Parsing

Stefan D. Bruda

CS 310, Winter 2025

Parsing

- Interface to lexical analysis:

```
typename vocab;          /* alphabet + end-of-string */
const vocab EOS;         /* end-of-string pseudo-token */
vocab gettoken(void); /* returns next token */
```

- Parsing = determining whether the current input belongs to the given language
  - In practice a parse tree is constructed in the process as well
- General method: Not as efficient as for finite automata
  - Various paths with different states and also different stack content along them
  - Careful housekeeping (dynamic programming) reduces the otherwise exponential complexity to $O(n^3)$
  - We want linear time instead = deterministic PDA
    - General idea: try to determine what to do next based on the next token in the input

Recursive Descent Parsing

- Basic idea: simulate a PDA for the given language
  - Stack manipulation = . . .

Recursive Descent Parsing

- Basic idea: simulate a PDA for the given language
  - Stack manipulation = function calls and returns
- Construct a function for each nonterminal
- Decide which function to call based on the next input token = linear complexity

```
vocab t;
void MustBe (vocab ThisToken) {
    if (t != ThisToken) { printf("reject"); exit(0); }
    t = gettoken();
}
void Balanced (void) {
    switch (t) {
    case EOS:
    case ONE: /* <empty> */
        break;
    default: /* 0 <balanced> 1 */
        MustBe(ZERO);
        Balanced();
        MustBe(ONE);
   } /* switch */     } /* Balanced */
```

```
int main (void) {
    t = gettoken();
    Balanced();
    /* accept iff
       t == EOS   */
}
```

Recursive Descent Parsing: Left Factoring

- Not all grammars are suitable for recursive descent:

$$
\begin{array}{rcl}
\langle \text{stmt} \rangle & ::= & \langle \text{empty} \rangle \\
& | & \texttt{VAR} := \langle \text{exp} \rangle \\
& | & \texttt{IF} \ \langle \text{exp} \rangle \ \texttt{THEN} \ \langle \text{stmt} \rangle \ \texttt{ELSE} \ \langle \text{stmt} \rangle \\
& | & \texttt{WHILE} \ \langle \text{exp} \rangle \ \texttt{DO} \ \langle \text{stmt} \rangle \\
& | & \texttt{BEGIN} \ \langle \text{seq} \rangle \ \texttt{END} \\
\langle \text{seq} \rangle & ::= & \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle \ ; \ \langle \text{seq} \rangle
\end{array}
$$

  - Both rules for $\langle \text{seq} \rangle$ begin with the same nonterminal
  - Impossible to decide which one to apply based only on the next token
  - Fortunately concatenation is distributive over union so we can fix the grammar (left factoring):

$$
\begin{array}{rcl}
\langle \text{seq} \rangle & ::= & \langle \text{stmt} \rangle \ \langle \text{seqTail} \rangle \\
\langle \text{seqTail} \rangle & ::= & \langle \text{empty} \rangle \mid ; \ \langle \text{seq} \rangle
\end{array}
$$

Recursive Descent Parsing: Ambiguity

- Some programming constructs are inherently ambiguous

$$\begin{array}{rcl} \langle \text{stmt} \rangle & ::= & \texttt{if (} \langle \text{exp} \rangle \texttt{)} \langle \text{stmt} \rangle \\ & | & \texttt{if (} \langle \text{exp} \rangle \texttt{)} \langle \text{stmt} \rangle \texttt{ else } \langle \text{stmt} \rangle \end{array}$$

- Solution: choose one path and stick to it (e.g., match the else-statement with the nearest else-less if statement)

```
case IF:
    t = gettoken();
    MustBe(OPEN_PAREN);
    Expression();
    MustBe(CLS_PAREN);
    Statement();
    if (t == ELSE) {
        t = gettoken();
        Statement();
    }
```

Ambiguity (C++): It's a type, it's a plane

- Consider the following code:

```
int y;
template <class T> void g(T& v) {
    T::x(y);
}
```

- The statement `T::x(y)` can be

Ambiguity (C++): It's a type, it's a plane

- Consider the following code:

```
int y;
template <class T> void g(T& v) {
    T::x(y);
}
```

- The statement `T::x(y)` can be
  - A function call (member function `x` of `T` applied to `y`), or
  - A declaration of `y` as a variable of type `T::x`

Ambiguity (C++): It's a type, it's a plane

- Consider the following code:

```
int y;
template <class T> void g(T& v) {
    T::x(y);
}
```

- The statement `T::x(y)` can be
  - A function call (member function `x` of `T` applied to `y`), or
  - A declaration of `y` as a variable of type `T::x`
- Resolution: unless otherwise specified, an identifier is assumed to refer to something that is not a type or template.
  - If we want something else, we use the keyword `typename`:

```
T::x(y);            // function x of T applied to y
typename T::x(y);   // y is a variable of type T::x
```

  - The sole purpose of `typename` is to resolve the ambiguity in the desired direction

Recursive Descent Parsing: Closure, etc.

- Any left recursion in the grammar will cause the parser to go into an infinite loop:

$$\langle \text{exp} \rangle \quad ::= \quad \langle \text{exp} \rangle \ \langle \text{addop} \rangle \ \langle \text{term} \rangle \mid \langle \text{term} \rangle$$

- Solution: eliminate left recursion using a closure

$$\begin{aligned} \langle \text{exp} \rangle &::= \langle \text{term} \rangle \ \langle \text{closure} \rangle \\ \langle \text{closure} \rangle &::= \langle \text{empty} \rangle \\ &\mid \langle \text{addop} \rangle \ \langle \text{term} \rangle \ \langle \text{closure} \rangle \end{aligned}$$

  - Not the same language theoretically, but differences not relevant in practice
- This being said, some languages are simply not parseable using recursive descent

$$\langle \text{palindrome} \rangle \quad ::= \quad \langle \text{empty} \rangle \mid 0 \mid 1 \mid 0 \ \langle \text{palindrome} \rangle \ 0 \mid 1 \ \langle \text{palindrome} \rangle \ 1$$

  - No way to know when to choose the $\langle \text{empty} \rangle$ rule
  - No way to choose between the second and the fourth rule
  - No way to choose between the third and the fifth rule

Recursive Descent Parsing: Sufficient Conditions

- *first*(α) = set of all initial tokens in the strings derivable from α
- *follow*($\langle$N$\rangle$) = set of all initial tokens in nonempty strings that may follow $\langle$N$\rangle$ (possibly including `EOS`)
- Sufficient conditions for a grammar to allow recursive descent parsing:
  - For $\langle$N$\rangle$ ::= $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ must have *first*(α_i) $\cap$ *first*(α_j) $= \emptyset$, $1 \leq i < j \leq n$
  - Whenever $\langle$N$\rangle \Rightarrow^* \varepsilon$ must have *follow*($\langle$N$\rangle$) $\cap$ *first*($\langle$N$\rangle$) $= \emptyset$
- Grammars that do not have these properties may be fixable using left factoring, closure, etc.

Recursive Descent Parsing: Sufficient Conditions

- *first*(α) = set of all initial tokens in the strings derivable from α
- *follow*($\langle$N$\rangle$) = set of all initial tokens in nonempty strings that may follow $\langle$N$\rangle$ (possibly including `EOS`)
- Sufficient conditions for a grammar to allow recursive descent parsing:
  - For $\langle \text{N} \rangle ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ must have *first*(α_i) $\cap$ *first*(α_j) $= \emptyset$, $1 \leq i < j \leq n$
  - Whenever $\langle \text{N} \rangle \Rightarrow^* \varepsilon$ must have *follow*($\langle$N$\rangle$) $\cap$ *first*($\langle$N$\rangle$) $= \emptyset$
- Grammars that do not have these properties may be fixable using left factoring, closure, etc.
- Method for constructing the recursive descent function `N()` for the nonterminal $\langle$N$\rangle$ with rules $\langle \text{N} \rangle ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:
  1. For $\alpha_i \neq \varepsilon$ apply the rewriting rule $\langle \text{N} \rangle ::= \alpha_i$ whenever the next token in the input is in *first*(α_i)
  2. For $\alpha_i = \varepsilon$ apply the rewriting rule $\langle \text{N} \rangle ::= \alpha_i$ (that is, $\langle \text{N} \rangle ::= \varepsilon$) whenever the next token in the input is in *follow*($\langle$N$\rangle$)
  3. Signal a syntax error in all the other cases

Recursive Descent Example

```
typedef enum { VAR, EQ, IF, ELSE, WHILE, OPN_BRACE, CLS_BRACE,
               OPN_PAREN, CLS_PAREN, SEMICOLON, EOS            } vocab;
vocab gettoken() {...}
vocab t;
void MustBe(vocab ThisToken) {...}

void Statement();
void Sequence();

int main() {
    t = gettoken();
    Statement();
    if (t != EOS) printf("String not accepted\n");
    return 0;         }
void Sequence() {
    if (t == CLS_BRACE) /* <empty> */ ;
    else { /* <statement> <sequence> */
        Statement();
        Sequence();
    }    }
```

Recursive Descent Example (cont'd)

```
void Statement() {
    switch(t) {
    case SEMICOLON: /* ; */
        t = gettoken();
        break;
    case VAR: /* VAR = <exp> */
        t = gettoken();
        MustBe(EQ);
        Expression();
        MustBe(SEMICOLON);
        break;
    case IF: /* if (<expr>) <statement> else <statement> */
        t = gettoken();
        MustBe(OPEN_PAREN);
        Expression();
        MustBe(CLS_PAREN);
        Statement();
        MustBe(ELSE);
        Statement();
        break;
```

Recursive Descent Example (cont'd)

```
    case WHILE: /* while (exp) <statement> */
        t = gettoken();
        MustBe(OPEN_PAREN);
        Expression();
        MustBe(CLS_PAREN);
        Statement();
        break;
    default: /* { <sequence } */
        MustBe(OPN_BRACE);
        Sequence();
        MustBe(CLS_BRACE);
    } /* switch */
} /* Statement () */
```

Constructing the Parse Tree

- The parse tree can be constructed through the recursive calls:
  - Each function creates a current node
  - The children are populated through recursive calls
  - The current node is then returned

```
class Node {...};

Node* Sequence() {
    Node* current = new Node(SEQ, ...);
    if (t == CLS_BRACE) /* <empty> */ ;
    else { /* <statement> <sequence> */
        current.addChild(Statement());
        current.addChild(Sequence());
    }
    return current;
}
```

Constructing the Parse Tree (cont'd)

```
Node* Statement() {
    Node* current;
    switch(t) {
    case SEMICOLON: /* ; */
        t = gettoken();
        return new Node(EMPTY);
        break;
    case VAR: /* VAR = <exp> */
        current = new Node(ASSIGN, ...);
        current.addChild(VAR, ...);
        t = gettoken();
        MustBe(EQ);
        current.addChild(Expression());
        MustBe(SEMICOLON);
        break;
    case IF: /* if (<expr>) <statement> else <statement> */
        current = new Node(COND, ...);
    /* ... */
    }
    return current;
}
```

Parsing in Practice: Bottom-Up Parsers

- Substantial effort may be needed to make grammars suitable for recursive descent
- Recursive descent code relatively difficult to generate
- Recursive descent code uses processor stack frames (expensive space-wise), it is more efficient to simulate a pushdown automaton
- Bottom-up parsing better in practice: Given $G = (N, \Sigma, S, R)$ construct the automaton $M = (\{p, q\}, \Sigma, N + \Sigma, \Delta, s, \{q\})$ with Δ containing exactly all the transitions:

$$\begin{array}{rrl} \text{shift} & \forall\, a \in \Sigma: & ((p, a, \varepsilon), (p, a)) \\ \text{reduce} & \forall\, A \to \alpha \in R: & ((p, \varepsilon, \alpha^{\mathbb{R}}), (p, A)) \\ \text{done} & & ((p, \varepsilon, S), (q, \varepsilon)) \end{array}$$

- Result highly nondeterministic, but nondeterminism fixable for weak-precedence grammars = most programming languages:
  - When to shift and when to reduce?
    - Establish a precedence relation $P \subseteq (N \cup \Sigma) \times \Sigma$
    - If (stack-top, input) $\in P$ then we reduce, else we shift
  - When we reduce, with what rule we reduce?
    - We use the logest rule = greedy (eat up the longest stack top)

Example or Bottom-Up Parsing

$$
\begin{array}{lcl}
E & \to & E + T \\
E & \to & T \\
T & \to & T * F \\
T & \to & F \\
F & \to & (E) \\
F & \to & y
\end{array}
$$

P	(	)	y	$+$	$*$	EOS
(						
)		✓		✓	✓	✓
y		✓		✓	✓	✓
$+$						
$*$						
E						
T		✓		✓		✓
F		✓		✓	✓	✓

$$
\begin{array}{lrll}
(& (p, a, \varepsilon), & (p, a) &), a \in \{+ . * . (,), y\} \\
(& (p, \varepsilon, T + E), & (p, E) &) \\
(& (p, \varepsilon, T), & (p, E) &) \\
(& (p, \varepsilon, F * T), & (p, T) &) \\
(& (p, \varepsilon, F), & (p, T) &) \\
(& (p, \varepsilon,)E(), & (p, F) &) \\
(& (p, \varepsilon, y), & (p, F) &) \\
(& (p, \varepsilon, E) & (q, \varepsilon) &)
\end{array}
$$

	Input	Stack
	y + y * y	ε
shift	+ y * y	y
red	+ y * y	F
red	+ y * y	T
red	+ y * y	E
shift	y * y	+ E
shift	* y	y + E
red	* y	F + E
red	* y	T + E
shift	y	* T + E
shift	ε	y * T + E
red	ε	F * T + E
g-red	ε	T + E
g-red	ε	E
done	ε	ε

red = reduce (unambiguous)
g-red = greedy reduce (longest rule)