Compilers: Principles, Techniques, and Tools

Chapter 2 A Simple Syntax-Directed Translator

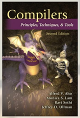
Dongjie He

University of New South Wales

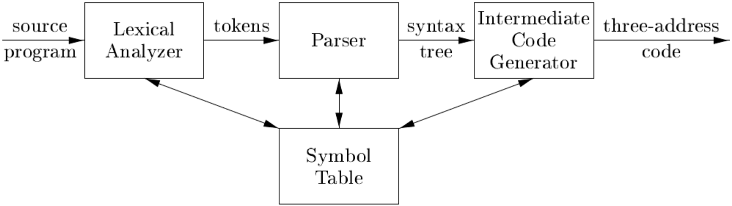
https://dongjiehe.github.io/teaching/compiler/

29 Jun 2023

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY · AUSTRALIA



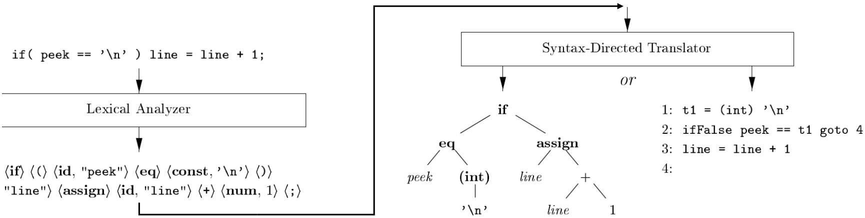
Simple Syntax-Directed Translators



- A *lexical analyzer* groups multicharacter constructs as **tokens**
- **syntax**: describes the proper form of the program
- **semantics**: defines what the program mean
- **intermediate code**: *abstract syntax trees* or *three-address code*

- **4 Tasks** $\longrightarrow$ **syntax-directed translation** $\longrightarrow$ **context-free grammars** or BNF (Backus-Naur Form)

- **Task 1**: translate arithmetic expressions from infix into prefix
  - e.g., "9+5*2" $\longrightarrow$ "+9*52"
- **Task 2**: scan basic tokens like numbers and identifiers
  - e.g., `count = count + increment;`
  - $\langle \mathbf{id}, \text{``}\texttt{count}\text{''} \rangle$ $\langle = \rangle$ $\langle \mathbf{id}, \text{``}\texttt{count}\text{''} \rangle$ $\langle + \rangle$ $\langle \mathbf{id}, \text{``}\texttt{incremental}\text{''} \rangle$ $\langle ; \rangle$
- **Task 3**: application of symbol tables
  - translate `{ int x; char y; { bool y; x; y; } x; y; }`
  - into `{ { x:int; y:bool; } x:int; y:char; }`
- **Task 4**: translate a source program into intermediate representations



Review: Context Free Grammar $G = (V, \Sigma, P, s)$

- Σ: a set of ***terminal*** symbols, or "tokens" in PL
- V: a set of ***nonterminals***, or "syntactic variables"
- $s \in V$ is the *start* nonterminal symbol
- P: a set of ***productions*** in form of $\alpha \to \beta$
  - $\alpha \in V$ and $\beta \in (V \cup \Sigma)^*$ (implying β could be ϵ)

- **Example 1** expressions consisting of digits, plus and minus signs

$$list \to list + digit \mid list - digit \mid digit$$
$$digit \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

  - **derivation**: derive strings from s, e.g., 9-5+2

$$list \to list + digit \to list + 2 \to list - digit + 2$$
$$\to list - 5 + 2 \to digit - 5 + 2 \to 9 - 5 + 2$$

Review: Context Free Grammar $G = (V, \Sigma, P, s)$

- Σ: a set of ***terminal*** symbols, or "tokens" in PL
- V: a set of ***nonterminals***, or "syntactic variables"
- $s \in V$ is the *start* nonterminal symbol
- P: a set of ***productions*** in form of $\alpha \to \beta$
  - $\alpha \in V$ and $\beta \in (V \cup \Sigma)^*$ (implying β could be ϵ)

- **Example 2** expresses a subset of Java statements

$$\begin{aligned} \mathit{stmt} \to\ & \mathtt{id} = \mathit{expr}\ ; \mid \mathtt{if}\ (\ \mathit{expr}\ )\ \mathit{stmt} \mid \mathtt{if}\ (\ \mathit{expr}\ )\ \mathit{stmt}\ \mathtt{else}\ \mathit{stmt} \\ & \mid \textbf{\textit{while}}\ (\ \mathit{expr}\ )\ \mathit{stmt} \mid \mathtt{do}\ \mathit{stmt}\ \mathtt{while}\ (\ \mathit{expr}\ )\ ; \mid \{\ \mathit{stmts}\ \} \end{aligned}$$

$$\mathit{stmts} \to \mathit{stmts}\ \mathit{stmt} \mid \epsilon \qquad \mathit{expr} \to \cdots$$

- **Example 3** expresses function calls, e.g., `max(x, y)`

$$\mathit{call} \to \mathtt{id}\ (\ \mathit{optparams}\ ) \qquad \mathit{optparams} \to \mathit{params} \mid \epsilon$$

$$\mathit{params} \to \mathit{params}\ ,\ \mathit{param} \mid \mathit{param} \qquad \mathit{param} \to \cdots$$

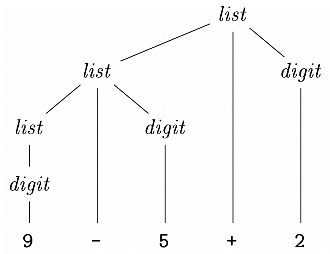
Review: Context Free Grammar $G = (V, \Sigma, P, s)$

- ***parse tree***: a tree pictorially shows how s derives a string in L_G
  - *root* labeled by s
  - *leaf* labeled by a terminal or ϵ
  - *interior node* labeled by a non-terminal
  - given an interior node N with label A, let $X_1, X_2, \cdots, X_n$ be the labels of N's children node from left to right, then $A \to X_1\ X_2\ \cdots\ X_n \in P$.
- a derivation $\Longleftrightarrow$ a parse tree, e.g., '9-5+2'

- the derivation of '9-5+2'

$$
\begin{aligned}
list &\to list + digit \\
&\to list + 2 \\
&\to list - digit + 2 \\
&\to list - 5 + 2 \\
&\to digit - 5 + 2 \\
&\to 9 - 5 + 2
\end{aligned}
$$

- the parse tree of '9-5+2'



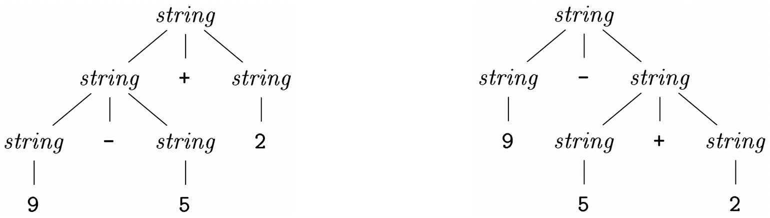
Task 1: translate arithmetic expressions from infix into prefix, e.g., "9+5*2" $\longrightarrow$ "+9*52".

- Step 1: construct a grammar to describe arithmetic expressions

How about this grammar?

$$\begin{aligned} \textit{string} \rightarrow\ & \textit{string} + \textit{string} \mid \textit{string} - \textit{string} \\ \mid\ & \textit{string} * \textit{string} \mid \textit{string} / \textit{string} \\ \mid\ & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

The grammar is *ambiguous*, e.g., '9-5+2' has two parse trees.



Associativity of Operators

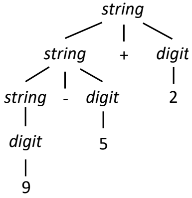
- an operand *x* with operator *op* on both sides,
  - **right-associative** if *x* belongs to the right *op*, e.g., `a = b = c`
  - **left-associative** if *x* belongs to the left *op*, e.g., +, -, *, /
- different associativity requires different grammar
  - grammar for the **right-associative** example `a = b = c`

$$right \rightarrow letter = right \mid letter$$
$$letter \rightarrow a \mid b \mid \;\cdots \mid z$$

  - grammar for the **left-associative** example, +, -, *, /

$$\begin{aligned} string \rightarrow\ & string + digit \mid string - digit \\ & \mid string * digit \mid string / digit \\ & \mid digit \\ digit \rightarrow\ & 0 \mid 1 \mid 2 \mid 3 \mid 4 \\ & \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

new parse tree for '9-5+2'



How about this grammar?

Associativity of Operators

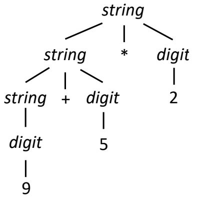
- an operand *x* with operator *op* on both sides,
  - **right-associative** if *x* belongs to the right *op*, e.g., `a = b = c`
  - **left-associative** if *x* belongs to the left *op*, e.g., $+$, -, *, $/$
- different associativity requires different grammar
  - grammar for the **right-associative** example `a = b = c`

$$right \rightarrow letter = right \mid letter$$
$$letter \rightarrow a \mid b \mid \ \cdots \mid z$$

  - grammar for the **left-associative** example, $+$, -, *, $/$

$$\begin{aligned} string \rightarrow\ & string + digit \mid string - digit \\ & \mid string * digit \mid string / digit \\ & \mid digit \\ digit \rightarrow\ & 0 \mid 1 \mid 2 \mid 3 \mid 4 \\ & \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Unwanted parse tree for '9+5*2'



(9+5)*2?

Precedence of Operators

- op_1 has ***higher precedence*** than op_2 if op_1 takes its operands before op_2
  - * and / have higher precedence than + and -
- grammar supports associativity and precedence of operators
  - support operators of lower precedence, i.e., +, -

$$expr \rightarrow expr + term \mid expr - term \mid term$$

  - support operators of higher precedence, i.e., *, /

$$term \rightarrow term * factor \mid term / factor \mid factor$$

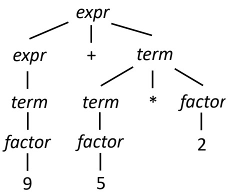
  - generate basic units in expressions

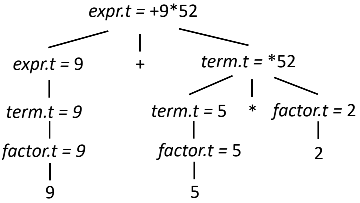
$$factor \rightarrow digit \mid (\ expr\ )$$
$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- One can generalize the idea to support any precedence levels (p50)

Task 1: translate arithmetic expressions from infix into prefix, e.g., "9+5*2" $\longrightarrow$ "+9*52".

- Step 2: present *syntax-directed definition* for the grammar
  - associate each ***grammar symbol*** with a set of **attributes**





  - associate each ***production*** with a set of **semantic rules**

productions	semantic rules	production	semantic rules
$expr \rightarrow expr_1 + term$	$expr.t = + \mathbin{\Vert} expr_1.t \mathbin{\Vert} term.t$	$expr \rightarrow term$	$expr.t = term.t$
$expr \rightarrow expr_1 - term$	$expr.t = - \mathbin{\Vert} expr_1.t \mathbin{\Vert} term.t$	$term \rightarrow factor$	$term.t = factor.t$
$term \rightarrow term_1 * factor$	$term.t = * \mathbin{\Vert} term_1.t \mathbin{\Vert} factor.t$	$factor \rightarrow digit$	$factor.t = digit.t$
$term \rightarrow term_1 / factor$	$term.t = / \mathbin{\Vert} term_1.t \mathbin{\Vert} factor.t$	$factor \rightarrow (\ expr \ )$	$factor.t = expr.t$

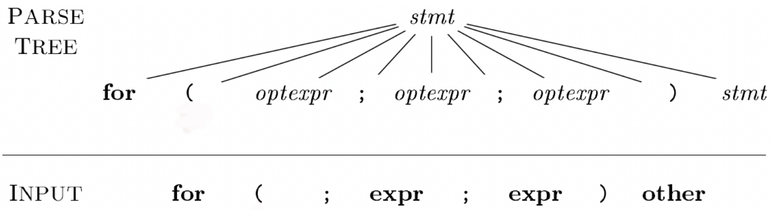
Task 1: translate arithmetic expressions from infix into prefix, e.g., "9+5*2" $\longrightarrow$ "+9*52".

- Step 3: specify the evaluation order of the semantic rules (i.e., define a ***syntax-directed translation scheme***)
  - embed *semantic actions* (program fragments) within production bodies

$$
\begin{aligned}
expr &\rightarrow expr_1 + term \ \{\ expr.t = + \ || \ expr_1.t \ || \ term.t \ \} \\
expr &\rightarrow expr_1 - term \ \{\ expr.t = - \ || \ expr_1.t \ || \ term.t \ \} \\
term &\rightarrow term_1 * factor \ \{\ term.t = * \ || \ term_1.t \ || \ factor.t \ \} \\
term &\rightarrow term_1 / factor \ \{\ term.t = / \ || \ term_1.t \ || \ factor.t \ \} \\
expr &\rightarrow term \ \{\ expr.t = term.t \ \} \\
term &\rightarrow factor \ \{\ term.t = factor.t \ \} \\
factor &\rightarrow digit \ \{\ factor.t = digit.t \ \} \\
factor &\rightarrow (\ expr\ ) \ \{\ factor.t = expr.t \ \}
\end{aligned}
$$

Task 1: translate arithmetic expressions from infix into prefix, e.g., "9+5*2" $\longrightarrow$ "+9*52".

- Step 4: build a parser for translation
  - ***parsing***: construct a parser tree for the string of input terminals
  - ***top-down parsing***: start from root labeled with non-terminal *s* and repeatedly peform following two steps
    - find a node *N* (labeled with *A*) at which a subtree is to be constructed
    - select a production for *A* to construct children at *N*
  - e.g., `for ( ; expr ; expr ) other`



- Step 4: build a parser for translation
  - ***parsing***: construct a parser tree for the string of input terminals
  - ***top-down parsing***: start from root labeled with non-terminal s and repeatedly peform following two steps
    - find a node N (labeled with A) at which a subtree is to be constructed
    - select a production for A to construct children at N
  - ***recursive-descent parsing***:
    - a kind of top-down parsing in which a set of recursive procedures is used to process the input
    - each non-terminal in the grammar is associated with a procedures
  - ***predictive parsing***:
    - a simple form of *recursive-descent parsing*
    - *lookahead* symbol unambiguously determines the next production
    - e.g., "$lookahead = \textbf{for}$" $\Rightarrow$ $stmt \rightarrow \texttt{for}\ (\ optexpr\ ;\ optexpr\ ;\ optexpr\ )\ stmt$
    - mimic the body of the chosen production

$$match(\texttt{for});\ match('(');$$
$$optexpr();\ match(';');\ optexpr();\ match(';');\ optexpr();$$
$$match(')');\ stmt();$$

    - add actions into procedures for translation

- Step 4: build a parser for translation
  - consider $expr \rightarrow expr + term$:

$$expr()\{\ expr();\ \ match('+');\ \ term();\}$$

  - eliminate ***left recursion*** by grammar rewriting

$$\begin{array}{ccc} A \rightarrow A\alpha & & A \rightarrow \beta R \\ & \Longrightarrow & \\ A \rightarrow \beta & & R \rightarrow \alpha R \mid \epsilon \end{array}$$

Grammar for Infix Expression

$$\begin{aligned} expr &\rightarrow expr + term \\ &\mid expr - term \mid term \\ term &\rightarrow term * factor \\ &\mid term / factor \mid factor \\ factor &\rightarrow digit \mid (\ expr\ ) \end{aligned}$$

Grammar for Infix Expression without Recursion

$$\begin{aligned} expr &\rightarrow term\ rexpr \\ rexpr &\rightarrow +\ term\ rexpr \mid -\ term\ rexpr \mid \epsilon \\ term &\rightarrow factor\ rterm \\ rterm &\rightarrow *\ factor\ rterm \mid /\ factor\ rterm \mid \epsilon \\ factor &\rightarrow digit \mid (\ expr\ ) \end{aligned}$$

- Step 4: build a parser for translation

  - *expr* → *term rexpr*

```
Expr expr() {
  Term t = term();
  Rexpr re = rexpr();
  Expr expr = new Expr(t, re);
  expr.attr = re.op + t.attr + re.attr;
  return expr;
}
```

  - other procedure (e.g., term(), rterm(), factor()) can be implemented similarly.

  - *rexpr* → + *term rexpr* | − *term rexpr* | ϵ

```
Rexpr rexpr() { Rexpr r = new Rexpr();
  if (lookahead == '+'||lookahead == '-'){
      r.op = String.valueOf((char)lookahead);
      match(lookahead);
      Term t = term();
      Rexpr re = rexpr();
      r.attr = re.op + t.attr + re.attr;
  }
  return r;
}
```

- A link to the complete program:

  `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/Infix2Prefix`

Play a Demo!

Task 2: scan basic tokens like numbers and identifiers, e.g., "`cnt = cnt + inc;`" $\Rightarrow$ " $\langle \textbf{id}, \text{``}\texttt{cnt}\text{''}\rangle$ $\langle = \rangle$ $\langle \textbf{id}, \text{``}\texttt{cnt}\text{''}\rangle$ $\langle + \rangle$ $\langle \textbf{id}, \text{``}\texttt{inc}\text{''}\rangle$ $\langle ; \rangle$".

Scanner Sketch/Pseudocode

```
Token scan() {
  Step 1: skip white space and comments
  Step 2: handle numbers
  Step 3: handle reserved words and identifiers
  /*if we get here, treat read-ahead character peek as a token*/
  Token t = new Token(peek);
  peek = blank /*initialization*/
  return t;
}
```

- *peek*: hold next input for deciding on the token to be returned.
- reads ahead only when it must, otherwise, *peek* is set to a blank.

Task 2: scan basic tokens like numbers and identifiers, e.g., "`cnt = cnt + inc;`" $\Rightarrow$ " $\langle \textbf{id}, \text{``cnt''} \rangle$ $\langle = \rangle$ $\langle \textbf{id}, \text{``cnt''} \rangle$ $\langle + \rangle$ $\langle \textbf{id}, \text{``inc''} \rangle$ $\langle ; \rangle$".

- Step 1: skip white space and comments
  - skip while space

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

  - skipping comments is leaved as an assgnment.
    - "// single line comments"
    - "/* multiple lines comments */ "

Task 2: scan basic tokens like numbers and identifiers, e.g., "`cnt = cnt + inc;`" $\Rightarrow$ " $\langle$**id**, "`cnt`"$\rangle$ $\langle=\rangle$ $\langle$**id**, "`cnt`"$\rangle$ $\langle+\rangle$ $\langle$**id**, "`inc`"$\rangle$ $\langle;\rangle$".

- Step 2: handle numbers

e.g., "31 + 28 + 59" $\Rightarrow$ "$\langle \textbf{num}, 31\rangle$ $\langle+\rangle$ $\langle \textbf{num}, 28\rangle$ $\langle+\rangle$ $\langle \textbf{num}, 59\rangle$ "

```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

Think about how to support **float**.

Task 2: scan basic tokens like numbers and identifiers, e.g., "`cnt = cnt + inc;`" ⇒ " $\langle \textbf{id}, \text{“cnt”}\rangle$ $\langle = \rangle$ $\langle \textbf{id}, \text{“cnt”}\rangle$ $\langle + \rangle$ $\langle \textbf{id}, \text{“inc”}\rangle$ $\langle ; \rangle$".

- Step 3: handle reserved words and identifiers
  - *Hashtable words* = **new** *Hashtable*()
  - set up reserved key words in *words*

```
if ( peek holds a letter ) {
    collect letters or digits into a buffer b;
    s = string formed from the characters in b;
    w = token returned by words.get(s);
    if ( w is not null ) return w;
    else {
        Enter the key-value pair (s, ⟨id, s⟩) into words
        return token ⟨id, s⟩;
    }
}
```

> **Task 2:** scan basic tokens like numbers and identifiers, e.g., "`cnt = cnt + inc;`" $\Rightarrow$ " $\langle \textbf{id}, \text{“}\texttt{cnt}\text{”}\rangle$ $\langle = \rangle$ $\langle \textbf{id}, \text{“}\texttt{cnt}\text{”}\rangle$ $\langle + \rangle$ $\langle \textbf{id}, \text{“}\texttt{inc}\text{”}\rangle$ $\langle ; \rangle$".

- A link to the complete program

  `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/Lexer`

Play a Demo!

Task 3: An application of symbol tables by translating
"`{ int x; char y; { bool y; x; y; } x; y; }`" into
"`{ { x:int; y:bool; } x:int; y:char; }`"

- Grammar for the source program

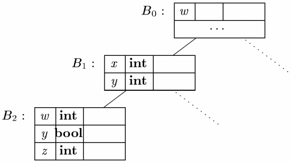
$$\textit{program} \rightarrow \textit{block} \qquad \textit{block} \rightarrow \{\ \textit{decls stmts}\ \}$$

$$\textit{decls} \rightarrow \textit{decls decl} \mid \epsilon \qquad \textit{decl} \rightarrow \textbf{type id}\ ;$$

$$\textit{stmts} \rightarrow \textit{stmts stmt} \mid \epsilon \qquad \textit{factor} \rightarrow \textbf{id} \qquad \textit{stmt} \rightarrow \textit{block} \mid \textit{factor};$$

- *Most-closely nested* rule: an identier x is in the scope of the most-closely nested declaration of x
- One Symbol Table per Scope $\Rightarrow$ chained symbol tables

$$
\begin{array}{l}
\{ \quad \textbf{int}\ x_1;\ \textbf{int}\ y_1; \\
\quad \{ \quad \textbf{int}\ w_2;\ \textbf{bool}\ y_2;\ \textbf{int}\ z_2; \\
\qquad \cdots w_2 \cdots;\ \cdots x_1 \cdots;\ \cdots y_2 \cdots;\ \cdots \\
\quad \} \\
\quad \cdots w_0 \cdots;\ \cdots x_1 \cdots;\ \cdots y_1 \cdots; \\
\}
\end{array}
$$



Task 3: An application of symbol tables by translating
"`{ int x; char y; { bool y; x; y; } x; y; }`" into
"`{ { x:int; y:bool; } x:int; y:char; }`"

- A Java implementation of chained symbol tables **Env**

```
class Env {
  private Hashtable table;
  protected Env prev;
  public Env(Env p) {
    table = new Hashtable(); prev = p;
  }
  public void put(String s, Symbol sym) {
    table.put(s, sym);
  }
  public Symbol get(String s) {
    Symbol found = e.table.get(s);
     if (found != null) return found;
     if (e.prev != null) return e.prev.get(s);
    return null;
}}
```

Task 3: An application of symbol tables by translating
"`{ int x; char y; { bool y; x; y; } x; y; }`" into
"`{ { x:int; y:bool; } x:int; y:char; }`"

- Translation Scheme

$$
\begin{aligned}
\mathit{program} \rightarrow & \{\mathit{top} = \mathit{null};\}\mathit{block} \\
\mathit{block} \rightarrow & \{\mathit{saved} = \mathit{top}; \mathit{top} = \textbf{new } \mathit{Env}(\mathit{top}); \mathit{print}(``\{"); \} \\
& \mathit{decls}\ \mathit{stmts}\ \}\{\mathit{top} = \mathit{saved}; \mathit{print}(``\}"); \} \\
\mathit{decls} \rightarrow & \mathit{decls}\ \mathit{decl} \mid \epsilon \\
\mathit{decl} \rightarrow & \textbf{type}\ \textbf{id}\ ; \{s = \textbf{new } \mathit{Symbol}; \\
& s.\mathit{type} = \textbf{type}.\mathit{lexeme}; \mathit{top}.\mathit{put}(\textbf{id}.\mathit{lexeme}, s); \} \\
\mathit{stmts} \rightarrow & \mathit{stmts}\ \mathit{stmt} \mid \epsilon \\
\mathit{stmt} \rightarrow & \mathit{block} \mid \mathit{factor}; \mathit{print}(``;"); \\
\mathit{factor} \rightarrow & \textbf{id}\{s = \mathit{top}.\mathit{get}(\textbf{id}.\mathit{lexeme}); \mathit{print}(\textbf{id}.\mathit{lexeme}); \\
& \mathit{print}(``:"); \mathit{print}(s.\mathit{type}); \}
\end{aligned}
$$

Task 3: An application of symbol tables by translating
"`{ int x; char y; { bool y; x; y; } x; y; }`" into
"`{ { x:int; y:bool; } x:int; y:char; }`"

- A link to the complete program

  `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/SymbolTable`

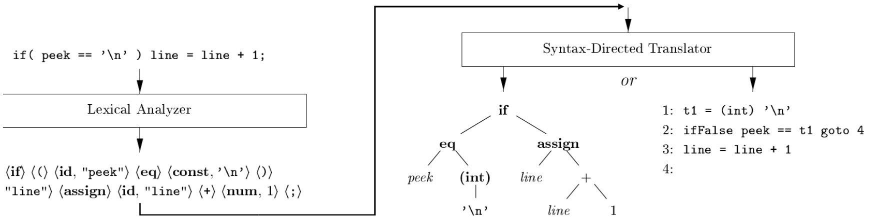
Play a Demo!

Task 4: translate a source program into intermediate representations

- Grammar for the source program

program →	*block*	*block* →	*stmts*
stmts →	*stmts stmt* | ϵ	*stmt* →	*expr* ; | *block*
stmt →	**if** (*expr*) *stmt*	*stmt* →	**while** (*expr*) *stmt*
stmt →	**do** *stmt* **while** (*expr*) ;	*expr* →	*rel* = *expr* | *rel*
rel →	*add* < *add* | *add*	*add* →	*add* + *term* | *term*
term →	*term* ∗ *factor* | *factor*	*factor* →	(*expr*) | **num**

- Two kinds of IR: syntax tree & three-address code



Task 4-1: construct the syntax tree of a source program

- The translation scheme of constructing the syntax tree

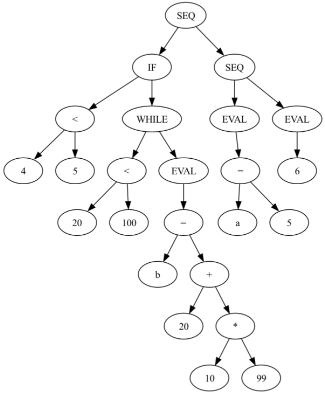
$$
\begin{aligned}
&\mathit{program} \to \mathit{block}\{\ \mathit{return\ block\ n};\ \} \\
&\mathit{block} \to \{\mathit{stmts}\}\quad \{\ \mathit{block.n} = \mathit{stmts.n};\ \} \\
&\mathit{stmts} \to \epsilon\quad \{\ \mathit{stmts.n} = \textbf{null};\ \} \\
&\qquad \mid \mathit{stmts}_1\ \mathit{stmt}\quad \{\ \mathit{stmts.n} = \textbf{new}\ \mathit{Seq}(\mathit{stmts}_1.n, \mathit{stmt.n});\ \} \\
&\mathit{stmt} \to \mathit{block}\quad \{\ \mathit{stmt.n} = \mathit{block.n};\ \} \\
&\qquad \mid \mathit{expr}\quad \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{Eval}(\mathit{expr.n});\ \}; \\
&\qquad \mid \textbf{if}\ (\ \mathit{expr}\ )\ \mathit{stmt}_1\quad \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{If}(\mathit{expr.n}, \mathit{stmt}_1.n);\ \} \\
&\qquad \mid \textbf{while}\ (\ \mathit{expr}\ )\ \mathit{stmt}_1\quad \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{While}(\mathit{expr.n}, \mathit{stmt}_1.n);\ \} \\
&\qquad \mid \textbf{do}\ \mathit{stmt}_1\ \textbf{while}\ (\ \mathit{expr}\ )\ ;\quad \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{Do}(\mathit{stmt}_1.n, \mathit{expr.n});\ \} \\
&\mathit{expr} \to \mathit{rel}\quad \{\ \mathit{expr.n} = \mathit{rel.n};\ \} \\
&\qquad \mid \mathit{rel} = \mathit{expr}_1\quad \{\ \mathit{expr.n} = \textbf{new}\mathit{Assign}('=', \mathit{rel.n}, \mathit{expr}_1.n);\ \} \\
&\mathit{rel} \to \mathit{add}\quad \{\ \mathit{rel.n} = \mathit{add.n};\ \} \\
&\qquad \mid \mathit{add}_1 < \mathit{add}_2\quad \{\ \mathit{rel.n} = \textbf{Rel}('<', \mathit{add}_1.n, \mathit{add}_2.n);\ \} \\
&\mathit{add} \to \mathit{term}\quad \{\ \mathit{add.n} = \mathit{term.n};\ \} \\
&\qquad \mid \mathit{add}_1 + \mathit{term}\quad \{\ \mathit{add.n} = \textbf{new}\ \mathit{Op}('+', \mathit{add}_1.n, \mathit{term.n});\ \} \\
&\mathit{term} \to \mathit{factor}\quad \{\ \mathit{term.n} = \mathit{factor.n};\ \} \\
&\qquad \mid \mathit{term}_1 * \mathit{factor}\quad \{\ \mathit{term.n} = \textbf{new}\ \mathit{Op}('*', \mathit{term}_1.n, \mathit{factor.n});\ \} \\
&\mathit{factor} \to (\ \mathit{expr}\ )\quad \{\mathit{factor.n} = \mathit{expr.n};\} \\
&\qquad \mid \textbf{num}\quad \{\mathit{factor.n} = \textbf{new}\ \mathit{Num}(\mathit{num.value});\} \\
&\qquad \mid \textbf{id}\quad \{\mathit{factor.n} = \textbf{new}\ \mathit{Identifier}(\mathit{id.lexeme})\}
\end{aligned}
$$

- A link to the complete program

`https://github.com/DongjieHe/cptt/tree/main/assigns/a2/SyntaxTree`

Task 4-1: construct the syntax tree of a source program

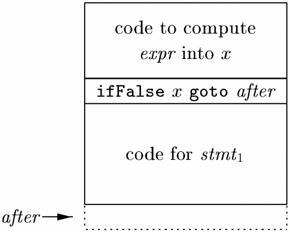
- Example: “{ if (4 < 5) { while (20 < 100) b = 20 + 10 * 99; } { a = 5; 6 ; } }”



Play a Demo!

Task 4-2: generate three-address code for a source program

- Normal form: $x = y$ **op** z
  - l-values: locations on the left side of an assignment
  - r-value: values of the right side of an ssignment
- Array load (store): $x[y] = z$ $(x = y[z])$
- Control Flow: **if**, **while**, **do while**, **for**, ...
  - `ifFalse` x `goto` L
  - `ifTrue` x `goto` L
  - `goto` L




```
class If extends Stmt {
        Expr E; Stmt S;
        public If(Expr x, Stmt y) { E = x; S = y; after = newlabel()
        public void gen() {
                Expr n = E.rvalue();
                emit( "ifFalse " + n.toString() + " goto " + after);
                S.gen();
                emit(after + ":");
        }
}
```

Task 4-2: generate three-address code for a source program

- Part of the Translation Scheme

$$
\begin{aligned}
program &\rightarrow \{block.s = program.s;\ block.e = program.e;\}\quad block \\
block &\rightarrow \{stmts.s = block.s;\ stmts.e = block.e;\}\quad \{stmts\} \\
stmts &\rightarrow \epsilon \mid \{l = freshLabel();\ stmt.s = stmts.s;\ stmt.e = l;\}\quad stmt \\
&\quad \{emit(l);\ stmts_1.s = l;\ stmts_1.e = stmts.e;\}\ stmts_1 \\
stmt &\rightarrow \{\ block.s = stmt.s;\ block.e = stmt.e;\ \}\quad block \\
&\quad \mid loc = expr\ ;\quad \{r = expr.gen();\ print(``" + loc + ``=" + r);\} \\
&\quad \mid \textbf{if}\ (\ expr\ )\ \{r = expr.gen();\ l = freshLabel(); \\
&\quad\ stmt_1.s = l;\ stmt_1.e = stmt.e; \\
&\quad\ print(``ifFalse" + r + ``goto" + stmt.e + ``;");\ emit(l);\}\ stmt_1 \\
&\quad \mid \textbf{while}\ (\ expr\ )\ \{r = expr.gen();\ l = freshLabel(); \\
&\quad\ print(``ifFalse" + r + ``goto" + stmt.e + ``;"); \\
&\quad\ emit(l);\ stmt_1.s = l;\ stmt_1.e = stmt.e;\} \\
&\quad\ stmt_1\ \{print(``goto" + stmt.s);\} \\
&\quad \mid \textbf{do}\ \{l = freshLabel();\ stmt_1.s = stmt.s;\ stmt_1.e = l;\}\ stmt_1\ \textbf{while}\ (\ expr\ )\ ; \\
&\quad\ \{emit(l);\ r = expr.gen();\ print(``if" + r + ``goto" + stmt.s + ``;");\}
\end{aligned}
$$

- A link to the complete program

`https://github.com/DongjieHe/cptt/tree/main/assigns/a2/nutshell`

Task 4-2: generate three-address code for a source program

Play a Demo!

```
{
    int i; int j; float[100] a; float v; float x;

    while ( true ) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if ( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
```

Summary

- Task 1: Infix to Prefix
  - CFL grammar
  - Operator: Associativity and Precedence
  - Predictive Parsing and remove Left Recursion
  - Syntax-directed Translation Scheme
- Task 2: simple scanner
  - skip blank space and comments
  - handle numbers
  - handle reserved words and identifiers
- Task 3: simple type inference
  - symbol table
- Task 4: intermediate code generation
  - syntax tree
  - three-address code

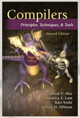
Compilers: Principles, Techniques, and Tools

Chapter 2 A Simple Syntax-Directed Translator

Dongjie He
University of New South Wales
https://dongjiehe.github.io/teaching/compiler/

29 Jun 2023

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY · AUSTRALIA



Lab 2: Get Familiar with Syntax-directed Translation

- Read tasks' implementation.
  - **Task 1**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/Infix2Prefix`
  - **Task 2**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/Lexer`
  - **Task 3**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/SymbolTable`
  - **Task 4-1**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/SyntaxTree`
  - **Task 4-2**: `https://github.com/DongjieHe/cptt/tree/main/assigns/a2/nutshell`
- construct a translator translating arithmetic expression from infix to postfix (hint: refer to Task 1).
- support comment **or** float number in the simple scanner in Task 2.
- support For-statement, i.e., `for` ($expr_1$; $expr_2$; $expr_3$) *stmt* in Task 4-1 **or** Task 4-2.