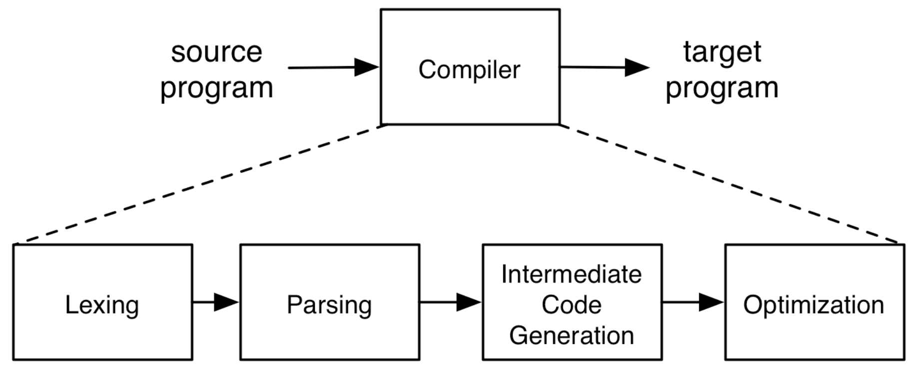
CMSC 330: Organization of Programming Languages

Parsing

Steps of Compilation



Parsing

- There are many efficient techniques for parsing, i.e., turning strings into parse trees
  - Examples:
    - LL(k), SLR(k), LR(k), LALR(k)…
    - Take CMSC 430 for more details
- One simple technique is recursive descent parsing
  - This is a “top-down” parsing algorithm

Recursive Descent Parsing

- Goal: determine if we can produce the string to be parsed from the grammar's start symbol
- Approach: recursively replace nonterminals with right-hand sides of their productions
- At each step, we'll keep track of two facts:
  - What tree node are we trying to match?
  - What is the *lookahead* (the next token of the input string)?
    - The lookahead helps guide the selection of the production used to replace a nonterminal

Recursive Descent Parsing (cont.)

- At each step there are three possible cases:
  - If we're trying to match a terminal, and the lookahead is that token, then succeed, advance the lookahead, and continue
  - If we're trying to match a nonterminal, pick which production to apply based on the lookahead
  - Otherwise fail with a parsing error

Parsing Example

E → id = n | {L}

L → E ; L | ε

  - Here n is an integer and id is an identifier

- One input might be {x = 3; {y = 4;};}
  - This would get turned into a list of tokens

    `{ x = 3 ; { y = 4 ; } ; }`
  - And we want to turn it into a parse tree

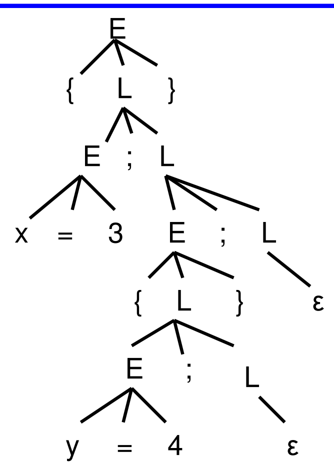
Parsing Example (cont.)

E → id = n | {L}

L → E ; L | ε

{x = 3 ; {y = 4 ;} ;}

lookahead



Recursive Descent Parsing (cont.)

- The key step is choosing which production should be selected
- Two approaches are:
  - Backtracking:
    - Choose some production
    - If it fails, try a different production
    - The parse fails if all choices fail
  - Predictive parsing:
    - Analyze the grammar to find FIRST sets for productions
    - Compare with lookahead to decide which production to select
    - The parse fails if the lookahead does not match FIRST

First Sets

- Motivating example:
  - The lookahead is x
  - Given the grammar S → xyz | abc
    - Select S → xyz since 1st terminal in RHS matches x
  - Given the grammar S → A | B   A → x | y   B → z
    - Select S → A, since A can derive string beginning with x
- In general:
  - Choose a production that can derive a sentential form beginning with the lookahead
  - Need to know what terminal may be first in any sentential form derived from a nonterminal / production

First Sets

- Definition: *First*(γ), for any terminal or nonterminal γ, is the set of initial terminals of all strings that γ may expand to
  - We'll use this to decide what production to apply
- Examples:
  - Given the grammar S → xyz | abc
    - First(xyz) = {x}, First(abc) = {a}
    - First(S) = First(xyz) U First(abc) = {x, a}
  - Given the grammar S → A | B   A → x | y   B → z
    - First(x) = {x}, First(y) = {y}, First(A) = {x, y}
    - First(z) = {z}, First(B) = {z}
    - First(S) = {x, y, z}

Calculating First(γ)

- For a terminal a, First(a) = {a}
- For a nonterminal N
  - If N → ε, then add ε to First(N)
  - If N → $\alpha_1\ \alpha_2\ \ldots\ \alpha_n$, then (note the α_i are all the symbols on the right side of one single production):
    - Add First($\alpha_1\alpha_2 \ldots \alpha_n$) to First(N), where First($\alpha_1\ \alpha_2 \ldots \alpha_n$) is defined as
      - First(α_1) if $\varepsilon \notin$ First(α_1)
      - Otherwise (First(α_1) – ε) ∪ First($\alpha_2 \ldots \alpha_n$)
    - If $\varepsilon \in$ First(α_i) for all i, $1 \le i \le k$, then add ε to First(N)

First() Examples

E → id = n | {L}	E → id = n | {L} | ε
L → E ; L | ε	L → E ; L
First(id) = {id}	First(id) = {id}
First("=") = {"="}	First("=") = {"="}
First(n) = {n}	First(n) = {n}
First("{")= {"{"}	First("{")= {"{"}
First("}")= {"}"}	First("}")= {"}"}
First(";")= {";"}	First(";")= {";"}
First(E) = {id, "{" }	First(E) = {id, "{", ε}
First(L) = {id, "{", ε}	First(L) = {id, "{", ";"}

Recursive Descent Parser Implementation

- For terminals, create a function match(a)
  - If the lookahead is a it consumes the lookahead by advancing the lookahead to the next token, and returns
  - Otherwise fails with a parse error if lookahead is not a
- For each nonterminal N, create a function parse_N
  - Called when we're trying to parse a part of the input which corresponds to (or can be derived from) N
  - parse_S for the start symbol S begins the parse

Parser Implementation (cont.)

- The body of parse_N for a nonterminal N does the following:
  - Let $N \rightarrow \beta_1 \mid \ldots \mid \beta_k$ be the productions of N
    - Here β_i is the entire right side of a production- a sequence of terminals and nonterminals
  - Pick the production $N \rightarrow \beta_i$ such that the lookahead is in $First(\beta_i)$
    - It must be that $First(\beta_i) \cap First(\beta_j) = \emptyset$ for $i \neq j$
    - If there is no such production, but $N \rightarrow \varepsilon$ then return
    - Otherwise fail with a parse error
  - Suppose $\beta_i = \alpha_1\ \alpha_2 \ldots \alpha_n$. Then call parse_$\alpha_1$(); ... ; parse_$\alpha_n$() to match the expected right-hand side, and return

Parser Implementation (cont.)

- The parse is built on procedure calls
- Procedures may be (mutually) recursive

Recursive Descent Parser

- Given grammar $S \rightarrow xyz \mid abc$

  First(xyz) = {x}, First(abc) = {a}
- Parser:

```
parse_S() {
  if (lookahead == "x") {
    match("x"); match("y"); match("z");        // S → xyz
  } else if (lookahead == "a") {
    match("a"); match("b"); match("c");        // S → abc
  } else error();
}
```

Recursive Descent Parser

- Given grammar $S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z$
  - First(A) = {x, y}, First(B) = {z}
- Parser:

```
parse_S() {
  if (lookahead == “x”) ||
    lookahead == “y”)
    parse_A();    // S → A
  else
    if (lookahead == “z”)
      parse_B();  // S → B
    else error();
}
```

```
parse_A() {
  if (lookahead == “x”)
    match(“x”);     // A → x
  else
    if (lookahead == “y”)
      match(“y”);  // A → y
    else error();
}

parse_B() {
  if (lookahead == “z”)
    match(“z”);     // B → z
  else error();
}
```