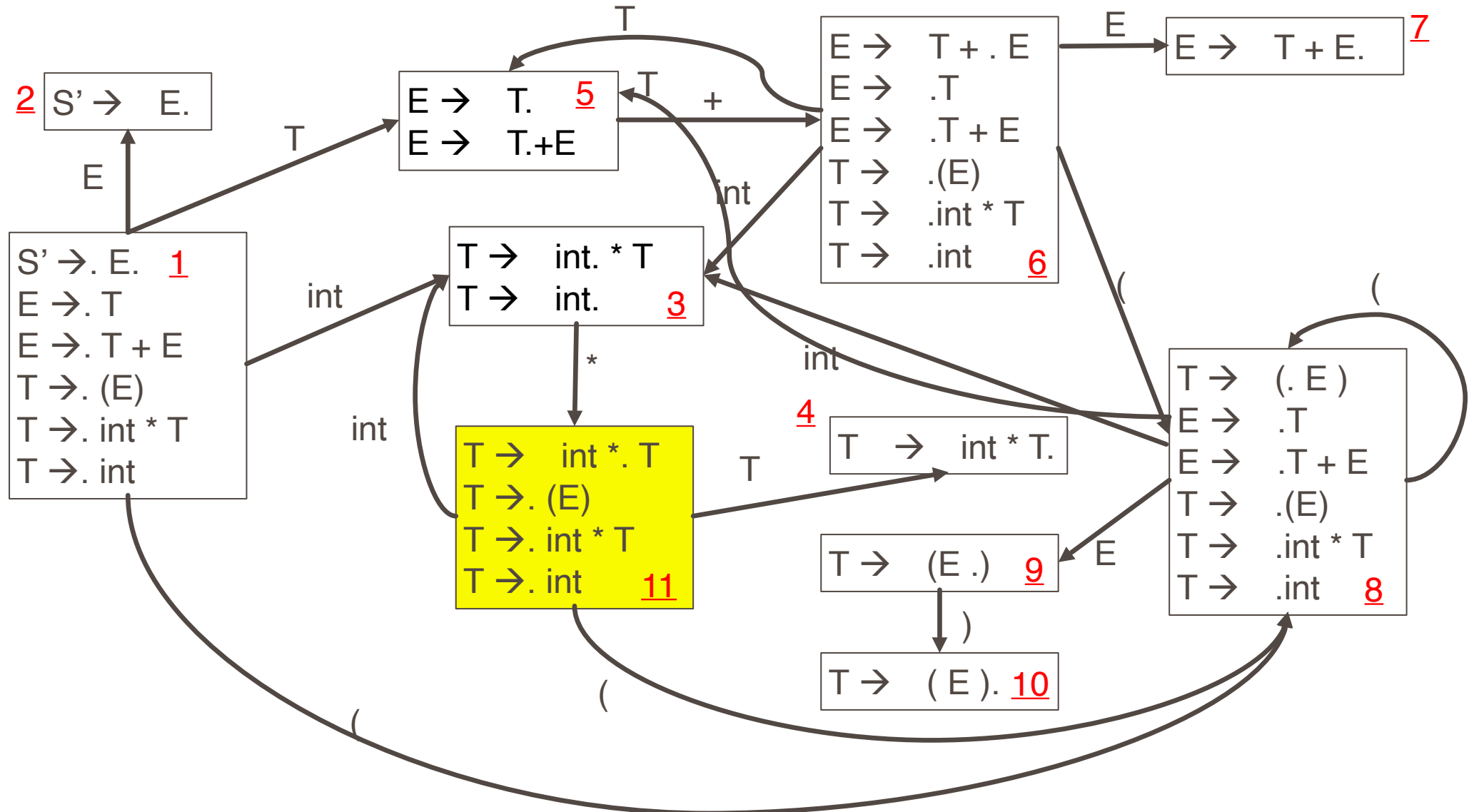
Parsing $\text{int} * T \mid \$$



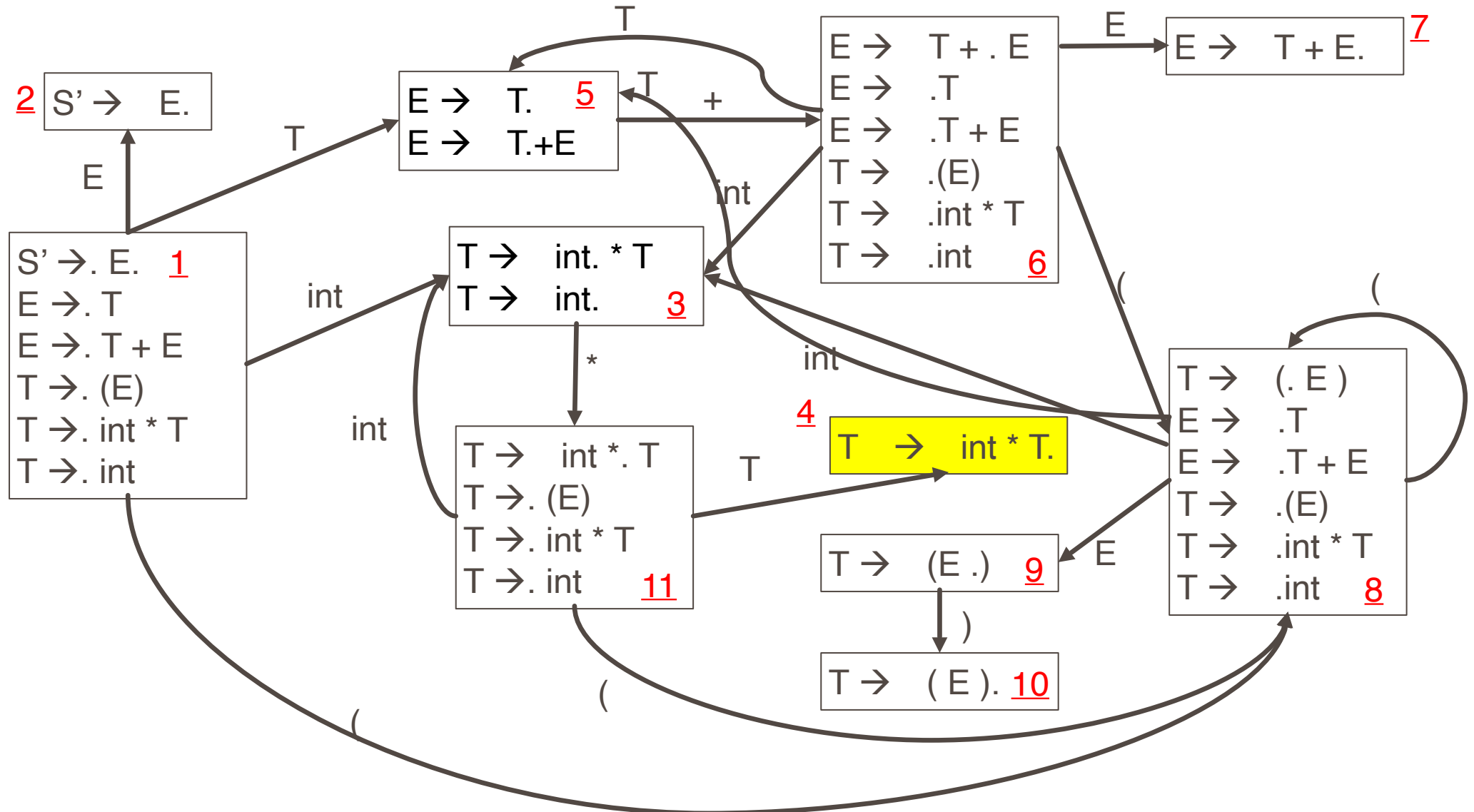
Parsing int * T | \$



Parsing int * T | \$



Parsing int * T | \$



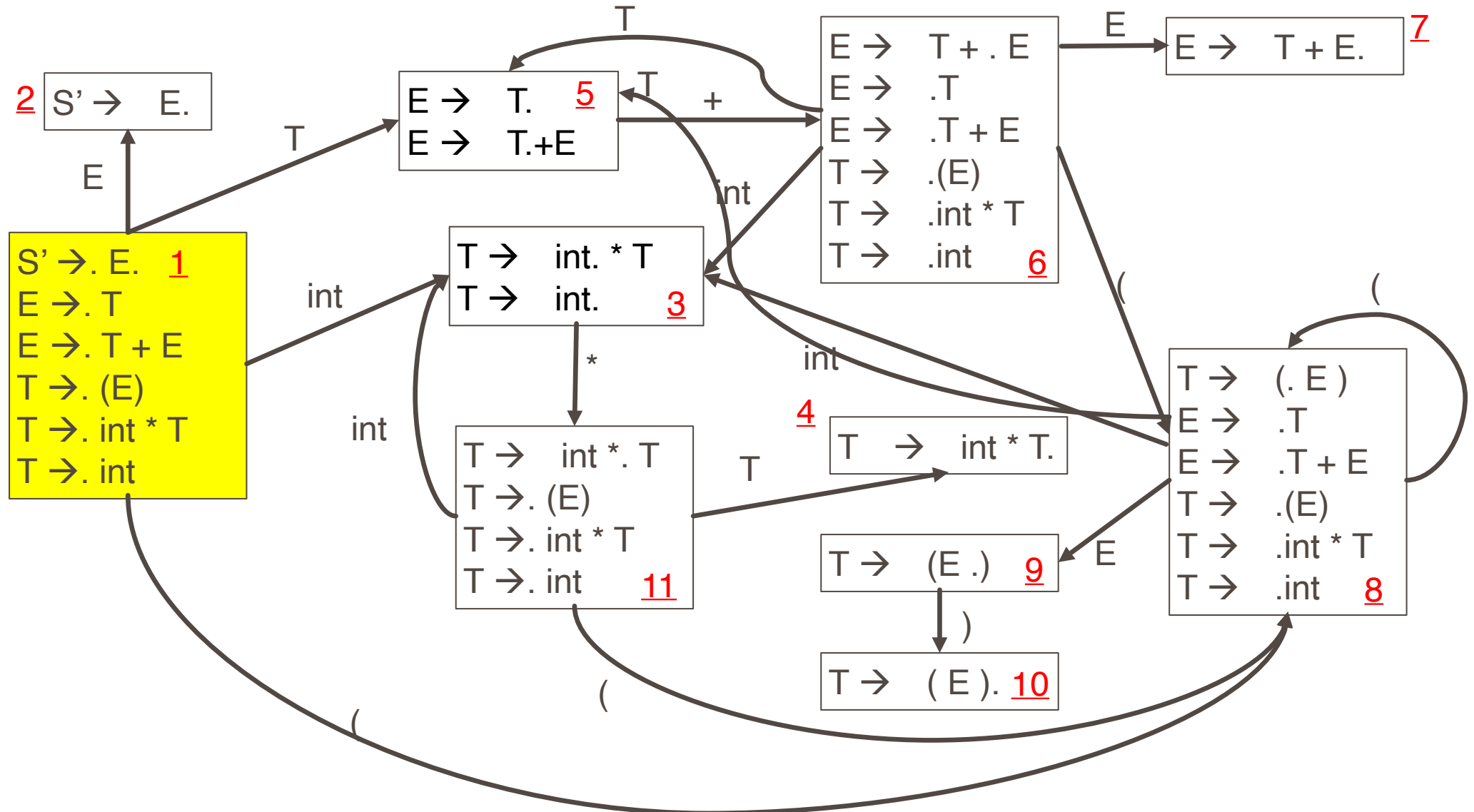
Parsing $\text{int} * \text{int} \$$

| Configuration | DFA Halt State | Action |
|--------------------------------|-------------------------|---------------------------------------|
| $\text{int} * \text{int} \$$ | 1 | shift |
| $\text{int} * \text{int} \$$ | 3 (* not in Follow(T)) | shift |
| $\text{int} * \text{int} \$$ | 11 | shift |
| $\text{int} * \text{int} \$$ | 3 (\$ Follow (T)) | reduce $T \rightarrow \text{int}$ |
| $\text{int} * T \$$ | 4 (\$ Follow (T)) | reduce $T \rightarrow \text{int} * T$ |

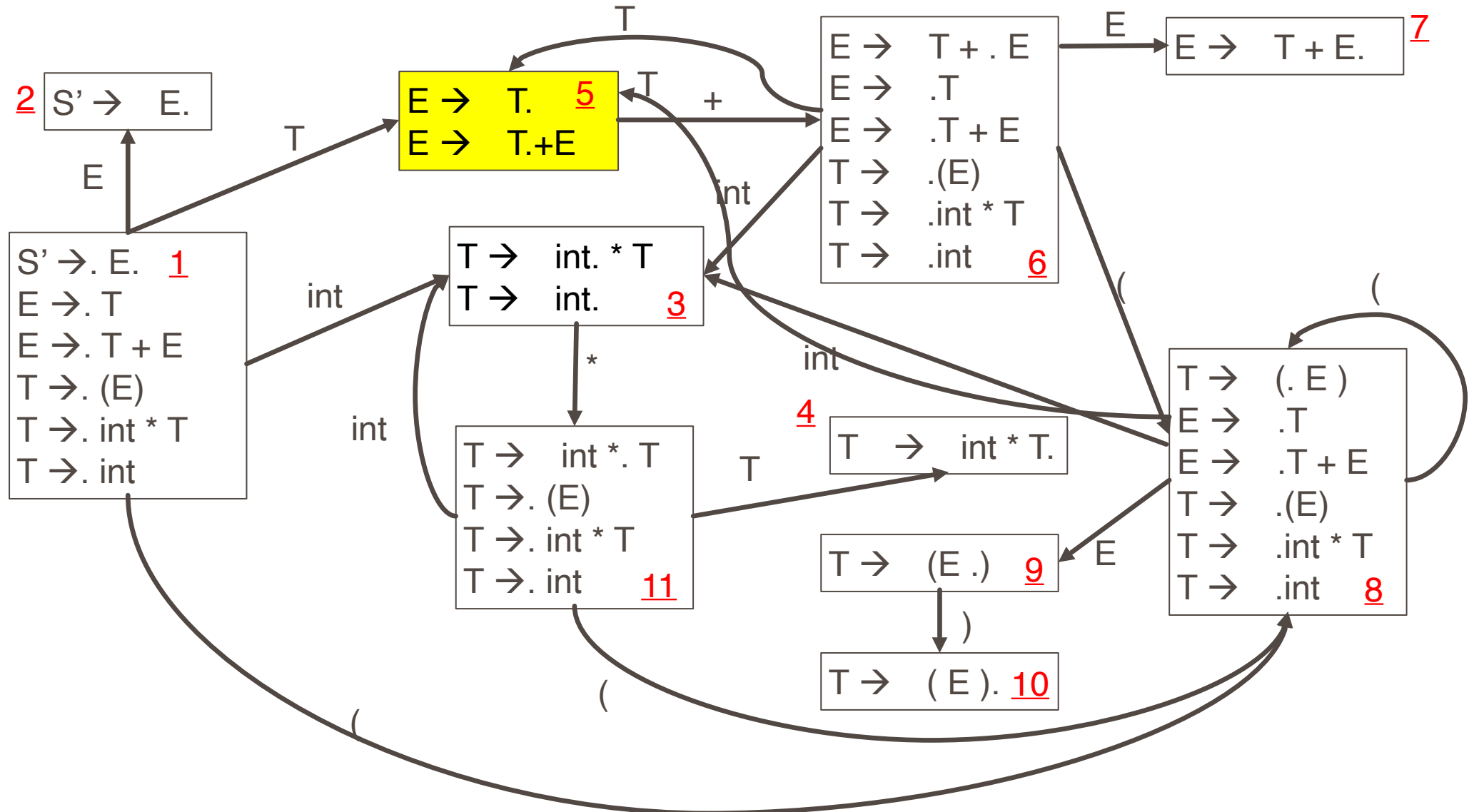
Parsing |int * int \$

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

Parsing T | \$



Parsing T | \$



Parsing $\text{int} * \text{int} \$$

| Configuration | DFA Halt State | Action |
|--------------------------------|-------------------------|---------------------------------------|
| $\text{int} * \text{int} \$$ | 1 | shift |
| $\text{int} * \text{int} \$$ | 3 (* not in Follow(T)) | shift |
| $\text{int} * \text{int} \$$ | 11 | shift |
| $\text{int} * \text{int} \$$ | 3 (\$ Follow (T)) | reduce $T \rightarrow \text{int}$ |
| $\text{int} * T \$$ | 4 (\$ Follow (T)) | reduce $T \rightarrow \text{int} * T$ |
| $T \$$ | 5 (\$ Follow (E)) | reduce $E \rightarrow T$ |

Parsing $\text{int} * \text{int} \$$

| Configuration | DFA Halt State | Action |
|--------------------------------|-------------------------|---------------------------------------|
| $\text{int} * \text{int} \$$ | 1 | shift |
| $\text{int} * \text{int} \$$ | 3 (* not in Follow(T)) | shift |
| $\text{int} * \text{int} \$$ | 11 | shift |
| $\text{int} * \text{int} \$$ | 3 (\$ Follow (T)) | reduce $T \rightarrow \text{int}$ |
| $\text{int} * T \$$ | 4 (\$ Follow (T)) | reduce $T \rightarrow \text{int} * T$ |
| $T \$$ | 5 (\$ Follow (E)) | reduce $E \rightarrow T$ |
| $E \$$ | | accept |

An Improvement

- Rerunning the automaton at each step is wasteful
 - Most of the work is repeated
- Change stack to contain pairs $\langle \text{Symbol, DFA State} \rangle$
 - DFA State is the state of the automaton on each prefix of the stack
- For a stack $\langle \text{sym}_1, \text{state}_1 \rangle \dots \langle \text{sym}_n, \text{state}_n \rangle$
 - state_n is the final state of the DFA on $\text{sym}_1 \dots \text{sym}_n$
- The bottom of the stack is $\langle \text{any, start} \rangle$ where
 - any is any dummy symbol
 - start is the start state of the DFA

Goto Table

- Define $\text{goto}[i, A] = j$ if $\text{state}_i \xrightarrow{A} \text{state}_j$
- goto is the transition function of the DFA

Refined Parser Moves

- Shift x
 - Push $\langle a, x \rangle$ on the stack
 - a is current input
 - x is a DFA state
- Reduce $X \rightarrow \alpha$
 - As before
- Accept
- Error

Action Table

- For each state s_i and terminal a
 - If s_i has item $X \rightarrow \alpha.a\beta$ and $\text{goto}[i,a] = j$ then $\text{action}[i,a] = \text{shift } j$
 - If s_i has item $X \rightarrow \alpha.$ and $a \in \text{Follow}(X)$ and $X \neq S'$ then $\text{action}[i,a] = \text{reduce } X \rightarrow \alpha$
 - If s_i has item $S' \rightarrow S.$ then $\text{action}[i,\$] = \text{accept}$
 - Otherwise, $\text{action}[i,a] = \text{error}$

SLR Parsing Algorithm

Let $I = w\$$ be initial input

Let $j = 0$

Let DFA state 1 have item $S' \rightarrow .S$

Let stack = $\langle \text{dummy}, 1 \rangle$

repeat

case action[top_state(stack), $I[j]$] of

shift k : push $\langle I[j++], k \rangle$

reduce $X \rightarrow A$:

pop $|A|$ pairs,

push $\langle X, \text{goto}[\text{top_state}(\text{stack}), X] \rangle$

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)