CS345H: Programming Languages

Lecture 6: Parsing Algorithms

Thomas Dillig

Outline

- Extend CFGs to build parse trees

Outline

- Extend CFGs to build parse trees
- We will build a parser that recognizes a CFG

Outline

- Extend CFGs to build parse trees
- We will build a parser that recognizes a CFG
- We will look at syntactic grammar restrictions that allows our algorithm to always succeed

Outline

- Extend CFGs to build parse trees
- We will build a parser that recognizes a CFG
- We will look at syntactic grammar restrictions that allows our algorithm to always succeed
- Error recovery

Extending CFGs for program parsing

- CFGs describe the structure of a program.

Extending CFGs for program parsing

- CFGs describe the structure of a program.
- But we also need this structure in form of a tree, not just a yes/no answer

Extending CFGs for program parsing

- CFGs describe the structure of a program.
- But we also need this structure in form of a tree, not just a yes/no answer
- Insight: We do not need all program structure, only the relevant part

Extending CFGs for program parsing

- CFGs describe the structure of a program.
- But we also need this structure in form of a tree, not just a yes/no answer
- Insight: We do not need all program structure, only the relevant part
- We call this an abstract syntax tree

ASTs

- Consider the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$

ASTs

- Consider the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- And the string `5 + (2 + 3)`

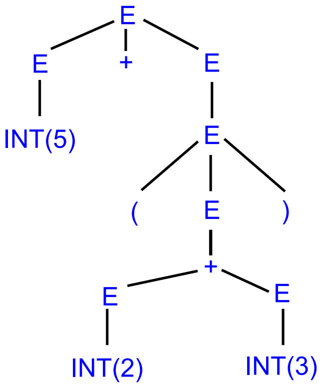
ASTs

- Consider the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- And the string `5 + (2 + 3)`
- After lexical analysis as string of tokens:
  `INT(5) '+' '(' INT(2) '+' INT(3) ')'`

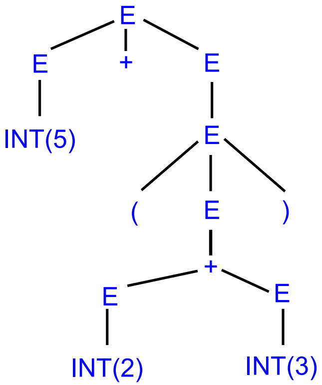
ASTs

- Consider the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- And the string `5 + (2 + 3)`
- After lexical analysis as string of tokens:
  `INT(5) '+' '(' INT(2) '+' INT(3) ')'`
- During parsing, we built a parse tree:

Example of Parse Tree

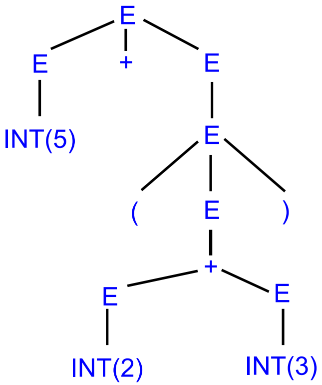


Example of Parse Tree



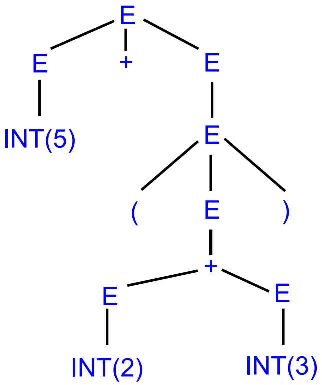
- Captures the nesting structure

Example of Parse Tree



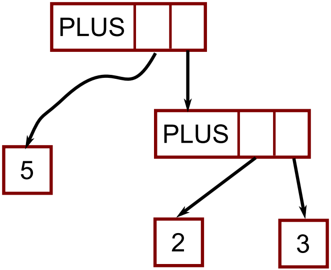
- Captures the nesting structure
- But too much information!

Example of Parse Tree

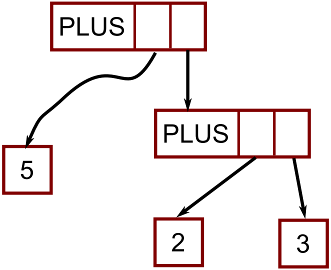


- Captures the nesting structure
- But too much information!
- Example: We do not care about the parentheses

Example of Abstract Syntax Tree

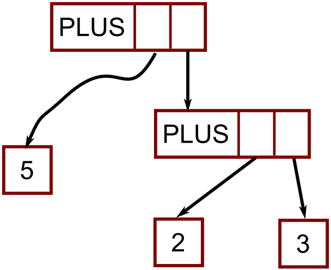


Example of Abstract Syntax Tree



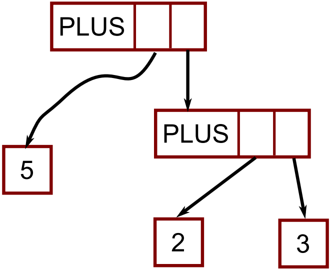
- Also captures the nesting structure

Example of Abstract Syntax Tree



- Also captures the nesting structure
- But abstracts from the concrete syntax

Example of Abstract Syntax Tree



- Also captures the nesting structure
- But abstracts from the concrete syntax
- More compact and easier to use

Semantic Actions to built the AST

- Each grammar symbol has one attribute

Semantic Actions to built the AST

- Each grammar symbol has one attribute
- For terminals (lexer tokens), the attribute is just the token

Semantic Actions to built the AST

- Each grammar symbol has one attribute
- For terminals (lexer tokens), the attribute is just the token
- Each production has a action computing its resulting attribute

Semantic Actions to built the AST

- Each grammar symbol has one attribute
- For terminals (lexer tokens), the attribute is just the token
- Each production has a action computing its resulting attribute
- Written as: $X \rightarrow Y_1 \ldots Y_n\{\text{action}\}$

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$

- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side

- Recall: The value of each terminal is just its token

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side
- Recall: The value of each terminal is just its token
- Assume value of symbol S is given by $S.val$

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side
- Recall: The value of each terminal is just its token
- Assume value of symbol S is given by $S.val$
- Grammar annotated with actions to compute the AST:

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side
- Recall: The value of each terminal is just its token
- Assume value of symbol S is given by $S.val$
- Grammar annotated with actions to compute the AST:

$$E \rightarrow \text{int} \ \{\texttt{E.val} = \texttt{int.val}\}$$

Semantic Actions: An Example

- Consider again the grammar: $E \to \text{int} \mid (E) \mid E + E$

- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side

- Recall: The value of each terminal is just its token

- Assume value of symbol S is given by $S.val$

- Grammar annotated with actions to compute the AST:

$$
\begin{array}{lcl}
E & \to & \text{int} \ \{\texttt{E.val} = \texttt{int.val}\} \\
E & \to & E_1 + E_2
\end{array}
$$

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side
- Recall: The value of each terminal is just its token
- Assume value of symbol S is given by $S.val$
- Grammar annotated with actions to compute the AST:

$$\begin{array}{lcl} E & \rightarrow & \text{int} \ \{\texttt{E.val} = \texttt{int.val}\} \\ E & \rightarrow & E_1 + E_2 \ \{\texttt{E.val} = \texttt{makeAstPlus}(\texttt{E}_1\texttt{.val}, \texttt{E}_2\texttt{.val})\} \end{array}$$

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \mathsf{int} \mid (E) \mid E + E$
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side
- Recall: The value of each terminal is just its token
- Assume value of symbol S is given by $S.val$
- Grammar annotated with actions to compute the AST:

$$
\begin{array}{lll}
E & \rightarrow & \mathsf{int} \;\; \{\texttt{E.val} = \texttt{int.val}\} \\
E & \rightarrow & E_1 + E_2 \;\; \{\texttt{E.val} = \texttt{makeAstPlus}(\texttt{E}_1\texttt{.val}, \texttt{E}_2\texttt{.val})\} \\
E & \rightarrow & (E')
\end{array}
$$

Semantic Actions: An Example

- Consider again the grammar: $E \rightarrow \text{int} \mid (E) \mid E + E$
- For each non-terminal on left-hand side, define its value in terms of symbols on right-hand side
- Recall: The value of each terminal is just its token
- Assume value of symbol S is given by $S.val$
- Grammar annotated with actions to compute the AST:

$$
\begin{array}{lll}
E & \rightarrow & \text{int} \;\; \{\texttt{E.val} = \texttt{int.val}\} \\
E & \rightarrow & E_1 + E_2 \;\; \{\texttt{E.val} = \texttt{makeAstPlus}(\texttt{E}_1\texttt{.val}, \texttt{E}_2\texttt{.val})\} \\
E & \rightarrow & (E') \;\; \{\texttt{E.val} = \texttt{E}'\texttt{.val}\}
\end{array}
$$

Semantic Actions to built the AST

- You can think of semantic actions as defining a system of equations that describe the values of the let-hand sides in terms of values on the right-hand side

Semantic Actions to built the AST

- You can think of semantic actions as defining a system of equations that describe the values of the let-hand sides in terms of values on the right-hand side

- Recall again

$$
\begin{array}{lcl}
E & \rightarrow & \textsf{int} \;\; \{\texttt{E.val} = \texttt{int.val}\} \\
E & \rightarrow & E_1 + E_2 \;\; \{\texttt{E.val} = \texttt{makeAstPlus}(\texttt{E}_1\texttt{.val}, \texttt{E}_2\texttt{.val})\} \\
E & \rightarrow & (E') \;\; \{\texttt{E.val} = \texttt{E}'\texttt{.val}\}
\end{array}
$$

Semantic Actions to built the AST

- You can think of semantic actions as defining a system of equations that describe the values of the let-hand sides in terms of values on the right-hand side
- Recall again

$$
\begin{array}{lll}
E & \rightarrow & \text{int} \;\; \{\texttt{E.val} = \texttt{int.val}\} \\
E & \rightarrow & E_1 + E_2 \;\; \{\texttt{E.val} = \texttt{makeAstPlus}(\texttt{E}_1\texttt{.val}, \texttt{E}_2\texttt{.val})\} \\
E & \rightarrow & (E') \;\; \{\texttt{E.val} = \texttt{E}'\texttt{.val}\}
\end{array}
$$

- Question: What order do we need to evaluate these equations to compute a solution?

Semantic Actions to built the AST

- You can think of semantic actions as defining a system of equations that describe the values of the let-hand sides in terms of values on the right-hand side

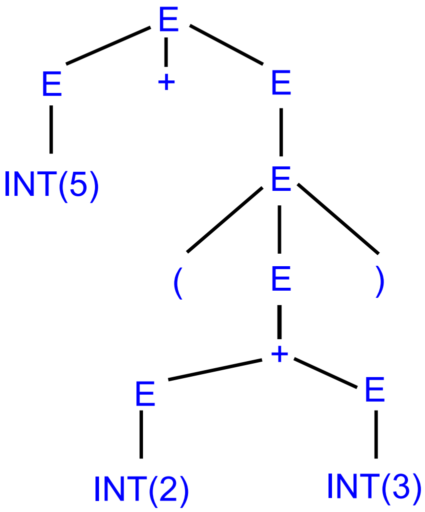
- Recall again

$$\begin{array}{lcl} E & \rightarrow & \text{int} \;\; \{\texttt{E.val} = \texttt{int.val}\} \\ E & \rightarrow & E_1 + E_2 \;\; \{\texttt{E.val} = \texttt{makeAstPlus}(\texttt{E}_1\texttt{.val}, \texttt{E}_2\texttt{.val})\} \\ E & \rightarrow & (E') \;\; \{\texttt{E.val} = \texttt{E}'\texttt{.val}\} \end{array}$$

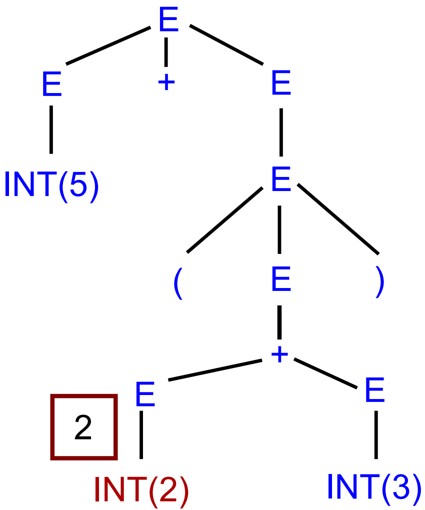
- Question: What order do we need to evaluate these equations to compute a solution?

- Answer: Bottom-up

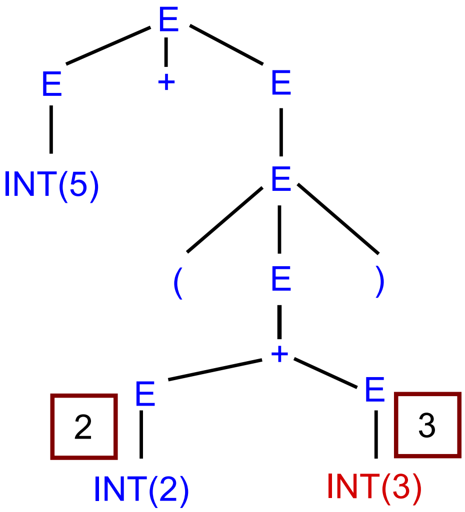
Semantic Actions: An Example cont.



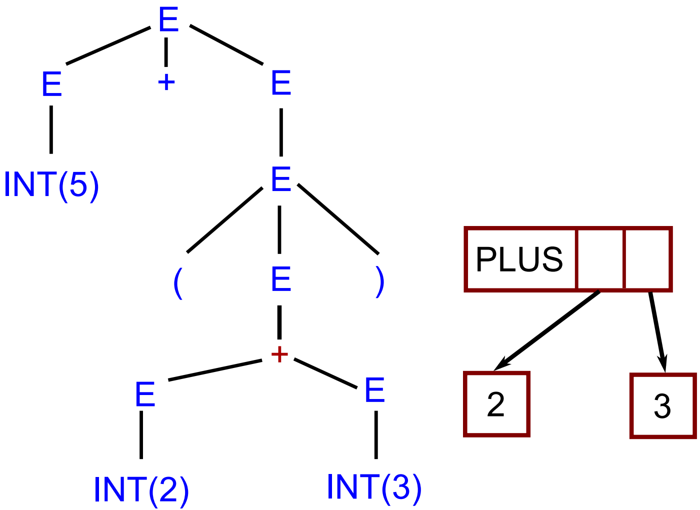
Semantic Actions: An Example cont.



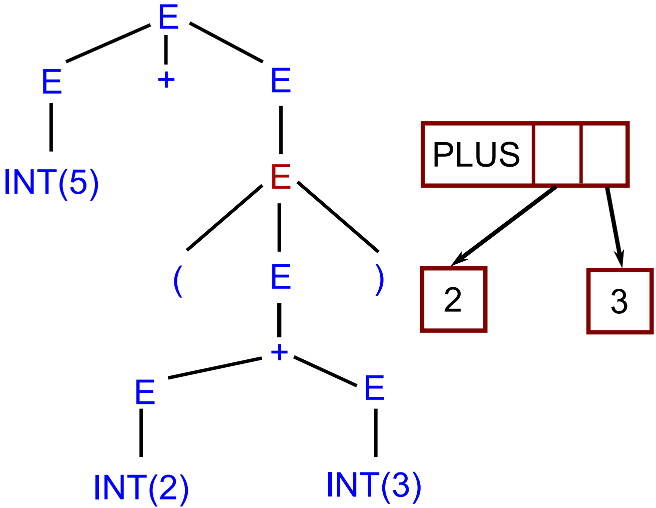
Semantic Actions: An Example cont.



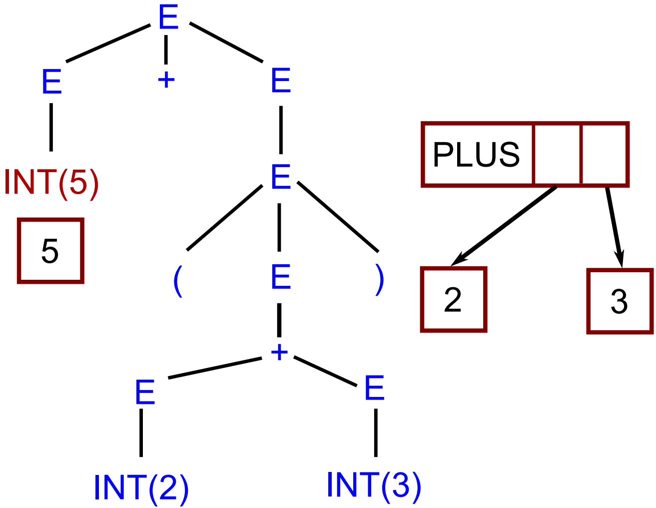
Semantic Actions: An Example cont.



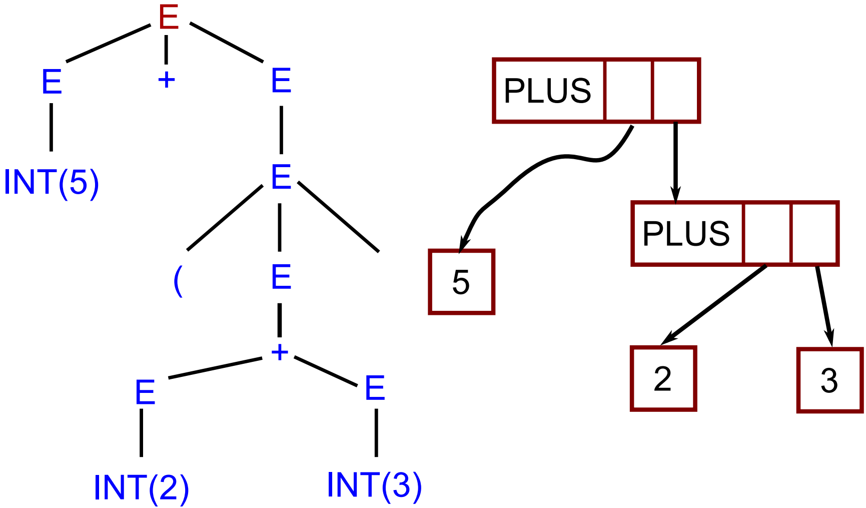
Semantic Actions: An Example cont.



Semantic Actions: An Example cont.



Semantic Actions: An Example cont.



Semantic Actions

- We have seen how we can use semantic actions to build the AST

Semantic Actions

- We have seen how we can use semantic actions to build the AST
- Next: How to build the parser that will allow us to execute these semantic actions

Parsing

- Consider the non-ambiguous grammar for simple arithmetic expressions:

$$\begin{array}{lcl} S & \rightarrow & E \mid E + S \\ E & \rightarrow & \textsf{int} \mid \textsf{int} * E \mid (S) \end{array}$$

Parsing

- Consider the non-ambiguous grammar for simple arithmetic expressions:

$$\begin{array}{rcl} S & \rightarrow & E \mid E + S \\ E & \rightarrow & \textsf{int} \mid \textsf{int} * E \mid (S) \end{array}$$

- Assume token stream is `( INT`$_5$` )`

Parsing

- Consider the non-ambiguous grammar for simple arithmetic expressions:

$$\begin{array}{rcl} S & \rightarrow & E \mid E + S \\ E & \rightarrow & \text{int} \mid \text{int} * E \mid (S) \end{array}$$

- Assume token stream is ($\texttt{INT}_5$)
- Idea: Start with start symbol S and try rules for S in order, backtrack if we made the wrong choice

Parsing

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

S

(INT5)

↑

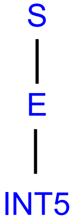
Parsing

$$
\begin{aligned}
S &\rightarrow E \mid E + S \\
E &\rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)
\end{aligned}
$$

(INT5)

Parsing

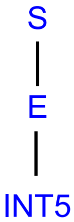
$$S \rightarrow E \mid E + S$$
$$E \rightarrow \textsf{int} \mid \textsf{int} * E \mid (S)$$



(INT5)

Parsing

$$
\begin{aligned}
S &\rightarrow E \mid E + S \\
E &\rightarrow \textsf{int} \mid \textsf{int} * E \mid (S)
\end{aligned}
$$



Mismatch! (is not INT
Backtrack...

(INT5)

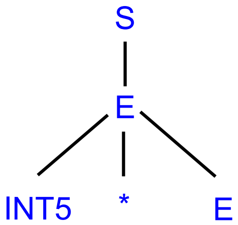
Parsing

$$
\begin{aligned}
S &\rightarrow E \mid E + S \\
E &\rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)
\end{aligned}
$$

(INT5)

Parsing

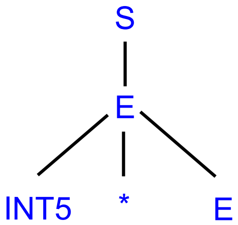
$$
\begin{array}{lcl}
S & \rightarrow & E \mid E + S \\
E & \rightarrow & \mathsf{int} \mid \mathsf{int} * E \mid (S)
\end{array}
$$



(INT5)

Parsing

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



(INT5)

Mismatch: (is not INT
Backtrack again...

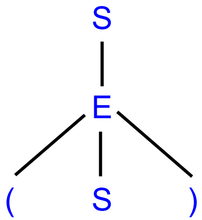
Parsing

$$S \rightarrow E \mid E + S$$
$$E \rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)$$

(INT5)

Parsing

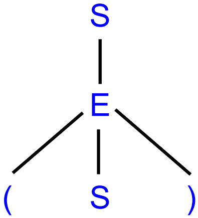
$$S \rightarrow E \mid E + S$$
$$E \rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)$$



(INT5)

Parsing

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

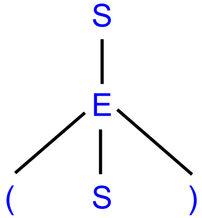


(INT5)

Match! Advance input

Parsing

$$S \rightarrow E \mid E + S$$
$$E \rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)$$

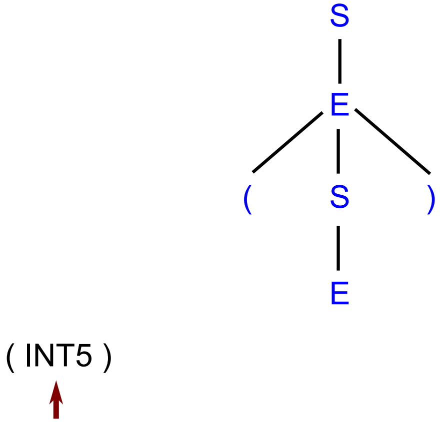


(INT5)

Match! Advance input

Parsing

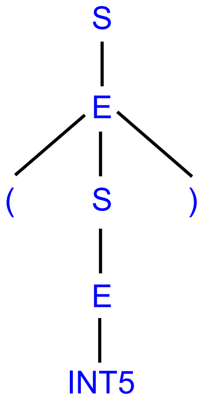
$$
\begin{aligned}
S &\rightarrow \textcolor{red}{E} \mid E + S \\
E &\rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)
\end{aligned}
$$



Parsing

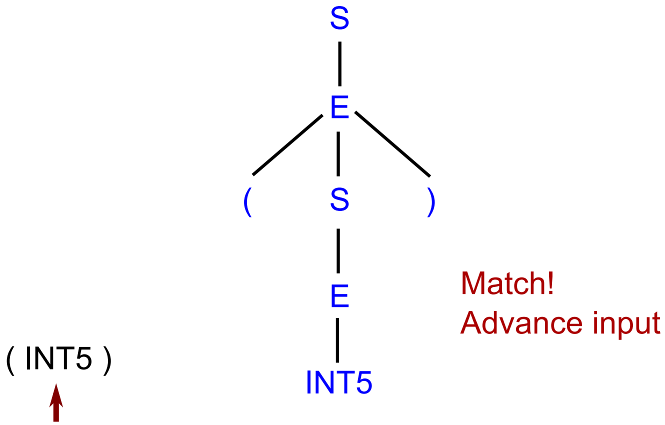
$$S \rightarrow E \mid E + S$$
$$E \rightarrow \textcolor{red}{\text{int}} \mid \text{int} * E \mid (S)$$

(INT5)



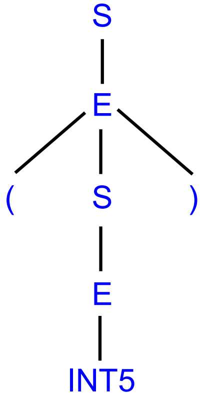
Parsing

$$
\begin{aligned}
S &\rightarrow E \mid E + S \\
E &\rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)
\end{aligned}
$$



Parsing

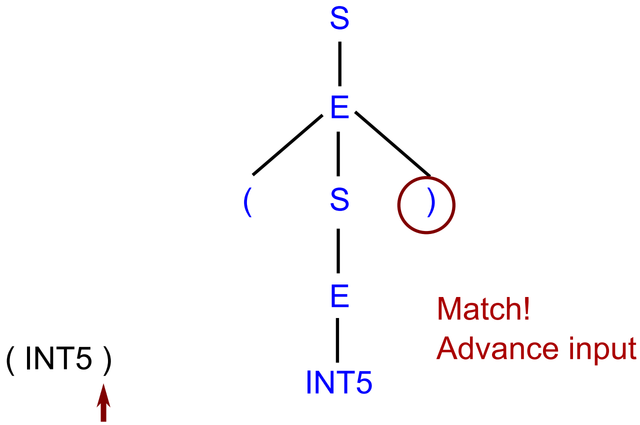
$$S \rightarrow E \mid E + S$$
$$E \rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)$$



(INT5)

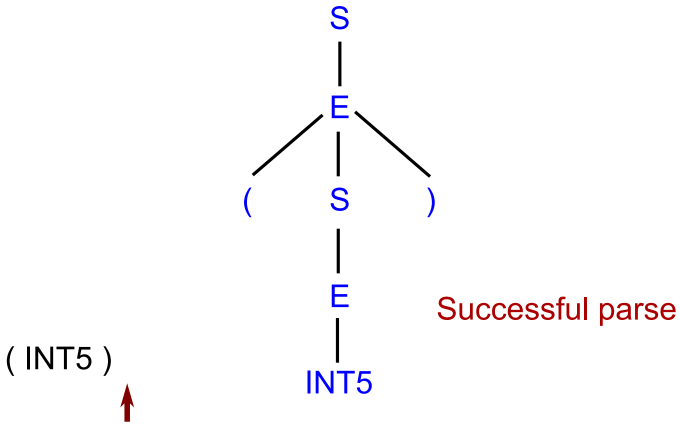
Parsing

$$S \rightarrow E \mid E + S$$
$$E \rightarrow \mathsf{int} \mid \mathsf{int} * E \mid (S)$$



Parsing

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



Recursive Descent Parsing

- This parsing strategy is called recursive-descent parsing

Recursive Descent Parsing

- This parsing strategy is called recursive-descent parsing
- It is easy to automate this strategy: For this assume:

Recursive Descent Parsing

- This parsing strategy is called recursive-descent parsing
- It is easy to automate this strategy: For this assume:
  - `TOKEN` is the type of tokens

Recursive Descent Parsing

- This parsing strategy is called recursive-descent parsing
- It is easy to automate this strategy: For this assume:
  - `TOKEN` is the type of tokens
  - `next` is global pointer to array of `TOKEN`'s

Recursive Descent Parsing 1

- Define boolean functions that check token stream for match and advance the `next` pointer

Recursive Descent Parsing 1

- Define boolean functions that check token stream for match and advance the `next` pointer
  - Generic function for each terminal:
    `bool term(TOKEN tok) { return token == *next++;}`

Recursive Descent Parsing 1

- Define boolean functions that check token stream for match and advance the `next` pointer
  - Generic function for each terminal:
    `bool term(TOKEN tok) { return token == *next++;}`
  - For the n'th production of a non-terminal S, we will define
    `bool S_n() { ...  }`

Recursive Descent Parsing 1

- Define boolean functions that check token stream for match and advance the `next` pointer
  - Generic function for each terminal:
    `bool term(TOKEN tok) { return token == *next++;}`
  - For the n'th production of a non-terminal S, we will define
    `bool S_n() { ... }`
  - To try all productions of a non-terminal S, we will define
    `bool S() { ... }`

Recursive Descent Parsing 2

- For production $S \rightarrow E$
  ```
  bool S_1() { return E(); }
  ```

Recursive Descent Parsing 2

- For production $S \rightarrow E$
```
bool S_1() { return E(); }
```

- For production $S \rightarrow E + S$
```
bool S_2() { return E() && term(PLUS) && S(); }
```

Recursive Descent Parsing 2

- For production $S \rightarrow E$

```
bool S_1() { return E(); }
```

- For production $S \rightarrow E + S$

```
bool S_2() { return E() && term(PLUS) && S(); }
```

- For all production S (with backtracking)

```
bool S() {
  TOKEN* save = next;
  if(S_1() == true) return true;
  next = save;
  return S_2(); }
```

Recursive Descent Parsing 2

- For production $S \rightarrow E$

```
bool S_1() { return E(); }
```

- For production $S \rightarrow E + S$

```
bool S_2() { return E() && term(PLUS) && S(); }
```

- For all production S (with backtracking)

```
bool S() {
  TOKEN* save = next;
  if(S_1() == true) return true;
  next = save;
  return S_2(); }
```

- Or, equivalently written as

```
bool S() {
  return ((next = save, S_1())
    || ((next = save, S_2()) }
```

Recursive Descent Parsing 3

- Now, the functions $E \rightarrow \text{int} \mid \text{int} * E \mid (S)$:

Recursive Descent Parsing 3

- Now, the functions $E \rightarrow \text{int} \mid \text{int} * E \mid (S)$:

```
bool E_1() { return TERM(INT); }
```

Recursive Descent Parsing 3

- Now, the functions $E \rightarrow \text{int} \mid \text{int} * E \mid (S)$:

```
bool E_1() { return TERM(INT); }
bool E_2() { return TERM(INT) &&
  term(TIMES) && T(); }
```

Recursive Descent Parsing 3

- Now, the functions $E \rightarrow \text{int} \mid \text{int} * E \mid (S)$:

```
bool E_1() { return TERM(INT); }
bool E_2() { return TERM(INT) &&
  term(TIMES) && T(); }
bool E_3() { return TERM(LPAREN) && S() &&
  TERM(RPAREN) }
```

Recursive Descent Parsing 3

- Now, the functions $E \rightarrow \text{int} \mid \text{int} * E \mid (S)$:

```
bool E_1() { return TERM(INT); }
bool E_2() { return TERM(INT) &&
  term(TIMES) && T(); }
bool E_3() { return TERM(LPAREN) && S() &&
  TERM(RPAREN) }
```

- For all productions in E, again with backtracking:

```
bool E() {
  TOKEN* save = next;
  return (next = save, E_1()) ||
  (next = save, E_2()) ||
  (next = save, E_3())
}
```

Complete Parser

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

bool S_1() { return E(); }
bool S_2() { return E() && term(PLUS) && S(); }
bool S() { return ((next = save, S_1())
    || ((next = save, S_2()) }

bool E_1() { return TERM(INT); }
bool E_2() { return TERM(INT) &&
  term(TIMES) && T(); }
bool E_3() { return TERM(LPAREN) && S() &&
  TERM(RPAREN) }
bool E() {
  TOKEN* save = next;
  return (next = save, E_1()) ||
  (next = save, E_2()) ||
  (next = save, E_3())
}
```

Recursive Descent Parsing 4

- To start this parser, initialize `next` to the first token and call `S()`

Recursive Descent Parsing 4

- To start this parser, initialize `next` to the first token and call `S()`
- This simulates the example parse and is easy to implement by hand

Are we done?

- Consider a production of the form

$$S \rightarrow Sa$$

Are we done?

- Consider a production of the form

$$S \to Sa$$

- We will generate the following functions using our scheme:

```
bool S_1() { return S() && term(a); }
bool S() { return S_1; }
```

Are we done?

- Consider a production of the form

$$S \to Sa$$

- We will generate the following functions using our scheme:

```
bool S_1() { return S() && term(a); }
bool S() { return S_1; }
```

- Here, `S()` goes into an infinite loop

Are we done?

- Consider a production of the form

$$S \rightarrow Sa$$

- We will generate the following functions using our scheme:

```
bool S_1() { return S() && term(a); }
bool S() { return S_1; }
```

- Here, `S()` goes into an infinite loop
- General Problem: If for some non-terminal S, it is possible to derive $S \rightarrow^* S\alpha$, recursive descent does not work

Are we done?

- Consider a production of the form

$$S \rightarrow Sa$$

- We will generate the following functions using our scheme:

```
bool S_1() { return S() && term(a); }
bool S() { return S_1; }
```

- Here, `S()` goes into an infinite loop
- General Problem: If for some non-terminal S, it is possible to derive $S \rightarrow^* S\alpha$, recursive descent does not work
- Such grammars are called left-recursive

Eliminating Left-Recursion

- Fortunately, it is always possible to eliminate left-recursion from grammars

Eliminating Left-Recursion

- Fortunately, it is always possible to eliminate left-recursion from grammars
- Example: Consider the grammar:

$$S \to S\alpha \mid \beta$$

Eliminating Left-Recursion

- Fortunately, it is always possible to eliminate left-recursion from grammars
- Example: Consider the grammar:

$$S \to S\alpha \mid \beta$$

- This grammar generates all strings starting with one β and followed by one or more αs

Eliminating Left-Recursion

- Fortunately, it is always possible to eliminate left-recursion from grammars
- Example: Consider the grammar:

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

- This grammar generates all strings starting with one β and followed by one or more αs
- Can rewrite using right-recursion:

Eliminating Left-Recursion

- Fortunately, it is always possible to eliminate left-recursion from grammars
- Example: Consider the grammar:

$$S \to S\alpha \mid \beta$$

- This grammar generates all strings starting with one β and followed by one or more αs
- Can rewrite using right-recursion:

$$\begin{array}{lcl} S & \to & \beta S' \\ S' & \to & \alpha S' \mid \varepsilon \end{array}$$

Eliminating Left-Recursion cont.

- In general:

$$S \to S\alpha_1 \mid \ldots \mid S\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

Eliminating Left-Recursion cont.

- In general:

$$S \to S\alpha_1 \mid \ldots \mid S\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- Insight: All strings derived from S start with one of $\beta_1, \ldots, \beta_m$ and continue with several instances of $\alpha_1, \ldots, \alpha_n$

Eliminating Left-Recursion cont.

- In general:

$$S \to S\alpha_1 \mid \ldots \mid S\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- Insight: 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:

Eliminating Left-Recursion cont.

- In general:

$$S \to S\alpha_1 \mid \ldots \mid S\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- Insight: 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:

$$\begin{array}{lcl} S & \to & \beta_1 S' \mid \ldots \mid \beta_m S' \\ S' & \to & \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \varepsilon \end{array}$$

Eliminating Left-Recursion cont.

- In general:

$$S \to S\alpha_1 \mid \ldots \mid S\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- Insight: 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:

$$\begin{array}{lcl} S & \to & \beta_1 S' \mid \ldots \mid \beta_m S' \\ S' & \to & \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \varepsilon \end{array}$$

- Easy to generalize this procedure slightly for non-direct left-recursion, such as

$$\begin{array}{lcl} S & \to & A\alpha \\ A & \to & S\beta \mid \varepsilon \end{array}$$

Recursive Descent Parsing

- Result: Recursive Descent parsing can parse any non-ambiguous grammar

Recursive Descent Parsing

- Result: Recursive Descent parsing can parse any non-ambiguous grammar
- Downside: Potentially expensive to backtrack

Recursive Descent Parsing

- Result: Recursive Descent parsing can parse any non-ambiguous grammar
- Downside: Potentially expensive to backtrack
- Left-recursion must be eliminated for recursive descent parsing to work, but this can be done automatically

Recursive Descent Parsing

- Result: Recursive Descent parsing can parse any non-ambiguous grammar
- Downside: Potentially expensive to backtrack
- Left-recursion must be eliminated for recursive descent parsing to work, but this can be done automatically
- In practice, you can often eliminate much backtracking by restricting the grammar

Other Parsing Algorithms

- Researchers works for 20 years to develop efficient paring algorithms, known as LL(1), LR(1), etc

Other Parsing Algorithms

- Researchers works for 20 years to develop efficient paring algorithms, known as LL(1), LR(1), etc
- All these algorithms avoid branching by some (bounded) token lookahead and only work on some grammars.

Other Parsing Algorithms

- Researchers works for 20 years to develop efficient paring algorithms, known as LL(1), LR(1), etc
- All these algorithms avoid branching by some (bounded) token lookahead and only work on some grammars.
- However: With computers getting faster every year, recursive descent parsing is very popular

Other Parsing Algorithms

- Researchers works for 20 years to develop efficient paring algorithms, known as LL(1), LR(1), etc
- All these algorithms avoid branching by some (bounded) token lookahead and only work on some grammars.
- However: With computers getting faster every year, recursive descent parsing is very popular
- Example: GCC and G++ both use a hand-written recursive descent parser

Other Parsing Algorithms

- Researchers works for 20 years to develop efficient paring algorithms, known as LL(1), LR(1), etc
- All these algorithms avoid branching by some (bounded) token lookahead and only work on some grammars.
- However: With computers getting faster every year, recursive descent parsing is very popular
- Example: GCC and G++ both use a hand-written recursive descent parser
- However, you will use the parser-generator `bison` for your homework which has some restrictions on your grammar. Read the posted manual!

Dealing with Errors

- Reality: Not every string of tokens can be parsed

Dealing with Errors

- Reality: Not every string of tokens can be parsed
- Example: `let let lambda x .   .`

Dealing with Errors

- Reality: Not every string of tokens can be parsed
- Example: `let let lambda x .   .`
- Option 1: Abort with an error message

Dealing with Errors

- Reality: Not every string of tokens can be parsed
- Example: `let let lambda x .   .`
- Option 1: Abort with an error message
- This is what you will do in PA2

Dealing with Errors

- Reality: Not every string of tokens can be parsed
- Example: `let let lambda x .   .`
- Option 1: Abort with an error message
- This is what you will do in PA2
- Often a reasonable choice

Dealing with Errors

- Reality: Not every string of tokens can be parsed
- Example: `let let lambda x .  .`
- Option 1: Abort with an error message
- This is what you will do in PA2
- Often a reasonable choice
- Option 2: Try to continue parsing after some tokens to report more errors

Dealing with Errors

- Reality: Not every string of tokens can be parsed
- Example: `let let lambda x .   .`
- Option 1: Abort with an error message
- This is what you will do in PA2
- Often a reasonable choice
- Option 2: Try to continue parsing after some tokens to report more errors
- Often results in garbage error reports

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses

- Typically, try inserting and deleting tokens until program compiles

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses
- Typically, try inserting and deleting tokens until program compiles
- Drawbacks:

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses
- Typically, try inserting and deleting tokens until program compiles
- Drawbacks:
  - Hard to implement

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses
- Typically, try inserting and deleting tokens until program compiles
- Drawbacks:
  - Hard to implement
  - Can be very slow

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses
- Typically, try inserting and deleting tokens until program compiles
- Drawbacks:
  - Hard to implement
  - Can be very slow
  - "Nearby" program is often not intended program

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses
- Typically, try inserting and deleting tokens until program compiles
- Drawbacks:
  - Hard to implement
  - Can be very slow
  - "Nearby" program is often not intended program
- This used to be a big research area, but today nobody cares

Dealing with Errors cont.

- Option 3: Try to find "nearby" program that parses
- Typically, try inserting and deleting tokens until program compiles
- Drawbacks:
  - Hard to implement
  - Can be very slow
  - "Nearby" program is often not intended program
- This used to be a big research area, but today nobody cares
- Question: Why is this the case?

Real Example

- Cornell developed a programming language called CUPL that parsed every program

Real Example

- Cornell developed a programming language called CUPL that parsed every program
- If you feed to following to the CUPL compiler:
  "To be, or not to be, that is the question:
  Whether 'tis Nobler in the mind to suffer
  The Slings and Arrows of outrageous Fortune,
  Or to take Arms against a Sea of troubles,
  ... "

Real Example

- Cornell developed a programming language called CUPL that parsed every program
- If you feed to following to the CUPL compiler:
  "To be, or not to be, that is the question:
  Whether 'tis Nobler in the mind to suffer
  The Slings and Arrows of outrageous Fortune,
  Or to take Arms against a Sea of troubles,
  ... "
- `Unknown construct "To be", did you mean BEGIN?`

Real Example

- Cornell developed a programming language called CUPL that parsed every program
- If you feed to following to the CUPL compiler:
  "To be, or not to be, that is the question:
  Whether 'tis Nobler in the mind to suffer
  The Slings and Arrows of outrageous Fortune,
  Or to take Arms against a Sea of troubles,
  ... "
- `Unknown construct "To be", did you mean BEGIN?`
- `Unknown construct ", or", did you mean "VAR or" ?`

Real Example

- Cornell developed a programming language called CUPL that parsed every program
- If you feed to following to the CUPL compiler:
  "To be, or not to be, that is the question:
  Whether 'tis Nobler in the mind to suffer
  The Slings and Arrows of outrageous Fortune,
  Or to take Arms against a Sea of troubles,
  ... "
- `Unknown construct "To be", did you mean BEGIN?`
- `Unknown construct ", or", did you mean "VAR or" ?`
- ...

Real Example

- Cornell developed a programming language called CUPL that parsed every program
- If you feed to following to the CUPL compiler:
  "To be, or not to be, that is the question:
  Whether 'tis Nobler in the mind to suffer
  The Slings and Arrows of outrageous Fortune,
  Or to take Arms against a Sea of troubles,
  ... "
- `Unknown construct "To be", did you mean BEGIN?`
- `Unknown construct ", or", did you mean "VAR or" ?`
- ...
- Final output:

Real Example

- Cornell developed a programming language called CUPL that parsed every program
- If you feed to following to the CUPL compiler:
  "To be, or not to be, that is the question:
  Whether 'tis Nobler in the mind to suffer
  The Slings and Arrows of outrageous Fortune,
  Or to take Arms against a Sea of troubles,
  ... "
- `Unknown construct "To be", did you mean BEGIN?`
- `Unknown construct ", or", did you mean "VAR or" ?`
- ...
- Final output: `BEGIN END`