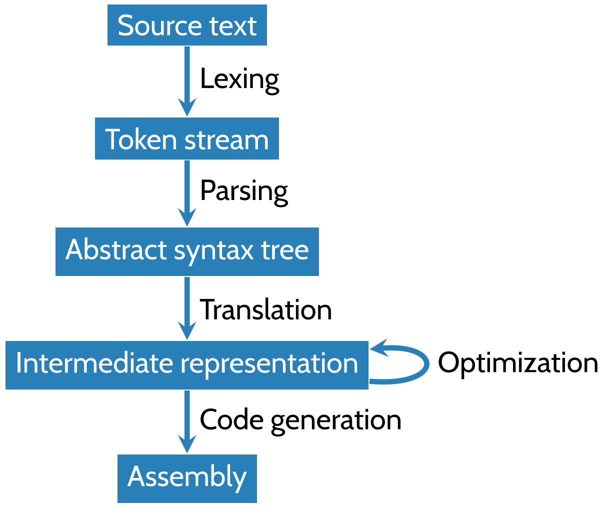
COS320: Compiling Techniques

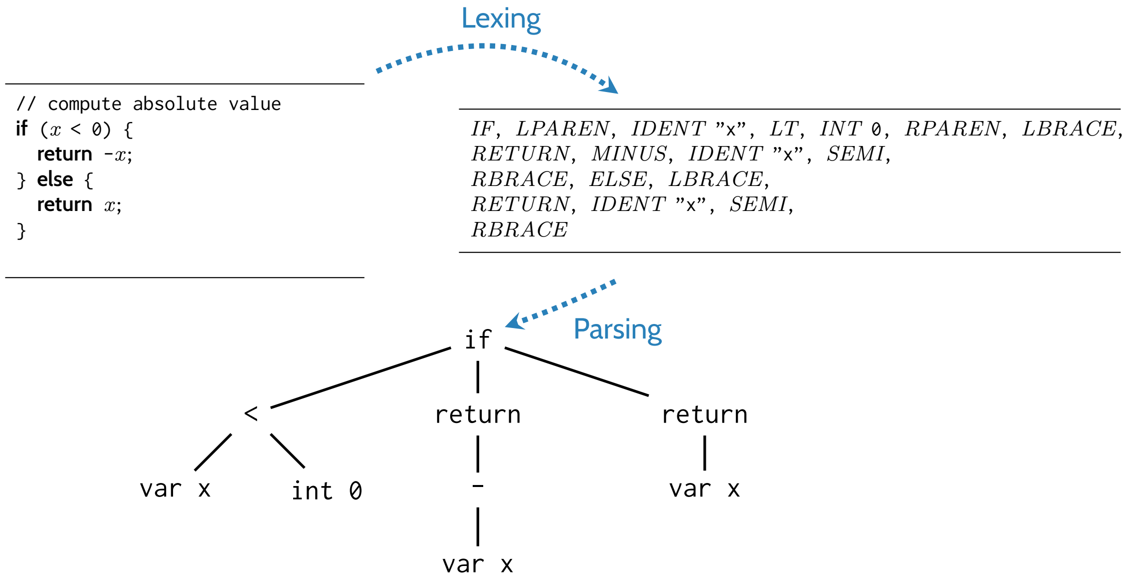
Zak Kincaid

January 29, 2026

Parsing I: Context-free languages

Compiler phases (simplified)



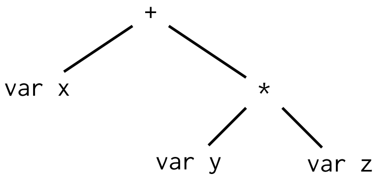


```
// compute absolute value
if (x < 0) {
  return -x;
} else {
  return x;
}
```

IF, *LPAREN*, *IDENT* "x", *LT*, *INT* 0, *RPAREN*, *LBRACE*,
RETURN, *MINUS*, *IDENT* "x", *SEMI*,
RBRACE, *ELSE*, *LBRACE*,
RETURN, *IDENT* "x", *SEMI*,
RBRACE

- The *parsing* phase of a compiler takes in a stream of tokens (produced by a lexer), and builds an *abstract syntax tree* (AST).
    - Parser is responsible for reporting syntax errors if the token stream cannot be parsed
    - Variable scoping, type checking, … handled later (*semantic analysis*)

- The *parsing* phase of a compiler takes in a stream of tokens (produced by a lexer), and builds an *abstract syntax tree* (AST).
    - Parser is responsible for reporting syntax errors if the token stream cannot be parsed
    - Variable scoping, type checking, … handled later (*semantic analysis*)
- An *abstract syntax tree* is a tree that represents the syntactic structure of the source code
    - "Abstract" in the sense that it omits details of the concrete syntax
        - semi-colons, parens, braces, whitespace, comments, …
    - E.g., the following have the same abstract syntax tree:
        - `x + y * z`
        - `x + (y * z)`
        - `(x) + (y * z)`
        - `((x) + (y * z))`



Implementing a parser

- Option 1: By-hand (recursive descent)
  - Clang, gcc (since 3.4)
  - Libraries can make this easier (e.g., parser combinators – `parsec`)

Implementing a parser

- Option 1: By-hand (recursive descent)
  - Clang, gcc (since 3.4)
  - Libraries can make this easier (e.g., parser combinators – `parsec`)
- Option 2: Use a parser generator
  - Much easier to get right (“specification is the implementation”)
    - Parser generator warns of ambiguities, ill-formed grammars, etc.
  - gcc (before 3.4), Glasgow Haskell Compiler, OCaml compiler
  - Parser generators: Yacc, Bison, ANTLR, **menhir**

Defining syntax

- Recall:
  - An *alphabet* Σ is a finite set of symbols (e.g., $\{0, 1\}$, ASCII, unicode).
  - A *word* (or *string*) over Σ is a sequence of symbols in Σ
  - A *language* over Σ is a set of words over Σ
- The set of syntactically valid programs in a programming language is a language
  - Conceptually: alphabet is ASCII or Unicode
  - In practice: (often) over token types
    - Lexer gives us a higher-level view of source text that makes it easier to work with
- This language is typically specified by a *context-free grammar*

```
<expr> ::=<int>
         | <var>
         | <expr>+<expr>
         | <expr>*<expr>
         | (<expr>)
```

- Well-formed expressions (character-level):
  `3+2*x`,
  `(x*100) + (y*10) + z`, ...
- Well-formed expressions (token-level):
  `INT+INT*VAR`, `(VAR*INT)+(VAR*INT)+VAR`...

Why not regular expressions?

- Programming languages are typically not regular.
- E.g., the language of valid expressions
- See: *pumping lemma*, *Myhill-Nerode theorem* – COS 487

Context-free grammars

- A *context-free grammar* $G = (N, \Sigma, R, S)$ consists of:
  - N: a finite set of *non-terminal symbols*
  - Σ: a finite alphabet (or set of *terminal symbols*), disjoint from N
  - $R \subseteq N \times (N \cup \Sigma)^*$ a finite set of *rules* or *productions*
    - Rules often written $A \to w$
    - A is a non-terminal (*left-hand side*)
    - w is a word over N and Σ (*right-hand side*)
  - $S \in N$: the starting non-terminal.

Context-free grammars

- A *context-free grammar* $G = (N, \Sigma, R, S)$ consists of:
  - N: a finite set of *non-terminal symbols*
  - Σ: a finite alphabet (or set of *terminal symbols*), disjoint from N
  - $R \subseteq N \times (N \cup \Sigma)^*$ a finite set of *rules* or *productions*
    - Rules often written $A \to w$
    - A is a non-terminal (*left-hand side*)
    - w is a word over N and Σ (*right-hand side*)
  - $S \in N$: the starting non-terminal.
- *Backus-Naur form* is specialized syntax for writing context-free grammars
  - Non-terminal symbols are written between <,>s
  - Rules written as `<expr> ::= <expr>+<expr>`
  - | abbreviates multiple productions w/ same left-hand side
    - `<expr> ::= <expr>+<expr> | <expr>*<expr>` means
      `<expr> ::= <expr>+<expr>`
      `<expr> ::= <expr>*<expr>`

Derivations

- A *derivation* consists of a finite sequence of words $w_1, \ldots, w_n \in (N \cup \Sigma)^*$ such that $w_1 = S$ and for each i, w_{i+1} is obtained from w_i by replacing a non-terminal symbol with the right-hand-side of one of its rules
  - Example:
    - Grammar: `<S> ::= <S><S> | (<S>) |` ϵ
    - Derivations:
      `<S>` $\Rightarrow$ `(<S>)` $\Rightarrow$ `()`
      `<S>` $\Rightarrow$ `<S><S>` $\Rightarrow$ `<S>(<S>)` $\Rightarrow$ `(<S>)(<S>)` $\Rightarrow$ `()(<S>)` $\Rightarrow$ `()()`
      `<S>` $\Rightarrow$ `<S><S>` $\Rightarrow$ `<S>(<S>)` $\Rightarrow$ `<S>()` $\Rightarrow$ `(<S>)()` $\Rightarrow$ `((<S>))()` $\Rightarrow$ `(())()`
  - Formally:
    - For each i, there is some $u, v \in (N \cup \Sigma)^*$ some $A \in N$, and some $x \in (N \cup \Sigma)^*$ such that $w_i = uAv$, $w_{i+1} = uxv$, and $(A, x) \in R$.
- The set of all strings $w \in \Sigma^*$ such that G has a derivation of w is the *language* of G, written $\mathcal{L}(G)$.

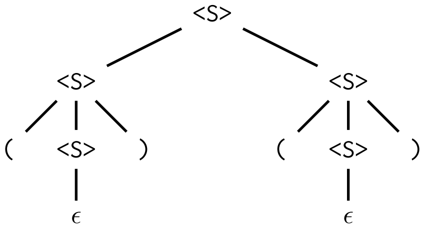
Derivations

- A *derivation* consists of a finite sequence of words $w_1, \ldots, w_n \in (N \cup \Sigma)^*$ such that $w_1 = S$ and for each i, w_{i+1} is obtained from w_i by replacing a non-terminal symbol with the right-hand-side of one of its rules
  - Example:
    - Grammar: `<S> ::= <S><S> | (<S>) |` ϵ
    - Derivations:
      <S> ⇒ (<S>) ⇒ ()
      <S> ⇒ <S><S> ⇒ <S>(<S>) ⇒ (<S>)(<S>) ⇒ ()(<S>) ⇒ ()()
      <S> ⇒ <S><S> ⇒ <S>(<S>) ⇒ <S>() ⇒ (<S>)() ⇒ ((<S>))() ⇒ (())()
  - Formally:
    - For each i, there is some $u, v \in (N \cup \Sigma)^*$ some $A \in N$, and some $x \in (N \cup \Sigma)^*$ such that $w_i = uAv$, $w_{i+1} = uxv$, and $(A, x) \in R$.
- The set of all strings $w \in \Sigma^*$ such that G has a derivation of w is the *language* of G, written $\mathcal{L}(G)$.
- A derivation is *leftmost* if we always substitute the leftmost non-terminal, and *rightmost* if we always substitute the rightmost non-terminal.

Parse trees

- A *parse tree* is a tree representation of a derivation
  - Each leaf node is labelled with a terminal
  - Each internal node is labelled with a non-terminal
    - If an internal node has label X, its children (read left-to-right) are the right-hand-side of a rule w/ left-hand-side X
  - The root is labelled with the start symbol

Parse tree for `()()`, with grammar `<S> ::= <S><S> | (<S>) |` ϵ



Parse trees

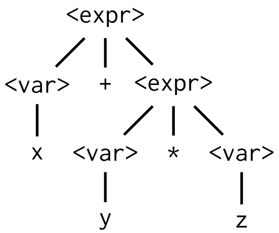
- A *parse tree* is a tree representation of a derivation
    - Each leaf node is labelled with a terminal
    - Each internal node is labelled with a non-terminal
        - If an internal node has label X, its children (read left-to-right) are the right-hand-side of a rule w/ left-hand-side X
    - The root is labelled with the start symbol
- Construct a parse tree from a derivating by "parallelizing" non-terminals
- Parse tree corresponds to *many* derivations
    - Exactly one leftmost derivation (and exactly one rightmost derivation).

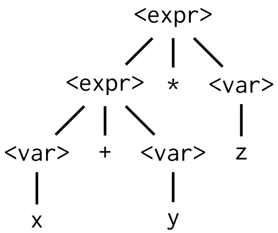
Ambiguity

- A context-free grammar is *ambiguous* if there are two different parse trees for the same word.
    - Equivalently: a grammar is ambiguous if some word has two different left-most derivations

```
<expr> ::=<int> | <var> | <expr>+<expr> | <expr>*<expr> | (<expr>)
 <var> ::=a | … | z
 <int> ::=0 | … | 9
```

$\boxed{\texttt{x+y*z}}$





Eliminating ambiguity

- Ambiguity can often be eliminated by refactoring the grammar
    - Some languages are *inherently ambiguous*: context-free, but every grammar that accepts the language is ambiguous. E.g. $\{a^i b^j c^k : i = j \text{ or } j = k\}$.

Eliminating ambiguity

- Ambiguity can often be eliminated by refactoring the grammar
  - Some languages are *inherently ambiguous*: context-free, but every grammar that accepts the language is ambiguous. E.g. $\{a^i b^j c^k : i = j$ or $j = k\}$.
- Unambiguous expression grammar

```
  <expr> ::=<term>+<expr> | <term>
  <term> ::=<term>*<factor> | <factor>
<factor> ::=<var> | <int> | (<expr>)
```

  - + associates to the right and and $\star$ associates to the left (recursive case right (respectively, left) of operator)
  - $\star$ has higher precedence than + ($\star$ is farther from start symbol)

Regular languages are context-free

Suppose that L is a regular language. Then there is an NFA $A = (Q, \Sigma, \Delta, s, F)$ such that $\mathcal{L}(A) = L$. How can we construct a context-free grammar that accepts L?

Regular languages are context-free

Suppose that L is a regular language. Then there is an NFA $A = (Q, \Sigma, \Delta, s, F)$ such that $\mathcal{L}(A) = L$. How can we construct a context-free grammar that accepts L?
$G = (N, \Sigma, R, S)$, where:

- $N = Q$
- $S = s$
- $R = \{q ::= aq' : (q, a, q') \in \Delta\} \cup \{q ::= \epsilon : q \in F\}$

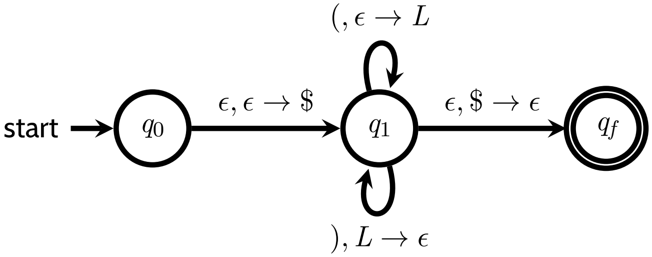
Regular languages are context-free

Suppose that L is a regular language. Then there is an NFA $A = (Q, \Sigma, \Delta, s, F)$ such that $\mathcal{L}(A) = L$. How can we construct a context-free grammar that accepts L?
$G = (N, \Sigma, R, S)$, where:

- $N = Q$
- $S = s$
- $R = \{q ::= aq' : (q, a, q') \in \Delta\} \cup \{q ::= \epsilon : q \in F\}$
- Consequence: could fold lexer definition into grammar definition
- Why not?
  - Separation of concerns
  - Ambiguity is easily understood at lexer level, not parser level
  - Parser generators only handle *some* context-free grammars
    - Non-determinism is easy at the lexer level (NFA $\rightarrow$ DFA conversion)
    - Non-determinism is hard at the parser level (deterministic CFL $\neq$ non-deterministic CFL)

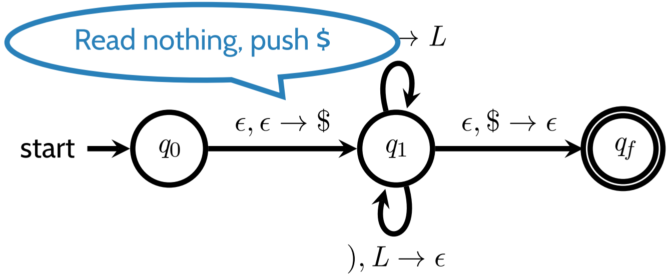
Pushdown automata

- *Pushdown automata* recognize context-free languages
  - PDA:Context-free lanuages :: DFA:Regular languages
  - PDA $\sim$ NFA + a *stack*
- ***Parser generator*** compiles (restricted) grammar to (restricted) PDA
- Pushdown automaton recognizing `<S> ::= <S><S> | (<S>) |` ϵ:
  - *Stack alphabet*: $\$$ marks bottom of the stack, L marks unbalanced left paren



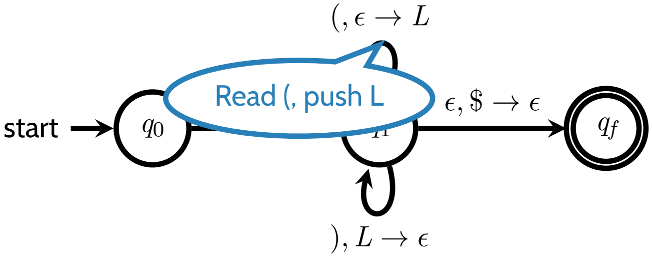
Pushdown automata

- *Pushdown automata* recognize context-free languages
  - PDA:Context-free lanuages :: DFA:Regular languages
  - PDA $\sim$ NFA + a *stack*
- ***Parser generator*** compiles (restricted) grammar to (restricted) PDA
- Pushdown automaton recognizing `<S> ::= <S><S> | (<S>) |` ϵ:
  - *Stack alphabet*: $\$$ marks bottom of the stack, L marks unbalanced left paren



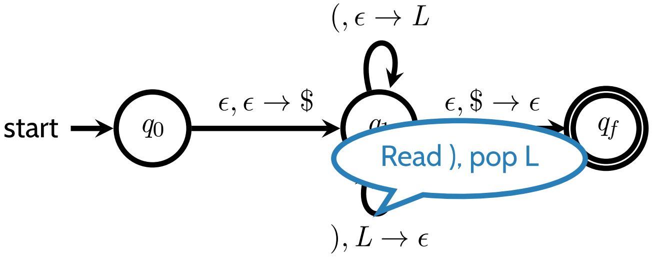
Pushdown automata

- *Pushdown automata* recognize context-free languages
  - PDA:Context-free lanuages :: DFA:Regular languages
  - PDA $\sim$ NFA + a *stack*
- ***Parser generator*** compiles (restricted) grammar to (restricted) PDA
- Pushdown automaton recognizing `<S> ::= <S><S> | (<S>) |` ϵ:
  - *Stack alphabet*: $\$$ marks bottom of the stack, L marks unbalanced left paren



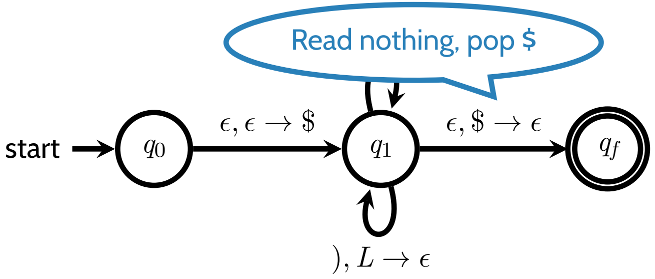
Pushdown automata

- *Pushdown automata* recognize context-free languages
  - PDA:Context-free lanuages :: DFA:Regular languages
  - PDA $\sim$ NFA + a *stack*
- ***Parser generator*** compiles (restricted) grammar to (restricted) PDA
- Pushdown automaton recognizing `<S> ::= <S><S> | (<S>) |` ϵ:
  - *Stack alphabet*: $\$$ marks bottom of the stack, L marks unbalanced left paren



Pushdown automata

- *Pushdown automata* recognize context-free languages
  - PDA:Context-free lanuages :: DFA:Regular languages
  - PDA $\sim$ NFA + a *stack*
- ***Parser generator*** compiles (restricted) grammar to (restricted) PDA
- Pushdown automaton recognizing `<S> ::= <S><S> | (<S>) |` ϵ:
  - *Stack alphabet*: $\$$ marks bottom of the stack, L marks unbalanced left paren


Read nothing, pop $
start
q_0
$\epsilon, \epsilon \rightarrow \$$
q_1
$\epsilon, \$ \rightarrow \epsilon$
q_f
$), L \rightarrow \epsilon$

Pushdown automata, formally

- A *push-down automaton* $A = (Q, \Sigma, \Gamma, \Delta, s, F)$ consists of
  - Q: a finite set of states
  - Σ: an (input) alphabet
  - Γ: a (stack) alphabet
  - $\Delta \subseteq \underbrace{Q}_{\text{source}} \times \underbrace{(\Sigma \cup \{\epsilon\})}_{\text{read input}} \times \underbrace{\Gamma^*}_{\text{read stack}} \times \underbrace{Q}_{\text{dest}} \times \underbrace{\Gamma^*}_{\text{write stack}}$, the transition relation
  - $s \in Q$: start state
  - $F \subseteq Q$: set of final (accepting) states

Pushdown automata, formally

- A *push-down automaton* $A = (Q, \Sigma, \Gamma, \Delta, s, F)$ consists of
  - Q: a finite set of states
  - Σ: an (input) alphabet
  - Γ: a (stack) alphabet
  - $\Delta \subseteq \underbrace{Q}_{\text{source}} \times \underbrace{(\Sigma \cup \{\epsilon\})}_{\text{read input}} \times \underbrace{\Gamma^*}_{\text{read stack}} \times \underbrace{Q}_{\text{dest}} \times \underbrace{\Gamma^*}_{\text{write stack}}$, the transition relation
  - $s \in Q$: start state
  - $F \subseteq Q$: set of final (accepting) states
- A pushdown automaton accepts a word w if w can be written as $w_1 w_2 \ldots w_n$ (each $w_i \in (\Sigma \cup \{\epsilon\})$) s.t. there exists $q_0, q_1, \ldots, q_n \in Q$ and $v_0, v_1, \ldots, v_n \in \Gamma^*$ such that
  1. $q_0 = s$ and $v_0 = \epsilon$ (i.e., the machine starts at the start state with an empty stactk)
  2. for all i, we have $(q_i, w_{i+1}, a, q_{i+1}, b) \in \Delta$, where $v_i = at$ and $v_{i+1} = bt$ for some $a, b, t \in \Gamma^*$
  3. $q_m \in F$. (i.e., the machine ends at a final state).