Compiler Construction

Lecture 11: Syntax Analysis VII (Practical Issues) & Semantic Analysis I (Attribute Grammars)

Summer Semester 2017

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-17/cc/

Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

Outline of Lecture 11

Recap: $LR(1)$ Parsing

$LR(1)$ Items and Sets

Observation: not every element of $\mathrm{fo}(A)$ can follow every occurrence of A
$\Longrightarrow$ refinement of $LR(0)$ items by adding possible lookahead symbols

Definition ($LR(1)$ items and sets)

Let $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma$ be start separated by $S' \to S$.

- If $S' \Rightarrow_r^* \alpha A a w \Rightarrow_r \alpha\beta_1\beta_2 a w$, then $[A \to \beta_1 \cdot \beta_2, a]$ is called an $LR(1)$ item for $\alpha\beta_1$.
- If $S' \Rightarrow_r^* \alpha A \Rightarrow_r \alpha\beta_1\beta_2$, then $[A \to \beta_1 \cdot \beta_2, \varepsilon]$ is called an $LR(1)$ item for $\alpha\beta_1$.
- Given $\gamma \in X^*$, $LR(1)(\gamma)$ denotes the set of all $LR(1)$ items for γ, called the $LR(1)$ set (or: $LR(1)$ information) of γ.
- $LR(1)(G) := \{LR(1)(\gamma) \mid \gamma \in X^*\}$.

$LR(1)$ Conflicts

Definition ($LR(1)$ conflicts)

Let $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma$ and $I \in LR(1)(G)$.

- I has a shift/reduce conflict if there exist $A \to \alpha_1 a \alpha_2, B \to \beta \in P$ and $x \in \Sigma_\varepsilon$ such that

$$[A \to \alpha_1 \cdot a\alpha_2, x], [B \to \beta\cdot, a] \in I.$$

- I has a reduce/reduce conflict if there exist $x \in \Sigma_\varepsilon$ and $A \to \alpha, B \to \beta \in P$ with $A \neq B$ or $\alpha \neq \beta$ such that

$$[A \to \alpha\cdot, x], [B \to \beta\cdot, x] \in I.$$

Lemma

$G \in LR(1)$ iff no $I \in LR(1)(G)$ contains conflicting items.

The $LR(1)$ Action Function

Definition ($LR(1)$ action function)

The $LR(1)$ action function

$$\mathrm{act} : LR(1)(G) \times \Sigma_\varepsilon \to \{\text{red } i \mid i \in [p]\} \cup \{\text{shift}, \text{accept}, \text{error}\}$$

is defined by

$$\mathrm{act}(I, x) := \begin{cases} \text{red } i & \text{if } i \neq 0, \pi_i = A \to \alpha \text{ and } [A \to \alpha\cdot, x] \in I \\ \text{shift} & \text{if } [A \to \alpha_1 \cdot x\alpha_2, y] \in I \text{ and } x \in \Sigma \\ \text{accept} & \text{if } [S' \to S\cdot, \varepsilon] \in I \text{ and } x = \varepsilon \\ \text{error} & \text{otherwise} \end{cases}$$

Corollary

For every $G \in CFG_\Sigma$, $G \in LR(1)$ iff its $LR(1)$ action function is well defined.

Outline of Lecture 11

Recap: $LR(1)$ Parsing

Generating Top-Down Parsers Using ANTLR

Generating Bottom-Up Parsers Using `yacc` and `bison`

LL and LR Parsing in Practice

Overview

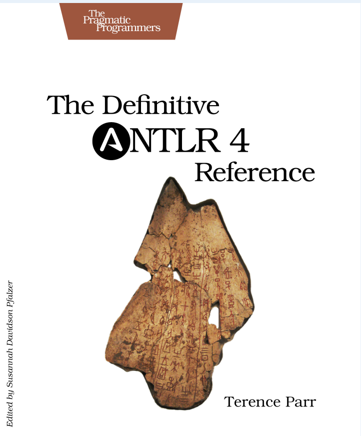
Semantic Analysis

Attribute Grammars

Overview of ANTLR

ANother Tool for Language Recognition

- Input: language description using EBNF grammars
- Output: recogniser for the language
- Supports recognisers for three kinds of input:
  - character streams (generation of scanner)
  - token streams (generation of parser)
  - node streams (generation of tree walker)
- Current version: ANTLR 4.7
  - generates $LL(*)$ recognisers: flexible choice of lookahead length
  - applies “longest match” principle
  - supports ambiguous grammars by using “first match” principle for rules
  - supports direct left recursion
  - targets Java, C++, C#, Python, JavaScript, Go, Swift
- Details:
  - http://www.antlr.org/
  - T. Parr: *The Definitive ANTLR 4 Reference*, Pragmatic Bookshelf, 2013



Example: Infix → Postfix Translator (Simple.g4)

```
grammar Simple;

// productions for syntax analysis
program returns [String s]: e=expr EOF {$s = $e.s;};
expr returns [String s]: t=term r=rest {$s = $t.s + $r.s;};
rest returns [String s]:
  PLUS t=term r=rest                    {$s = $t.s + "+" + $r.s;}
| MINUS t=term r=rest                   {$s = $t.s + "-" + $r.s;}
| /* empty */                           {$s = "";};
term returns [String s]: DIGIT          {$s = $DIGIT.text;};

// productions for lexical analysis
PLUS : '+';
MINUS : '-';
DIGIT : [0-9];
```

Java Code for Using Translator

```
import java.io.*;
import org.antlr.v4.runtime.*;
public class SimpleMain {
  public static void main(final String[] args)
  throws IOException {
    String printSource = null, printSymTab = null,
      printIR = null, printAsm = null;
    SimpleLexer lexer =  /* Create instance of lexer */
      new SimpleLexer(new ANTLRInputStream(args[0]));
    SimpleParser parser =  /* Create instance of parser */
      new SimpleParser(new CommonTokenStream(lexer));
    String postfix = parser.program().s;  /* Run translator */
    System.out.println(postfix);
  }
}
```

An Example Run

1. After installation, invoke ANTLR:
   ```
   $ java -jar /usr/local/lib/antlr-4.7-complete.jar Simple.g4
   ```
   (will generate `SimpleLexer.java`, `SimpleParser.java`, `Simple.tokens`, and `SimpleLexer.tokens`)
2. Use Java compiler:
   ```
   $ javac -cp /usr/local/lib/antlr-4.7-complete.jar Simple*.java
   ```
3. Run translator:
   ```
   $ java -cp .:/usr/local/lib/antlr-4.7-complete.jar SimpleMain '9-5+2'
   95-2+
   ```

Advantages of ANTLR

Advantages of ANTLR

- Generated (Java) code is similar to hand-written code
  - possible (and easy) to read and debug generated code
- Syntax for specifying scanners, parsers and tree walkers is identical
- Support for many target programming languages
- ANTLR is well supported and has an active user community

Outline of Lecture 11

Generating Bottom-Up Parsers Using `yacc` and `bison`

The `yacc` and `bison` Tools

Usage of `yacc` ("yet another compiler compiler"):

`spec.y`	$\xrightarrow{\texttt{yacc}}$	`y.tab.c`	`lex.yy.c`	$\xleftarrow{\texttt{[f]lex}}$	`spec.l`
`yacc` specification		Parser source	Scanner source		`[f]lex` specification

↓ `cc` ↓

`a.out`

Executable *LALR*(1) parser

Generating Bottom-Up Parsers Using yacc and bison

The yacc and bison Tools

Usage of yacc ("yet another compiler compiler"):

spec.y	$\xrightarrow{\texttt{yacc}}$	y.tab.c	lex.yy.c	$\xleftarrow{\texttt{[f]lex}}$	spec.l
yacc specification		Parser source	Scanner source		[f]lex specification

↓ cc ↓

a.out

Executable $LALR(1)$ parser

Like for [f]lex, a yacc specification is of the form

Declarations *(optional)*

%%

Rules

%%

Auxiliary procedures *(optional)*

Generating Bottom-Up Parsers Using yacc and bison

The yacc and bison Tools

Usage of yacc ("yet another compiler compiler"):

spec.y	$\xrightarrow{\texttt{yacc}}$	y.tab.c	lex.yy.c	$\xleftarrow{\texttt{[f]lex}}$ spec.l
yacc specification		Parser source	Scanner source	[f]lex specification

↓ cc ↓

a.out

Executable *LALR*(1) parser

Like for [f]lex, a yacc specification is of the form

Declarations (optional)

%%

Rules

%%

Auxiliary procedures (optional)

bison : upward-compatible GNU implementation of yacc
(more flexible w.r.t. file names, ...)

`yacc` Specifications

Declarations:
- Token definitions: `%token` *Tokens*
- Not every token needs to be declared (`'+', '=', ...`)
- Start symbol: `%start` *Symbol* (optional)
- C code for declarations etc.: `%{` *Code* `%}`

`yacc` Specifications

Declarations:
- Token definitions: `%token` *Tokens*
- Not every token needs to be declared (`'+', '=', ...`)
- Start symbol: `%start` *Symbol* (optional)
- C code for declarations etc.: `%{` *Code* `%}`

Rules: context-free productions and semantic actions
- $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ represented as

$$\begin{array}{lll} A & : & \alpha_1 \ \{\mathit{Action}_1\} \\ & | & \alpha_2 \ \{\mathit{Action}_2\} \\ & \vdots & \\ & | & \alpha_n \ \{\mathit{Action}_n\}; \end{array}$$

- Semantic actions = C statements for computing attribute values
- `$$` = attribute value of *A*
- `$i` = attribute value of *i*-th symbol on right-hand side
- Default action: `$$ = $1`

`yacc` Specifications

Declarations:
- Token definitions: `%token` *Tokens*
- Not every token needs to be declared (`'+', '=', ...`)
- Start symbol: `%start` *Symbol* (optional)
- C code for declarations etc.: `%{` *Code* `%}`

Rules: context-free productions and semantic actions
- $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ represented as

$$\begin{array}{lll} A & : & \alpha_1 \ \{\mathit{Action}_1\} \\ & | & \alpha_2 \ \{\mathit{Action}_2\} \\ & \vdots & \\ & | & \alpha_n \ \{\mathit{Action}_n\}; \end{array}$$

- Semantic actions = C statements for computing attribute values
- `$$` = attribute value of A
- `$i` = attribute value of i-th symbol on right-hand side
- Default action: `$$ = $1`

Auxiliary procedures: scanner (if not generated by `[f]lex`), error routines, ...

Example: Simple Desk Calculator I

```
%{/* SLR(1) grammar for arithmetic expressions (Example 9.9) */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%%
line   : expr '\n'          { printf("%d\n", $1); };
expr   : expr '+' term      { $$ = $1 + $3; }
       | term               { $$ = $1; };
term   : term '*' factor    { $$ = $1 * $3; }
       | factor             { $$ = $1; };
factor : '(' expr ')'       { $$ = $2; }
       | DIGIT              { $$ = $1; };
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) yylval = c - '0'; return DIGIT;
  return c;
}
```

Generating Bottom-Up Parsers Using `yacc` and `bison`

Example: Simple Desk Calculator II

```
$ yacc calc.y
$ cc y.tab.c -ly
$ a.out
2+3
5
$ a.out
2+3*5
17
```

An Ambiguous Grammar I

```
%{/* Ambiguous grammar for arithmetic expressions (Example 10.17) */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%%
line    : expr '\n'          { printf("%d\n", $1); };
expr    : expr '+' expr      { $$ = $1 + $3; }
        | expr '*' expr      { $$ = $1 * $3; }
        | DIGIT              { $$ = $1; };
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {yylval = c - '0'; return DIGIT;}
  return c;
}
```

An Ambiguous Grammar II

Invoking `yacc` with the option `-v` produces a report `y.output`:

```
State 8
    2 expr: expr . '+' expr
    2     | expr '+' expr .
    3     | expr . '*' expr

    '+'  shift and goto state 6
    '*'  shift and goto state 7
    '+'       [reduce with rule 2 (expr)]
    '*'       [reduce with rule 2 (expr)]

State 9
    2 expr: expr . '+' expr
    3     | expr . '*' expr
    3     | expr '*' expr .

    '+'  shift and goto state 6
    '*'  shift and goto state 7
    '+'       [reduce with rule 3 (expr)]
    '*'       [reduce with rule 3 (expr)]
```

Conflict Handling in `yacc`

Default conflict resolving strategy in `yacc`:

reduce/reduce: choose first conflicting production in specification

Conflict Handling in yacc

Default conflict resolving strategy in yacc:

reduce/reduce: choose first conflicting production in specification

shift/reduce: prefer shift

- resolves dangling-else ambiguity (Example 10.18) correctly
- also adequate for strong following weak operator ($*$ after $+$; Example 10.17) and for right-associative operators
- not appropriate for weak following strong operator and for left-associative operators ($\Longrightarrow$ reduce; see Example 10.17)

Generating Bottom-Up Parsers Using `yacc` and `bison`

Conflict Handling in `yacc`

Default conflict resolving strategy in `yacc`:

reduce/reduce: choose first conflicting production in specification

shift/reduce: prefer shift

- resolves dangling-else ambiguity (Example 10.18) correctly
- also adequate for strong following weak operator ($*$ after $+$; Example 10.17) and for right-associative operators
- not appropriate for weak following strong operator and for left-associative operators ($\Longrightarrow$ reduce; see Example 10.17)

For ambiguous grammar:

```
$ yacc ambig.y
conflicts: 4 shift/reduce
$ cc y.tab.c -ly
$ a.out
2+3*5
17
$ a.out
2*3+5
16
```

Precedences and Associativities in `yacc` I

General mechanism for resolving conflicts:

$$
\begin{array}{c}
\texttt{\%[left|right]}\ \mathit{Operators}_1 \\
\vdots \\
\texttt{\%[left|right]}\ \mathit{Operators}_n
\end{array}
$$

- operators in one line have given associativity and same precedence
- precedence increases over lines

Precedences and Associativities in yacc I

General mechanism for resolving conflicts:

%[left|right] *Operators₁*
⋮
%[left|right] *Operatorsₙ*

- operators in one line have given associativity and same precedence
- precedence increases over lines

Example 11.1

```
%left '+' '-'
%left '*' '/'
%right '^'
```

^ (right associative) binds stronger than * and / (left associative), which in turn bind stronger than + and − (left associative)

Precedences and Associativities in yacc II

```
%{/* Ambiguous grammar for arithmetic expressions
     with precedences and associativities */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%left '+'
%left '*'
%%
line    : expr '\n'  { printf("%d\n", $1); };
expr    : expr '+' expr    { $$ = $1 + $3; }
        | expr '*' expr    { $$ = $1 * $3; }
        | DIGIT            { $$ = $1; };
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {yylval = c - '0'; return DIGIT;}
  return c;
}
```

Precedences and Associativities in yacc III

```
$ yacc ambig-prio.y
$ cc y.tab.c -ly
$ a.out
2*3+5
11
$ a.out
2+3*5
17
```

Outline of Lecture 11

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity: LL wins

- LL parsing technique easier to understand
- recursive-descent parser easier to debug than LALR action tables

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity: LL wins

Generality: LALR wins

- "almost" $LL(1) \subseteq LALR(1)$ (only pathological counterexamples)
- LL requires elimination of left recursion and left factorization

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity: LL wins

Generality: LALR wins

Semantic actions: (see semantic analysis) LL wins

- actions can be placed anywhere in LL parsers without causing conflicts
- in LALR: implicit ε-productions
  $\Longrightarrow$ may generate conflicts

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity: LL wins

Generality: LALR wins

Semantic actions: (see semantic analysis) LL wins

Error handling: LL wins

- top-down approach provides context information
  $\Longrightarrow$ better basis for reporting and/or repairing errors

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity: LL wins

Generality: LALR wins

Semantic actions: (see semantic analysis) LL wins

Error handling: LL wins

Parser size: comparable

- LL: action table
- LALR: action/goto table

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity: LL wins

Generality: LALR wins

Semantic actions: (see semantic analysis) LL wins

Error handling: LL wins

Parser size: comparable

Parsing speed: comparable

- both linear in length of input program ($LL(1)$: see Lemma 7.15 for ε-free case)
- concrete figures tool dependent

LL and LR Parsing in Practice

In practice: use of $LL(1)/LL(*)$ or $LALR(1)$

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity: LL wins

Generality: LALR wins

Semantic actions: (see semantic analysis) LL wins

Error handling: LL wins

Parser size: comparable

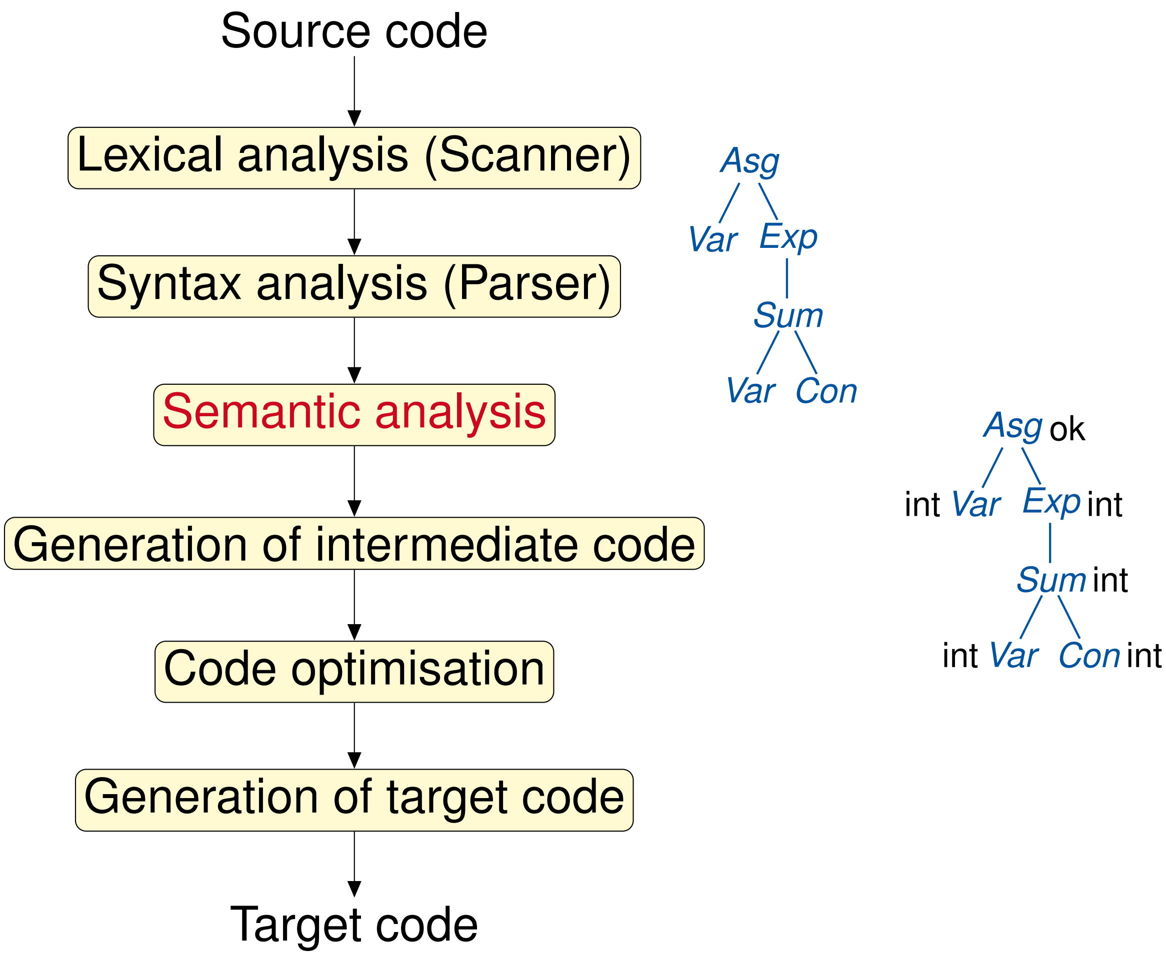
Parsing speed: comparable

Conclusion: choose LL when possible
(depending on available grammars and tools)

Outline of Lecture 11

Overview

Conceptual Structure of a Compiler



attribute grammars

Outline of Lecture 11

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?
- Is `x` defined before it is used?

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?
- Is `x` defined before it is used?
- Is the expression `3 * x + y` type consistent?

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?
- Is `x` defined before it is used?
- Is the expression `3 * x + y` type consistent?
- Where should the value of `x` be stored (register/stack/heap)?

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?
- Is `x` defined before it is used?
- Is the expression `3 * x + y` type consistent?
- Where should the value of `x` be stored (register/stack/heap)?
- Do `p` and `q` refer to the same memory location (aliasing)?
- ...

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?
- Is `x` defined before it is used?
- Is the expression `3 * x + y` type consistent?
- Where should the value of `x` be stored (register/stack/heap)?
- Do `p` and `q` refer to the same memory location (aliasing)?
- ...

These cannot be expressed using context-free grammars!

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?
- Is `x` defined before it is used?
- Is the expression `3 * x + y` type consistent?
- Where should the value of `x` be stored (register/stack/heap)?
- Do `p` and `q` refer to the same memory location (aliasing)?
- ...

These cannot be expressed using context-free grammars!

(For example, $\{ww \mid w \in \Sigma^+\} \notin CFL_\Sigma$)

Static Semantics

Static semantics

Refers to properties of program constructs

- which are true for every occurrence of this construct in every program execution and
- can be decided at compile time (“static”)
- but are context-sensitive and thus not expressible using context-free grammars (“semantics”).

Static Semantics

Static semantics

Refers to properties of program constructs

- which are true for every occurrence of this construct in every program execution and
- can be decided at compile time (“static”)
- but are context-sensitive and thus not expressible using context-free grammars (“semantics”).

Example properties

Static: type or declaredness of an identifier, number of registers required to evaluate an expression, ...

Dynamic: value of an expression, size of procedure stack, ...

Outline of Lecture 11

Recap: $LR(1)$ Parsing

Generating Top-Down Parsers Using ANTLR

Generating Bottom-Up Parsers Using yacc and bison

LL and LR Parsing in Practice

Overview

Semantic Analysis

Attribute Grammars

Attribute Grammars I

Goal: compute context-dependent but runtime-independent properties of a given program

Idea: enrich context-free grammar by semantic rules which annotate syntax tree with attribute values

$\Longrightarrow$ Semantic analysis = attribute evaluation

Result: attributed syntax tree

Attribute Grammars I

Goal: compute context-dependent but runtime-independent properties of a given program

Idea: enrich context-free grammar by semantic rules which annotate syntax tree with attribute values

$\Longrightarrow$ Semantic analysis = attribute evaluation

Result: attributed syntax tree

In greater detail:

- With every grammar symbol a set of attributes is associated.
- Two types of attributes are distinguished:

  Synthesized: bottom-up computation (from the leaves to the root)

  Inherited: top-down computation (from the root to the leaves)
- With every production a set of semantic rules is associated.

Attribute Grammars II

Advantage: attribute grammars provide a very flexible and broadly applicable mechanism for transporting information through the syntax tree ("syntax-directed translation")

- Attribute values: symbol tables, data types, code, error flags, ...
- Application in Compiler Construction:
  - static semantics
  - program analysis for optimisation
  - code generation
  - error handling
- Automatic attribute evaluation by compiler generators (cf. `ANTLR`'s and `yacc`'s synthesized attributes)
- Originally designed by D. Knuth for defining the semantics of context-free languages (Math. Syst. Theory 2 (1968), pp. 127–145)

Example: Type Checking I

Example 11.1 (Attribute grammar for type checking)

$$
\begin{aligned}
\mathit{Pgm} &\rightarrow \mathit{Dcl}\ \mathit{Cmd} \\
\mathit{Dcl} &\rightarrow \varepsilon \\
&\mid \mathit{Typ}\ \texttt{var}\,;\mathit{Dcl} \\
\mathit{Typ} &\rightarrow \texttt{int} \\
\mathit{Typ} &\rightarrow \texttt{bool} \\
\mathit{Cmd} &\rightarrow \varepsilon \\
&\mid \texttt{var} := \mathit{Exp}\,;\mathit{Cmd} \\
\mathit{Exp} &\rightarrow \texttt{num} \\
&\mid \texttt{var} \\
&\mid \mathit{Exp} + \mathit{Exp} \\
&\mid \mathit{Exp} < \mathit{Exp} \\
&\mid \mathit{Exp}\ \&\&\ \mathit{Exp}
\end{aligned}
$$

Example: Type Checking I

Example 11.1 (Attribute grammar for type checking)

$Pgm \to Dcl\ Cmd$	$ok.0 = ok.2$
$Dcl \to \varepsilon$	$st.0 = [id \mapsto \mathrm{err} \mid id \in Id]$
$\mid\ Typ\ \mathtt{var}\,;Dcl$	$st.0 = st.4[id.2 \mapsto typ.1]$
$Typ \to \mathtt{int}$	$typ.0 = \mathrm{int}$
$Typ \to \mathtt{bool}$	$typ.0 = \mathrm{bool}$
$Cmd \to \varepsilon$	$ok.0 = \mathrm{true}$
$\mid\ \mathtt{var} := Exp\,;Cmd$	$ok.0 = (env.0(id.1) = typ.3 \wedge ok.5)$
$Exp \to \mathtt{num}$	$typ.0 = \mathrm{int}$
$\mid\ \mathtt{var}$	$typ.0 = env.0(id.1)$
$\mid\ Exp + Exp$	$typ.0 = (typ.1 = typ.3 = \mathrm{int}\ ?\ \mathrm{int} : \mathrm{err})$
$\mid\ Exp < Exp$	$typ.0 = (typ.1 = typ.3 = \mathrm{int}\ ?\ \mathrm{bool} : \mathrm{err})$
$\mid\ Exp\ \&\&\ Exp$	$typ.0 = (typ.1 = typ.3 = \mathrm{bool}\ ?\ \mathrm{bool} : \mathrm{err})$

- Synthesized attributes: *id* (identifier name), *ok* (Boolean result), *st* (symbol table, mapping identifiers to types), *typ* (data type in $\{\mathrm{bool}, \mathrm{int}, \mathrm{err}\}$)

Example: Type Checking I

Example 11.1 (Attribute grammar for type checking)

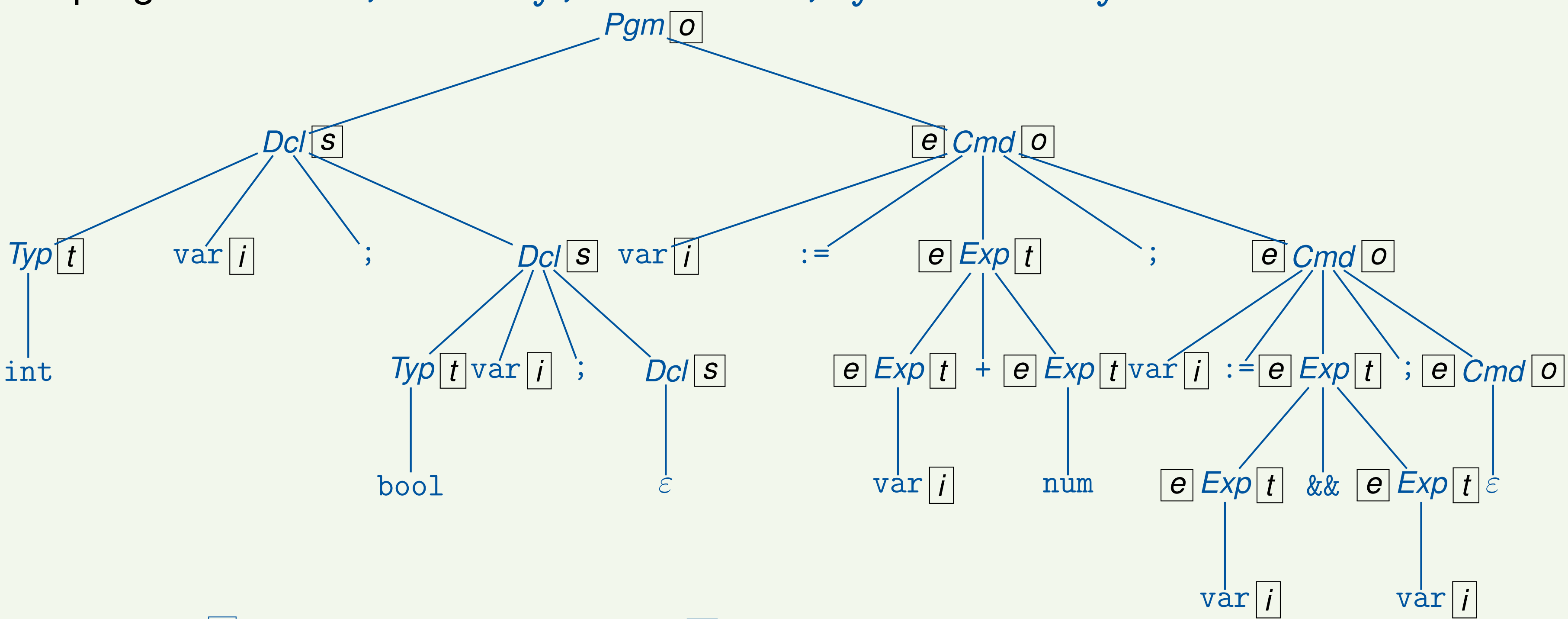
$Pgm \to Dcl\ Cmd$	$ok.0 = ok.2$	$env.2 = st.1$
$Dcl \to \varepsilon$	$st.0 = [id \mapsto \mathrm{err} \mid id \in Id]$	
$\mid\ Typ\ \mathtt{var}\,;Dcl$	$st.0 = st.4[id.2 \mapsto typ.1]$	
$Typ \to \mathtt{int}$	$typ.0 = \mathrm{int}$	
$Typ \to \mathtt{bool}$	$typ.0 = \mathrm{bool}$	
$Cmd \to \varepsilon$	$ok.0 = \mathrm{true}$	
$\mid\ \mathtt{var} := Exp\,;Cmd$	$ok.0 = (env.0(id.1) = typ.3 \wedge ok.5)$	$env.3 = env.0 \quad env.5 = env.0$
$Exp \to \mathtt{num}$	$typ.0 = \mathrm{int}$	
$\mid\ \mathtt{var}$	$typ.0 = env.0(id.1)$	
$\mid\ Exp + Exp$	$typ.0 = (typ.1 = typ.3 = \mathrm{int}\ ?\ \mathrm{int} : \mathrm{err})$	$env.1 = env.0 \quad env.3 = env.0$
$\mid\ Exp < Exp$	$typ.0 = (typ.1 = typ.3 = \mathrm{int}\ ?\ \mathrm{bool} : \mathrm{err})$	$env.1 = env.0 \quad env.3 = env.0$
$\mid\ Exp\ \&\&\ Exp$	$typ.0 = (typ.1 = typ.3 = \mathrm{bool}\ ?\ \mathrm{bool} : \mathrm{err})$	$env.1 = env.0 \quad env.3 = env.0$

- Synthesized attributes: *id* (identifier name), *ok* (Boolean result), *st* (symbol table, mapping identifiers to types), *typ* (data type in $\{\mathrm{bool}, \mathrm{int}, \mathrm{err}\}$)
- Inherited attributes: *env* (environment – same type as symbol table)

Example: Type Checking II

Example 11.2 (Attributed syntax tree)

For program `int x; bool y; x := x+1; y := x && y`:



($\boxed{e} = env$, $\boxed{i} = id$, $\boxed{o} = ok$, $\boxed{s} = st$, $\boxed{t} = typ$)