Lesson 2

• The structure of a compiler
• Overview of a Simple Compiler Front-end
  – Predictive top-down parsing
  – Syntax directed translation
  – Lexical analysis
Admins

• Office Hours:
  – Wednesday, 9 - 11 ➡️ my proposal
  – Monday, 18 - 19:30
  – Friday, 9-11

• Check your data and add the University ID (matricola) in the sheet
The Many Phases of a Compiler

1. Lexical analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Code Generator
7. Peephole Optimization

1, 2, 3, 4: Front-End
5, 6, 7: Back-End
Compiler Front- and Back-end

Source program (character stream)

- **Scanner** *(lexical analysis)*
  - Tokens
  - **Parser** *(syntax analysis)*
    - Parse tree
    - **Semantic Analysis**
      - Abstract syntax tree, or …
      - **Intermediate Code Generation**
        - Three address code, or …

Back end synthesis

- **Intermediate Code Generation**
  - Modified intermediate form
    - **Target Code Generation**
      - Assembly or object code
        - **Machine-Specific Code Improvement**
          - Modified assembly or object code

Front end analysis

- **Machine-Independent Code Improvement**
  - Three address code, or…
Single-pass vs. Multi-pass Compilers

• A collection of compilation phases is done only once (single pass) or multiple times (multi pass)

• **Single pass**: more efficient and uses less memory
  – requires everything to be defined before being used
  – standard for languages like Pascal, FORTRAN, C
  – Influenced the design of early programming languages

• **Multi pass**: needs more memory (to keep entire program), usually slower
  – needed for languages where declarations e.g. of variables may follow their use (Java, ADA, …)
  – allows better optimization of target code
Overview of a Simple Compiler Front-end

• Building a compiler involves:
  – Defining the syntax of a programming language
  – Develop a source code parser: we consider here *predictive parsing*
  – Implementing *syntax directed translation* to generate intermediate code
The Structure of the Front-End

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Intermediate representation

- Develop parser and code generator for translator
- Syntax definition (BNF grammar)
- IR specification
Syntax Definition

• Context-free grammar is a 4-tuple with
  – A set of tokens (*terminal* symbols)
  – A set of *nonterminals*
  – A set of *productions*
  – A designated *start symbol*
Example Grammar

Context-free grammar for simple expressions:

\[ G = \langle \{ \text{list}, \text{digit} \}, \{ +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}, P, \text{list} \rangle \]

with productions \( P = \)

\[ \text{list} \rightarrow \text{list} + \text{digit} \]

\[ \text{list} \rightarrow \text{list} - \text{digit} \]

\[ \text{list} \rightarrow \text{digit} \]

\[ \text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]
Derivation

• Given a CF grammar we can determine the set of all *strings* (sequences of tokens) generated by the grammar using *derivation*
  
  – We begin with the start symbol
  
  – In each step, we replace one nonterminal in the current *sentential form* with one of the right-hand sides of a production for that nonterminal
Derivation for the Example Grammar

\[
\text{list} \\
\Rightarrow \underline{\text{list}} + \text{digit} \\
\Rightarrow \underline{\text{list}} - \text{digit} + \text{digit} \\
\Rightarrow \underline{\text{digit}} - \text{digit} + \text{digit} \\
\Rightarrow 9 - \underline{\text{digit}} + \text{digit} \\
\Rightarrow 9 - 5 + \underline{\text{digit}} \\
\Rightarrow 9 - 5 + 2
\]

This is an example \textit{leftmost derivation}, because we replaced the leftmost nonterminal (underlined) in each step. Likewise, a \textit{rightmost derivation} replaces the rightmost nonterminal in each step.
Parse Trees

• The *root* of the tree is labeled by the start symbol
• Each *leaf* of the tree is labeled by a terminal (=token) or ε
• Each *interior node* is labeled by a nonterminal
• If $A \rightarrow X_1 X_2 \ldots X_n$ is a production, then node $A$ has immediate *children* $X_1, X_2, \ldots, X_n$ where $X_i$ is a (non)terminal or $\varepsilon$ ($\varepsilon$ denotes the *empty string*)
Parse Tree for the Example Grammar

Parse tree of the string \(9-5+2\) using grammar \(G\)

The sequence of leafs is called the \textit{yield} of the parse tree.
Consider the following context-free grammar:

\[ G = \langle \{ \text{string} \}, \{ +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}, P, \text{string} \rangle \]

with production \( P = \)

\[ \text{string} \rightarrow \text{string} + \text{string} | \text{string} - \text{string} | 0 | 1 | \ldots | 9 \]

This grammar is ambiguous, because more than one parse tree represents the string 9-5+2
Ambiguity (cont’d)
Associativity of Operators

*Left-associative* operators have *left-recursive* productions

\[
left \rightarrow left + \text{term} \mid \text{term}
\]

String \(a+b+c\) has the same meaning as \((a+b)+c\)

*Right-associative* operators have *right-recursive* productions

\[
right \rightarrow \text{term} = right \mid \text{term}
\]

String \(a=b=c\) has the same meaning as \(a=(b=c)\)
Precedence of Operators

Operators with higher precedence “bind more tightly”

\[
\begin{align*}
expr &\to expr + term \mid term \\
term &\to term * factor \mid factor \\
factor &\to number \mid ( expr )
\end{align*}
\]

String \(2+3\times5\) has the same meaning as \(2+(3\times5)\)
Syntax of Statements

\[ stmt \rightarrow id := expr \]
\[ \text{if } expr \text{ then } stmt \]
\[ \text{if } expr \text{ then } stmt \text{ else } stmt \]
\[ \text{while } expr \text{ do } stmt \]
\[ \text{begin } opt\_stmts \text{ end} \]

\[ opt\_stmts \rightarrow stmt ; opt\_stmts \]
\[ \varepsilon \]
The Structure of the Front-End

Source Program (Character stream) → Lexical analyzer → Syntax-directed translator → Intermediate representation

Develop parser and code generator for translator

Syntax definition (BNF grammar) → IR specification
Syntax-Directed Translation

- Uses a CF grammar to specify the syntactic structure of the language
- AND associates a set of attributes with the terminals and nonterminals of the grammar
- AND associates with each production a set of semantic rules to compute values of attributes
- A parse tree is traversed and semantic rules applied: after the tree traversal(s) are completed, the attribute values on the nonterminals contain the translated form of the input
Synthesized and Inherited Attributes

• An attribute is said to be …
  – \textit{synthesized} if its value at a parse-tree node is determined from the attribute values at the children of the node
  – \textit{inherited} if its value at a parse-tree node is determined by the parent (by enforcing the parent’s semantic rules)
Example Attribute Grammar
(Postfix Form)

Production

expr → expr₁ + term
expr → expr₁ - term
expr → term

term → 0
term → 1
...

term → 9

Semantic Rule

expr.t := expr₁.t // term.t // “+”
expr.t := expr₁.t // term.t // “-”
expr.t := term.t
term.t := “0”
term.t := “1”
...

term.t := “9”
Example Annotated Parse Tree

```
expr.t = "95-2+
  expr.t = "95-"
    expr.t = "9"
    term.t = "5"
    term.t = "9"
  term.t = "2"
```

9 - 5 + 2
Depth-First Traversals

procedure visit(n : node);
begin
  for each child \( m \) of \( n \), from left to right do
    visit(m);
  evaluate semantic rules at node \( n \)
end
Depth-First Traversals (Example)

Note: all attributes are of the synthesized type.
Translation Schemes

- A translation scheme is a CF grammar embedded with semantic actions

\[
rest \rightarrow + \ term \{ \text{print(“+”)} \} \ rest
\]

Embedded semantic action
Example Translation Scheme for Postfix Notation

\[
\begin{align*}
expr & \rightarrow expr + term & \{ \text{print("+") } \} \\
expr & \rightarrow expr - term & \{ \text{print("-") } \} \\
expr & \rightarrow term \\
term & \rightarrow 0 & \{ \text{print("0") } \} \\
term & \rightarrow 1 & \{ \text{print("1") } \} \\
\cdots & & \cdots \\
term & \rightarrow 9 & \{ \text{print("9") } \}
\end{align*}
\]
Example Translation Scheme (cont’d)

Translates $9-5+2$ into postfix $95-2+$
Parsing

• Parsing = *process of determining if a string of tokens can be generated by a grammar*

• For any CF grammar there is a parser that takes at most $O(n^3)$ time to parse a string of $n$ tokens

• Linear algorithms suffice for parsing programming language source code

• *Top-down parsing* “constructs” a parse tree from root to leaves

• *Bottom-up parsing* “constructs” a parse tree from leaves to root
Predictive Parsing

• *Recursive descent parsing* is a top-down parsing method
  – Each nonterminal has one (recursive) procedure that is responsible for parsing the nonterminal’s syntactic category of input tokens
  – When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information

• *Predictive parsing* is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations
Example Predictive Parser (Grammar)

\[
\begin{align*}
  \text{type} & \rightarrow \text{simple} \\
  & \mid ^ \text{id} \\
  & \mid \text{array} [ \text{simple} ] \text{ of type} \\
  \text{simple} & \rightarrow \text{integer} \\
  & \mid \text{char} \\
  & \mid \text{num dotdot num}
\end{align*}
\]
Example Predictive Parser
(Program Code)

procedure match(t : token);
begin
  if lookahead = t then
    lookahead := nexttoken()
  else error()
end;

procedure type();
begin
  if lookahead in { ‘integer’, ‘char’, ‘num’ } then
    simple()
  else if lookahead = ‘^’ then
    match( ‘^’ ); match(id)
  else if lookahead = ‘array’ then
    match( ‘array’ ); match( [ ‘); simple();
    match( ‘]’ ); match( ‘of’ ); type()
  else error()
end;

procedure simple();
begin
  if lookahead = ‘integer’ then
    match( ‘integer’ )
  else if lookahead = ‘char’ then
    match( ‘char’ )
  else if lookahead = ‘num’ then
    match( ‘num’ );
    match( ‘dotdot’ );
    match( ‘num’ )
  else error()
end;
Example Predictive Parser
(Execution Step 1)

Input: array [ num dotdot num ] of integer

Check lookahead and call match

\textit{match(‘array’)}

\texttt{type()}
Example Predictive Parser (Execution Step 2)

Input: array [ num dotdot num ] of integer

match ('array') match ('[') type()
Example Predictive Parser
(Execution Step 3)

Input: array [ num dotdot num ] of integer
Example Predictive Parser
(Execution Step 4)

\[
type() \\
match(\text{`array`}) match(\text{`[`}) simple() \\
match(\text{`num`}) match(\text{`dotdot`})
\]

Input: array [ num dotdot num ] of integer
Example Predictive Parser
(Execution Step 5)

Input: array [ num dotdot num ] of integer

\[
\text{type()}
\]
\[
\text{match('array')} \text{ match('[') simple()}
\]
\[
\text{match('num')} \text{ match('dotdot')} \text{ match('num')}
\]

lookahead
Example Predictive Parser
(Execution Step 6)

```
Input: array [ num dotdot num ] of integer
```

```
match('array') match('[') simple() match(']')
```

```
match('num') match('dotdot') match('num')
```

```
type()
```

```
lookahead
```
Example Predictive Parser
(Execution Step 7)

```
match('array') match(']') simple() match(']') match('of')

match('num') match('dotdot') match('num')
```

Input: array [ num dotdot num ] of integer
Example Predictive Parser
(Execution Step 8)

Input: array [ num dotdot num ] of integer

lookahead
FIRST

FIRST(\(\alpha\)) is the set of terminals that appear as the first symbols of one or more strings generated from \(\alpha\)

\[
\begin{align*}
type & \rightarrow simple \\
& \mid ^\wedge id \\
& \mid array [ simple ] of type \\
simple & \rightarrow integer \\
& \mid char \\
& \mid num dotdot num
\end{align*}
\]

FIRST(simple) = \{ integer, char, num \} 
FIRST(^ id) = \{ ^ \} 
FIRST(type) = \{ integer, char, num, ^, array \}
How to use FIRST

We use FIRST to write a predictive parser as follows

\[
\begin{align*}
expr &\to \text{term rest} \\
\text{rest} &\to + \text{term rest} \\
&\mid - \text{term rest} \\
&\mid \varepsilon
\end{align*}
\]

\[
\text{procedure rest();}
\begin{align*}
\quad &\text{begin} \\
&\quad \text{if lookahead in \text{FIRST}(+ \text{term rest}) then} \\
&\quad \quad \text{match(‘+’); term(); rest()} \\
&\quad \text{else if lookahead in \text{FIRST}(- \text{term rest}) then} \\
&\quad \quad \text{match(‘-’); term(); rest()} \\
&\quad \text{else return} \\
&\quad \text{end;}
\end{align*}
\]

When a nonterminal \( A \) has two (or more) productions as in

\[
A \to \alpha \\
\mid \beta
\]

Then FIRST(\( \alpha \)) and FIRST(\( \beta \)) must be disjoint for predictive parsing to work.
Left Factoring

When more than one production for nonterminal $A$ starts with the same symbols, the FIRST sets are not disjoint

$$stmt \rightarrow \text{if expr then stmt endif}$$
$$\| \text{if expr then stmt else stmt endif}$$

We can use left factoring to fix the problem

$$stmt \rightarrow \text{if expr then stmt opt\_else}$$
$$\text{opt\_else} \rightarrow \text{else stmt endif}$$
$$\| \text{endif}$$
Left Recursion

When a production for nonterminal $A$ starts with a self reference then a predictive parser loops forever

$$A \rightarrow A \alpha$$
$$\mid \beta$$
$$\mid \gamma$$

We can eliminate left recursive productions by systematically rewriting the grammar using right recursive productions

$$A \rightarrow \beta R$$
$$\mid \gamma R$$
$$R \rightarrow \alpha R$$
$$\mid \epsilon$$
A Translator for Simple Expressions

\[ expr \rightarrow expr + term \quad \{ \text{print(“+”)} \} \]
\[ expr \rightarrow expr - term \quad \{ \text{print(“-”)} \} \]
\[ expr \rightarrow term \]
\[ term \rightarrow 0 \quad \{ \text{print(“0”)} \} \]
\[ term \rightarrow 1 \quad \{ \text{print(“1”)} \} \]
\[ \ldots \]
\[ term \rightarrow 9 \quad \{ \text{print(“9”)} \} \]

After left recursion elimination:

\[ expr \rightarrow term \ rest \]
\[ rest \rightarrow + term \quad \{ \text{print(“+”)} \} \ rest \]
\[ rest \rightarrow - term \quad \{ \text{print(“+”)} \} \ rest \]
\[ rest \rightarrow \varepsilon \]
\[ term \rightarrow 0 \quad \{ \text{print(“0”)} \} \]
\[ term \rightarrow 1 \quad \{ \text{print(“1”)} \} \]
\[ \ldots \]
\[ term \rightarrow 9 \quad \{ \text{print(“9”)} \} \]
Code of the translator

expr $\rightarrow$ term rest

rest $\rightarrow$ + term { print("+") } rest
rest $\rightarrow$ - term { print("-") } rest
rest $\rightarrow$ $\varepsilon$

term $\rightarrow$ 0 { print("0") }
term $\rightarrow$ 1 { print("1") }
...
term $\rightarrow$ 9 { print("9") }

main()
{    lookahead = getchar();
    expr();
}
expr()
{    term(); rest(); }
rest()
{    if (lookahead == '+')
        {match('+'); term(); putchar('+'); rest();}
    else if (lookahead == '-')
        {match('-'); term(); putchar('-'); rest();}
    else {};
}
term()
{    if (isdigit(lookahead))
        {    printf("0"); match(lookahead);
            }
    else error();
}
match(int t)
{    if (lookahead == t)
        {lookahead = getchar();
         }else error();
}
error()
{    printf("Syntax error\n");
    exit(1);
}
main()
{
    lookahead = getchar();
    expr();
}
expr()
{
    term();
    while (1) /* optimized by inlining rest() and removing recursive calls */
    {
        if (lookahead == '+')
        {
            match('+'); term(); putchar('+');
        }
        else if (lookahead == '-')
        {
            match('-'); term(); putchar('-');
        }
        else break;
    }
}
term()
{
    if (isdigit(lookahead))
    {
        putchar(lookahead); match(lookahead);
    }
    else error();
}
match(int t)
{
    if (lookahead == t)
    {
        lookahead = getchar();
        else error();
    }
}
error()
{
    printf("Syntax error\n");
    exit(1);
}