Introduction to Parsing

CSCI 3136: Principles of Programming Languages

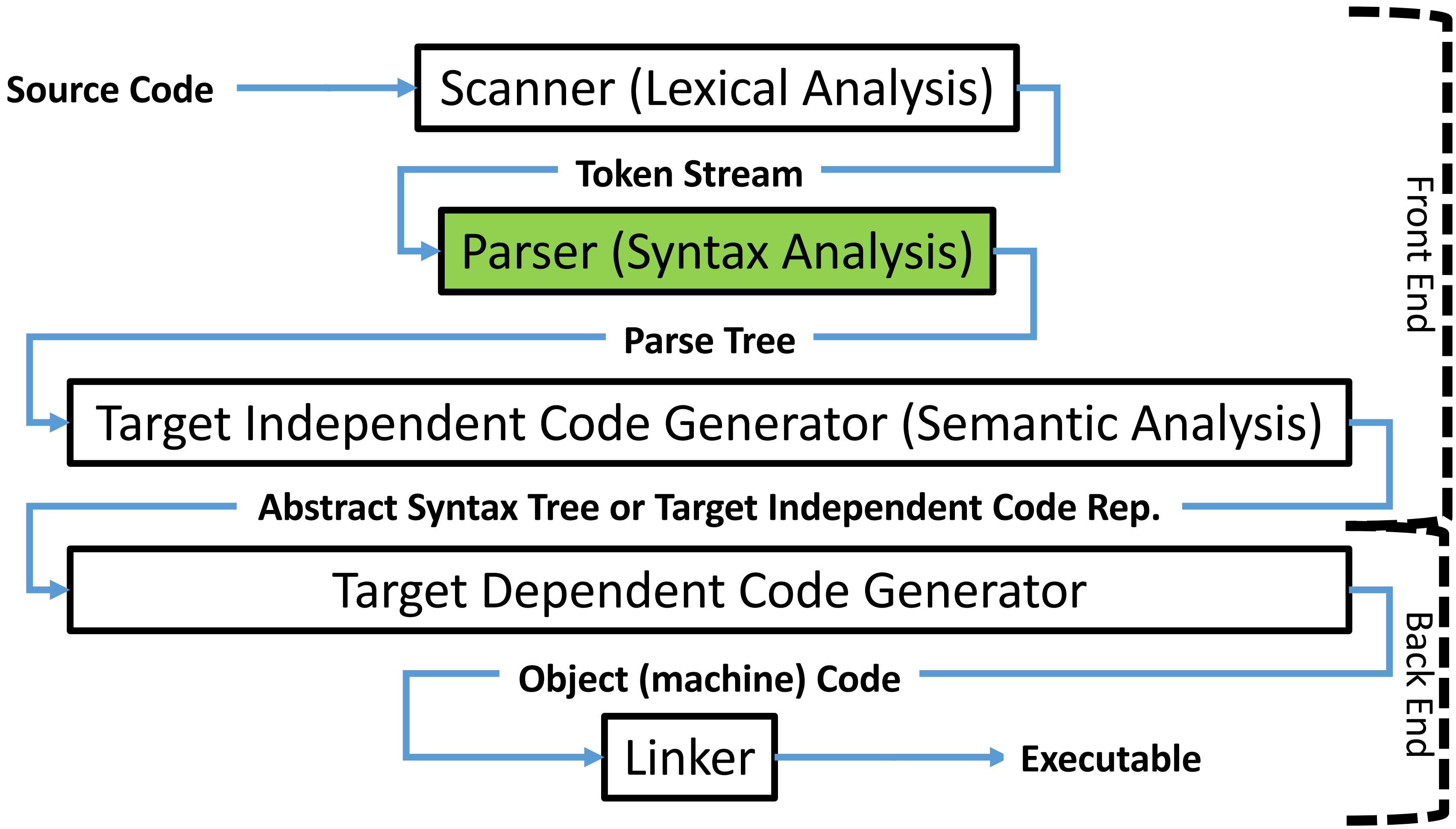
Agenda

- Announcements
  - Assignment 3 is out and due June 7
- Readings:
  - Today: 2.3.0, 2.3.1
  - Note: I recommend using alternative texts for this part of the course:
  - E..g, Hopcorft et al, “Introduction to Automata Theory”
- Lecture Contents
  - Introduction to Parsing

Why Do We Need a Parser?

- A scanner yields a stream of tokens
- Q: Is this sufficient to determine if the input is a valid program?
- A: No! Most programming languages are not regular!
  E.g. braces and brackets must match: $((1 + 3) * (3 + 2))$
- Scanners are useful for
  - Checking if program's tokens are correct
  - Providing higher level representation of programs
- Scanners cannot check if the syntax is correct
  - Analogy: Correctly spelled words do not make a correct sentence
- We need a different mechanism for checking syntax
- We need a parser

Recall: Phases of Compilation



Meet the Parser

- Parsing takes a stream of tokens
  - Checks whether the tokens represent a syntactically correct program
  - Creates a parse tree (a high level representation of the program)
- Question: How do we know what the correct syntax is?
- Answer: Based on the language specification
- Question: How do we specify the syntax
- Answer: By a grammar

Grammars

- Idea: Grammars specify the syntax of a language
- Example: English Sentences
  - *Sentence* → *Phrase Verb Phrase .*
  - *Phrase* → *Noun* | *Adjective Phrase*
  - *Adjective* → `big` | `small` | `green`
  - *Noun* → `boss` | `cheese`
  - *Verb* → `is` | `jumps` | `eats`

  Valid Sentences:
  - `Boss is big cheese.`
  - `Boss eats green cheese.`
  - `Green cheese jumps boss.`

  Not all valid sentences make sense!

Example: Arithmetic Expressions

Grammar

E → *E Op E*

E → **–** *E*

E → (*E*)

E → *Number*

E → *Identifier*

Op → +

Op → **–**

Op → /

Op → *

Valid Sentences

(1 + 2 - 3) * 4

- - 3

a + b

Typically programming languages are specified by Context Free Grammars (CFG)

Context Free Grammars (CFG)

A CFG G is a 4-tuple G = (V,Σ,P,S) where

- V is the set of non-terminals
  - Also known as “Variables”
  - Denoted by Capitalized letters/words
- Σ is the set of terminals
  - The text tokens returned by the scanner
- P is the set of productions
  - Of the form N → (Σ ∪ V)*, N ∈ V
  - Also known as “Rewriting Rules”
- S is the start symbol, S ∈ V

nonterminal or variable

terminal

$$\underbrace{E \rightarrow (\ E\ )}_{\text{production}}$$

A CFG Example: Expressions

- V = {E, Op}
- Σ = {identifier, number, (,), +, −, ∗, /}
- P={

  E → E Op E
  E → −E
  E → (E)
  E → number
  E → identifier
  Op → +
  Op → −
  Op → ∗
  Op → /

  }
- S = E

Notes on CFG Notation

- Note: Alternative productions can be merged using |
  - E.g., Op → + | - | * | /
- Several different notations are in use:
  - **Backus-Naur Form (BNF)** uses ::= instead of →
  - **Optional Components notation** N_{opt} means that N is optional in the production
  - **Regular Expressions in RHS notation** allows regular expressions of terminals and nonterminals
- Question: How do we use a grammar?
- We determine whether a program is *derivable* from the grammar

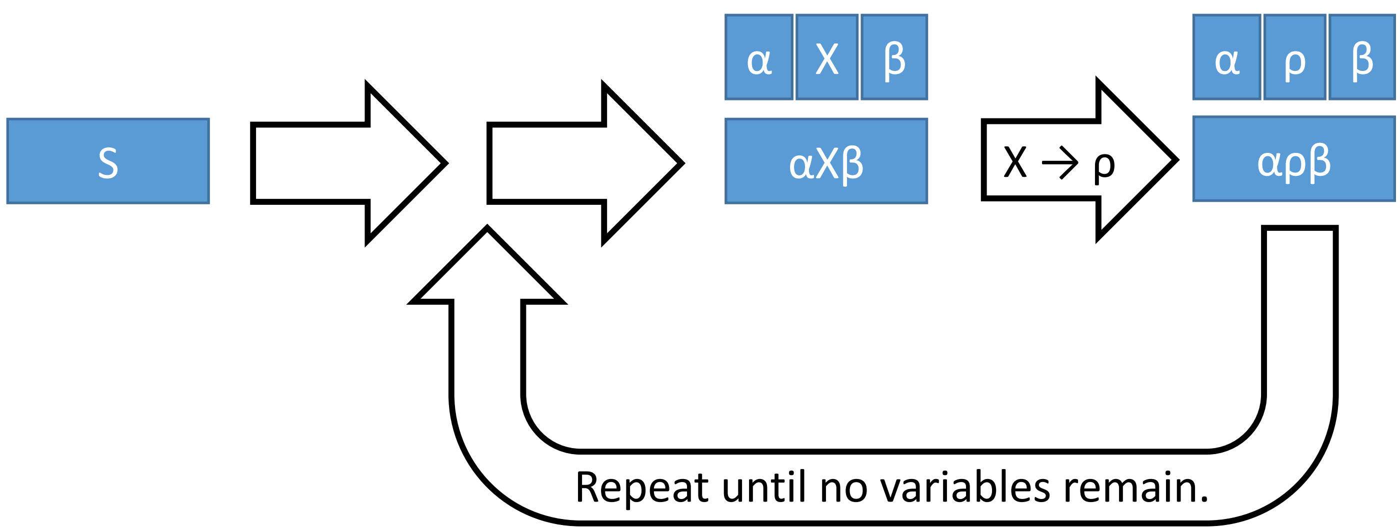
Derivations

- A derivation is a sequence of rewriting operations that starts with the string $\sigma = S$ and then repeats the following until σ contains only terminals:
  - Select a non-terminal in $X \in V$, such that $\sigma = \alpha X \beta$

    where $\alpha, \beta \in (V \cup \Sigma)*$
  - Select a production in $(X \rightarrow \rho) \in P$,
  - Replace X with ρ in the partial derivation σ

    I.e., $\sigma = \alpha \rho \beta$
- Eventually, σ will consist of only terminals, meaning the derivation is complete.

Derivations in a Nutshell



Derivation Example of an Expression

Derive (42 + 13) * 11

$\sigma = \mathbf{E}$

$\Rightarrow \mathbf{E}\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ \mathbf{E}\ )\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ \mathbf{E}\ \mathrm{Op}\ \mathrm{E}\ )\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ 42\ \mathbf{Op}\ \mathrm{E}\ )\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ 42 + \mathbf{E}\ )\ \mathrm{Op}\ \mathrm{E}$

$\Rightarrow (\ 42 + 13\ )\ \mathbf{Op}\ \mathrm{E}$

$\Rightarrow (\ 42 + 13\ ) * \mathbf{E}$

$\Rightarrow (\ 42 + 13\ ) * 11$

Grammar

1. *E → E Op E*
2. *E →* **–** *E*
3. *E → (E)*
4. *E → Number*
5. *E → Identifier*
6. *Op → +*
7. *Op →* **–**
8. *Op → /*
9. *Op → **

Definitions

- Definition: We write $S \Rightarrow^* \sigma$ if there exists a derivation

  $S \Rightarrow \sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots \Rightarrow \sigma$
- Definition: Every grammar G defines a language:

  $L(G)=\{\sigma \in \Sigma^* \mid S \Rightarrow^* \sigma\}$
- Definition: If G is a context-free grammar then L(G) is a context-free language.
- Example: What is the language defined by $G = (V, \Sigma, P, S)$
  - $V = \{S\}$
  - $\Sigma = \{0,1,\varepsilon\}$
  - P= {

    $S \rightarrow \varepsilon$

    $S \rightarrow 0\ S\ 1$

    }
  - S = S

The language $L(G) = \{0^n 1^n \mid n \geq 0\}$

Example 2

- What is the language defined by G = (V, Σ, P, S)
  - V = {S}
  - Σ = {0,1,ε}
  - P = {

    $S \rightarrow \varepsilon$

    $S \rightarrow 0S0$

    $S \rightarrow 1S1$

    }
  - S = S

The language $L(G) = \{\sigma\sigma^r \mid \sigma \in \Sigma*\}$
Note: σ^r means reverse of σ

- Observations:
  - These languages are nonregular
  - All regular languages are also context-free languages
  - There are more context-free than regular languages
- Q: How does we represent a derivation?

Parse Trees

- A program is syntactically correct if it can be derived from the grammar of the language it is written in.
- To analyze the program we need a better representation of it.
  - I.e., tokens are the input to the parser
- So, each derivation can be represented by a parse tree.

Structure of Parse Trees

- Root: S, the start nonterminal
- Internal nodes: nonterminals
- Leaf nodes: terminals (called the *yield* of the tree)
- Edge(X,w) : $X \in V$, $w \in \alpha$, where $(X \to \alpha) \in P$.

Parse Tree Example of an Expression

$\sigma = \mathbf{E}$

$\Rightarrow \mathbf{E}$ Op E

$\Rightarrow$ ($\mathbf{E}$) Op E

$\Rightarrow$ ($\mathbf{E}$ Op E) Op E

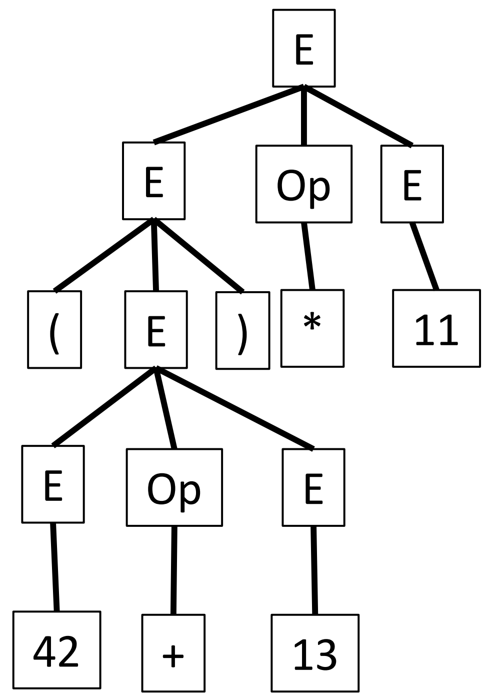
$\Rightarrow$ (42 **Op** E) Op E

$\Rightarrow$ (42 + $\mathbf{E}$) Op E

$\Rightarrow$ (42 + 13) **Op** E

$\Rightarrow$ (42 + 13) $*$ $\mathbf{E}$

$\Rightarrow$ (42 + 13) $*$ 11



Another Example: 1 + 2 * 3

This is ambiguous!

Grammar
1. E → E Op E
2. E → –E
3. E → (E)
4. E → number
5. E → identifier
6. Op → +
7. Op → –
8. Op → ∗
9. Op → /

σ = **E**

⇒ **E** Op E

⇒ **E** Op E Op E

⇒ 1 **Op** E Op E

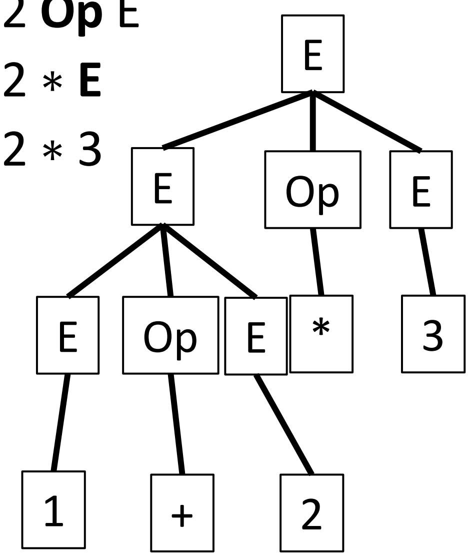
⇒ 1 + **E** Op E

⇒ 1 + 2 **Op** E

⇒ 1 + 2 ∗ **E**

⇒ 1 + 2 ∗ 3

(1 + 2) * 3



σ = **E**

⇒ **E** Op E

⇒ 1 **Op** E

⇒ 1 + **E**

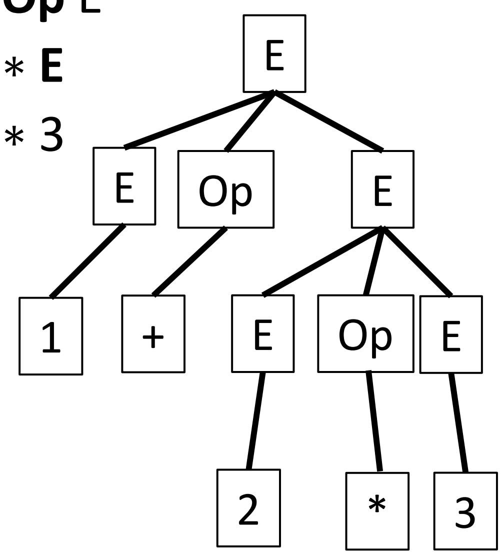
⇒ 1 + **E** Op E

⇒ 1 + 2 **Op** E

⇒ 1 + 2 ∗ **E**

⇒ 1 + 2 ∗ 3

1 + (2 * 3)



Ambiguity

- Observations:
  - There are infinitely many grammars to specify the same language
  - There may be multiple parse trees for the same sentence!
- Definition: If multiple parse trees can be generated by G for the same sentence, then G is *ambiguous*.
- Definition: If L does not have an unambiguous grammar, then L is *inherently ambiguous*
  - Usually not the case for programming languages!

An Unambiguous Expression Grammar

Grammar

1. E → T
2. E → E + T
3. E → E − T
4. T → F
5. T → T ∗ F
6. T → T / F
7. F → number
8. F → identifier
9. F → (E)

- Try deriving 1 + 2 * 3

Derivation Order

- Derivation orders refer to the order in which variables are replaced in the current partial derivation.
- The two most common ones are:
  - **Leftmost derivation** replaces the leftmost variable in each step
  - **Rightmost derivation** replaces the rightmost variable in each step

Leftmost Derivation Example 1+2*3

$E \Rightarrow \mathbf{E} + T$
$\Rightarrow \mathbf{T} + T$
$\Rightarrow \mathbf{F} + T$
$\Rightarrow 1 + \mathbf{T}$
$\Rightarrow 1 + \mathbf{T} * F$
$\Rightarrow 1 + \mathbf{F} * F$
$\Rightarrow 1 + 2 * \mathbf{F}$
$\Rightarrow 1 + 2 * 3$

Grammar

1. E → T
2. E → E + T
3. E → E − T
4. T → F
5. T → T * F
6. T → T / F
7. F → number
8. F → identifier
9. F → (E)

Rightmost Derivation Example 1+2*3

$E \Rightarrow E + \mathbf{T}$
$\Rightarrow E + T * \mathbf{F}$
$\Rightarrow E + \mathbf{T} * 3$
$\Rightarrow E + \mathbf{F} * 3$
$\Rightarrow \mathbf{E} + 2 * 3$
$\Rightarrow \mathbf{T} + 2 * 3$
$\Rightarrow \mathbf{F} + 2 * 3$
$\Rightarrow 1 + 2 * 3$

Grammar

1. E → T
2. E → E + T
3. E → E − T
4. T → F
5. T → T ∗ F
6. T → T / F
7. F → number
8. F → identifier
9. F → (E)

Where Are We?

- CFGs are used to specify programming language syntax
- Parsing finds the parse tree of the program (token stream)
- CFGs for programming languages must unambiguously capture the program structure.
- Parsers must be efficient:
  - A parser can be generated from a CFG that runs in $O(n^3)$ time
  - We prefer (require) linear time.
- How do we get this?

Regular Grammars: A Brief Aside

- A CFG is *right-linear* if all productions are of the form
  - $A \rightarrow \sigma B$, $\sigma \in \Sigma^*$, $B \in V$
  - $A \rightarrow \sigma$, $\sigma \in \Sigma^*$
- A CFG is *left-linear* if all productions are of the form
  - $A \rightarrow B\sigma$, $\sigma \in \Sigma^*$, $B \in V$
  - $A \rightarrow \sigma$, $\sigma \in \Sigma^*$
- A CFG is regular if it is right-linear or left-linear
- Regular grammars specify exactly the set of regular languages
- Regular grammars are too weak to specify most programming languages
- But, parsers generated from them run in linear time!
  - Why?
  - Are there more complex grammars for which linear time parsers exist?

LL and LR Grammars

Two kinds of unambiguous grammars that can be parsed efficiently

- LL(k) grammars
  - Are scanned Left-to-right and generate a Leftmost derivation
  - If the first letter in the current sentential form is a variable, k tokens look-ahead in the input suffice to decide which production to use to expand it.
- LR(k) grammars
  - Are scanned Left-to-right and generate a Rightmost derivation
  - The next k tokens in the input suffice to choose the next step the parser should perform.
- The syntax of almost every programming language can be described by LL(1) or LR(1) grammars!
  - How? Why?

S-Grammars

- First let's consider a very simple grammar
- An *S-grammar* or *simple grammar* is a special case of an LL(1)-grammar
- A CFG is an S-grammar if
  - Every production starts with a terminal
  - Productions for the same LHS start with different terminals

  E.g., If G contains A→aA and A→a then G is not simple!
- Idea: When using S-Grammars, selecting which rule to apply is easy.

Example: LL(1) Parsing (top-down) S-Grammar for Polish Notation

1. $S \rightarrow + SS$
2. $S \rightarrow - SS$
3. $S \rightarrow * SS$
4. $S \rightarrow / SS$
5. $S \rightarrow \text{neg } S$
6. $S \rightarrow \text{integer}$

Expression:

`– * + 1 2 3 4`

Is interpreted as:

`(1 + 2) * 3 – 4`

- How do we derive

  $\boxed{-}\,\boxed{*}\,\boxed{+}\,\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}$

$$
\begin{aligned}
S &\Rightarrow - \mathbf{S}\, S \\
&\Rightarrow - * \mathbf{S}\, S\, S \\
&\Rightarrow - * + \mathbf{S}\, S\, S\, S \\
&\Rightarrow - * + 1\, \mathbf{S}\, S\, S \\
&\Rightarrow - * + 1\, 2\, \mathbf{S}\, S \\
&\Rightarrow - * + 1\, 2\, 3\, \mathbf{S} \\
&\Rightarrow - * + 1\, 2\, 3\, 4
\end{aligned}
$$

- This is an example of LL(1) parsing
- How does a parser do this?

LL(1) Parsing of S-Grammars

Use a stack to store the
current sentential form
push(S) # push start variable
Loop until no more tokens:

```
t = next_token()
x = pop()
if x == t:
    continue
elseif x ∈ V:
    select production x → t α
    add children t α to node x
    push(α)
else:
    error
```

Grammar

1. S → + SS
2. S → − SS
3. S → ∗ SS
4. S → / SS
5. S → neg S
6. S → integer

Parse Expression:

`- * + 1 2 3 4`

This takes linear time!

Example: LR(1) Parsing (bottom-up) S-Grammar for Polish Notation

1. $S \rightarrow + SS$
2. $S \rightarrow - SS$
3. $S \rightarrow * SS$
4. $S \rightarrow / SS$
5. $S \rightarrow$ neg S
6. $S \rightarrow$ integer

Expression:

– * + 1 2 3 4

Is interpreted as:

(1 + 2) * 3 – 4

- How do we derive
  $\boxed{-}\boxed{*}\boxed{+}\boxed{1}\boxed{2}\boxed{3}\boxed{4}$

$\Leftarrow - * + \mathbf{S}\ 2\ 3\ 4$ [use rule 6]

$\Leftarrow - * + S\ \mathbf{S}\ 3\ 4$ [use rule 6]

$\Leftarrow - * \mathbf{S}\ 3\ 4$ [use rule 1]

$\Leftarrow - * S\ \mathbf{S}\ 4$ [use rule 6]

$\Leftarrow - \mathbf{S}\ 4$ [use rule 3]

$\Leftarrow - S\ \mathbf{S}$ [use rule 6]

$\Leftarrow \mathbf{S}$ [use rule 2]

- This is an example of LR(1) parsing
- How does a parser do this?

LR(1) Parsing of S-Grammars

Use a stack to store the
what has been seen so far
push(next_token()) # init stack
Loop until no more tokens:
  if $\exists(P \rightarrow \alpha)$ such that $\alpha \in$ Stack
    # reduce operation
    pop(α)
    push(P)
    add children α to node P
  else:
    # shift operation
    push(next_token())

Grammar

1. S → + SS
2. S → − SS
3. S → ∗ SS
4. S → / SS
5. S → neg S
6. S → integer

Parse Expression:

`- * + 1 2 3 4`

This takes linear time!

Building Parsers

- We now have some intuition about parsing algorithms
- But …
  - The above algorithms are for S-Grammars (too simple)
  - Want to generate parser given a grammar
- So ...
  - Assume that we will be using more complex grammars
  - How do we generate the parser?

Building an LL(1) Parser

- Basic Challenge: Given current token, which production does the parser select if next item in sentential form is a nonterminal

  E.g., if S is on the stack and input is +, then parser must select production $S \rightarrow +SS$

- In general: for input **a**, sentential form A . . ., either
  - $A \Rightarrow \alpha \Rightarrow^* \mathbf{a}\beta$
  - $A \Rightarrow \alpha \Rightarrow^* \varepsilon$ and derivation of A is succeeded by **a**.

- Intuitively, **a** is in the *predictor set* of $A \rightarrow \alpha$

  if $A\beta \Rightarrow \alpha\beta \Rightarrow^* a\gamma$, for $\beta,\gamma \in \Sigma^*$

  I.e., the parser selects $A \rightarrow \alpha$ if **a** is the input and in the *predictor set* of $A \rightarrow \alpha$

LL(1) Grammars

- **Definition**: A grammar is LL(1) if the predictor sets of all productions with the same LHS are disjoint.
- E.g. S-Grammars are LL(1)

  Grammar

  1. $S \rightarrow + SS$
  2. $S \rightarrow - SS$
  3. $S \rightarrow * SS$
  4. $S \rightarrow / SS$
  5. $S \rightarrow$ neg S
  6. $S \rightarrow$ integer

Production	Predictor Set
S → + S S	{+}
S → - S S	{-}
S → * S S	{*}
S → / S S	{/}
S → neg S	{neg}
S → integer	{integer}