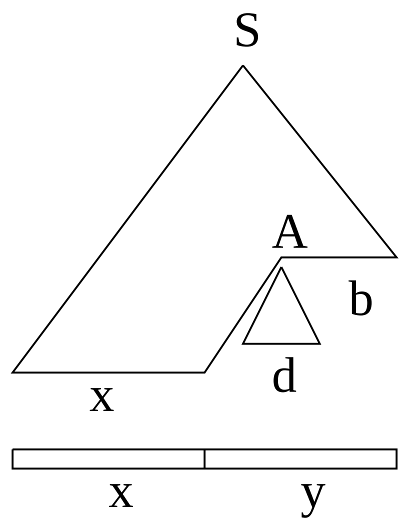
Class Information

- Reminder: Second homework due on Friday, February 14, before class.

Review: Top-Down Parsing - LL(1)



Basic Idea:

- The parse tree is constructed from the root, expanding **non-terminal** nodes on the tree's frontier following a left-most derivation
- The input program is read from left to right, and input tokens are read (consumed) as the program is parsed
- The next **non-terminal** symbol is replaced by one of its rules. The particular choice has to be unique, and uses parts of the input (partially parsed program), for instance the first **token** of the remaining input

Review: LL(1) Parsing

Basic idea:

> For any two productions $A ::= \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.

For some *rhs* $\alpha \in G$, define **FIRST**(α) as the set of tokens that appear as the first symbol in some string derived from α.

That is

$x \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* x\gamma$ for some γ, and

$\epsilon \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \epsilon$

For a non-terminal A, define **FOLLOW**(A) as the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it.

A terminal symbol has no FOLLOW set

$\boxed{\text{FIRST and FOLLOW sets can be constructed automatically}}$

For a string of grammar symbols α, define $\text{FIRST}(\alpha)$ as

- the set of terminal symbols that begin strings derived from α
- if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$ contains the set of tokens valid in the first position of α

STEP 1: Build $\text{FIRST}(X)$ for all grammar symbols X:

1. `if` X `is a terminal,` $\text{FIRST}(X)$ `is` $\{X\}$
2. `if` $X ::= \epsilon$`, then` $\epsilon \in \text{FIRST}(X)$

3. `iterate until no more terminals or` ϵ `can be added to any` $\text{FIRST}(X)$`:`

  `if` $X ::= Y_1 Y_2 \cdots Y_k$ `then`
    $a \in \text{FIRST}(X)$ `if` $a \in \text{FIRST}(Y_i)$
      `and` $\epsilon \in \text{FIRST}(Y_j)$ `for all` $1 \le j < i$
    $\epsilon \in \text{FIRST}(X)$ `if` $\epsilon \in \text{FIRST}(Y_i)$ `for all` $1 \le i \le k$
  `end iterate`

(If $\epsilon \notin \text{FIRST}(Y_1)$, then $\text{FIRST}(Y_i)$ is irrelevant, for $1 < i$)

STEP 2: Build $\text{FIRST}(\alpha)$ for $\alpha = X_1 X_2 \cdots X_n$:

- $a \in \text{FIRST}(\alpha)$ `if` $a \in \text{FIRST}(X_i)$
  `and` $\epsilon \in \text{FIRST}(X_j)$ `for all` $1 \leq j < i$
- $\epsilon \in \text{FIRST}(\alpha)$ `if` $\epsilon \in \text{FIRST}(X_i)$ `for all` $1 \leq i \leq n$

FOLLOW set construction

For a non-terminal A, define FOLLOW(A) as

> the set of terminals that can appear immediately to the right of A in some sentential form

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it

A terminal symbol has no FOLLOW set

To build FOLLOW(X) for non-terminal X:

1. `place eof in` FOLLOW(`⟨goal⟩`)

  <u>`iterate`</u> <u>`until`</u> `no more terminals or` ϵ `can be added to any` FOLLOW(`$X$`):

2. `if` $A ::= \alpha B \beta$ `then`
  `put` $\{\text{FIRST}(\beta) - \epsilon\}$ `in` FOLLOW(B)

3. `if` $A ::= \alpha B$ `then`
  `put` FOLLOW(A) `in` FOLLOW(B)

4. `if` $A ::= \alpha B \beta$ `and` $\epsilon \in$ FIRST(β) `then`
  `put` FOLLOW(A) `in` FOLLOW(B)

  <u>`end`</u> <u>`iterate`</u>

Review: LL(1) Grammar

Define $FIRST^{+}(\delta)$ for rule $A ::= \delta$

- $FIRST(\delta)$ - $\{\epsilon\} \cup$ Follow(A), if $\epsilon \in FIRST(\delta)$
- $FIRST(\delta)$ otherwise

A grammar is LL(1) iff

$(A ::= \alpha$ and $A ::= \beta)$ implies

$$FIRST^{+}(\alpha) \cap FIRST^{+}(\beta) = \emptyset$$

Recursive Descent Parsing

Now, we can produce a simple recursive descent parser from our favorite LL(1) expression grammar.

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated **parsing procedure** that can recognize any sequence of tokens generated by that non–terminal.
- There is a **main** routine to initialize all globals (e.g.: `token`) and call the start symbol. On return, check whether `token` == **eof**, and whether errors occurred.
- Within a parsing procedure, both non–terminals and terminals can be matched:
  - non–terminal A — call parsing procedure for A
  - token t — compare t with current input token; if match, consume input, otherwise ERROR
- Parsing procedures may contain code that performs some useful "computation" (syntax directed translation).

Recursive Descent Parsing (pseudo code)

	a	b	eof	other
S	aSb	ϵ	ϵ	error

```
main: {
  token := next_token( );
  if (S( ) and token == eof) print ‘‘accept’’ else print ‘‘error’’;
}


bool S:
  switch token {
    case a:  token := next_token( );
             if (not S( )) return false; // recursive call to S;
             if token == b {
                  token := next_token( )
                  return true;
             }
             else
                  return false;
             break;
    case b,
    case eof:return true;
             break;
    default: return false;
  }
```

How to parse input **a a a b b b** ?

Example: tinyL Language

<program> ::= <stmtlist> .

<stmtlist> ::= <stmt> <morestmts>

<morestmts> ::= ; <stmtlist> | ϵ

<stmt> ::= <assign> | <read> | <print>

<assign> ::= <variable> = <expr>

<read> ::= ? <variable>

<print> ::= ! <variable>

<expr> ::= + <expr> <expr> |
- <expr> <expr> |
* <expr> <expr> |
<variable> |
<digit>

<variable> :: = a | b | c | d | e

<digit> :: = 0 | 1 | 2 | 3 | ... | 9

Syntax Directed Translation

Examples:

1. Interpreter
2. Code generator
3. Type checker
4. Performance estimator

$\boxed{\text{Use hand-written recursive descent LL(1) parser}}$

Example: Interpreter

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | ... | 9

```
int expr:  // returns value of expression
  int val1, val2; // values
  switch token {
    case +:    token := next_token( );
               val1 = expr( ); val2 = expr( );
               return val1+val2;
    case 0..9: return digit( );
    ...
  }

int digit:  // returns value of constant
  switch token {
    case 1:    token := next_token( );
               return 1;
    case 2:    token := next_token( );
               return 2;
    ...
  }
```

Example: Interpreter

What happens when you parse subprogram

“**+ 2 + 1 2**” ?

The parsing produces:

```
5
```

Example: Simple Code Generation

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | … | 9

```
int expr:  // returns target register of operation
  int target_reg = next_register( ); // ‘‘fresh’’ register
  int reg1, reg2; // other registers
  switch token {
    case +:    token := next_token( );
               reg1 = expr( ); reg2 = expr( );
               CodeGen(ADD, reg1, reg2, target_reg);
               return target_reg;
    case 0..9: return digit( );
    ...
  }

int digit:  // returns target register of operation
  int target_reg = next_register( ); // ‘‘fresh’’ register
  switch token {
    case 1:    token := next_token( );
               CodeGen(LOADI, 1, target_reg);
               return target_reg;
    case 2:    token := next_token( );
               CodeGen(LOADI, 2, target_reg);
               return target_reg;
    ...
  }
```

Example: Simple Code Generation

What happens when you parse subprogram

“+ **2** + **1** **2**” ?

Assumption:

$\boxed{\text{first call to } \texttt{next\_register( )} \text{ will return 1}}$

The parsing produces:

```
LOADI 2 => r2
LOADI 1 => r4
LOADI 2 => r5
ADD r4, r5 => r3
ADD r2, r3 => r1
```

Example: Simple Type Checker

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | . . . | 9

```
string expr:  // returns type expression
  string type1, type2; // other type expressions
  switch token {
    case +:    token := next_token( );
               type1 = expr( ); type2 = expr( );
               if (type1 == ``int'' and type2 == ``int'') {
                    return ``int'' else
                    return ``error'';
               };
    case 0..9: return digit( );
    ...
  }

string digit:  // returns type expression
  switch token {
    case 1:    token := next_token( );
               return ``int'';
    case 2:    token := next_token( );
               return ``int'';
    ...
  }
```

Example: Simple Type Checker

What happens when you parse subprogram

“**+ 2 + 1 2**” ?

The parsing produces:

`‘‘int’’`

Example: Basic Performance Predictor

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | ... | 9

```
int expr:  // returns cycles needed to compute expression
  int cyc1, cyc2; // subexpression cycles
  switch token {
    case +:    token := next_token( );
               cyc1 = expr( ); cyc2 = expr( );
               return cyc1+cyc2+2 // ADD takes 2 cycles;
    case 0..9: return digit( );
    ...
  }

int digit:  // returns cycles
  switch token {
    case 1:    token := next_token( );
               return 1; // LOADI takes 1 cycle
    case 2:    token := next_token( );
               return 1; // LOADI takes 1 cycle
    ...
  }
```

Example: Basic Performance Predictor

What happens when you parse subprogram

“**+ 2 + 1 2**” ?

The parsing produces:

7

Next Lecture

Things to do:

Start programming in C. Check out the web for tutorials.

Next time:

- Imperative programming languages
- Introduction to C
- Pointers and dynamic memory management in C