richardwu.ca

CS 444/644 Course Notes

Compiler Construction

Ondřej Lhoták • Winter 2019 • University of Waterloo

Last Revision: March 5, 2019

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 January 7, 2019

1.1 Basic overview of a compiler

A **compiler** takes a source language and translates it to a target language. The source language could be, for example, C, Java, or JVM bytecode and the target language could be, for example, machine language or JVM bytecode.
A compiler could be divided into two parts:

Front-end analysis Front-end could be further divided into two parts: the first being scanning and parsing (assignment 1) and the second being context-sensitive analysis (assignment 2,3,4).

Some refer to context-sensitive analysis as "middle-end".

Back-end synthesis The backend could also be divided into two parts: the first being optimization (CS 744) and the second being code generation (assignment 5).

1.2 Overview of front-end analysis

Goal: is the input a valid program? An auxiliary step is to also generate information about the program for use in synthesis later on.
There are several steps in the front-end:

Scanning Split sequence of characters into sequence of tokens. Each token consists of its lexeme (actual characters) and its kind.

There are tools for generating the DFA expressions from a regular language e.g. lex.

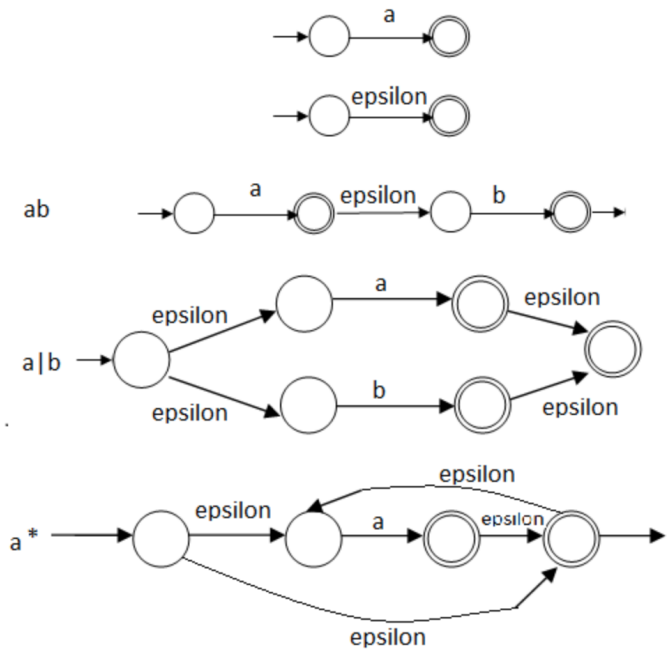
2 January 9, 2019

2.1 Scanning tools (lex)

Our goal is to specify our grammar in terms of regular expressions (regex) which lex can convert into a scanning DFA.
Review of regex to language (set of words):

RE	$L(e)$
$\varnothing$	$\{\}$
ϵ	$\{\epsilon\}$
$a \in \Sigma$	$\{a\}$
$e_1 e_2$	$\{xy \mid x \in L(e_1), y \in L(e_2)\}$
$e_1 \mid e_2$	$L(e_1) \cup L(e_2)$
e^*	$L(\epsilon \mid e \mid ee \mid eee \mid \ldots)$

The corresponding NFAs we can construct per each regex rule are



Let Σ be the set of characters, Q the set of states, q_0 the initial state, A the accepting states, and δ the transition function, an NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$ where

$$\text{NFA} : \delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q \text{ (subset of Q)}$$
$$\text{DFA} : \delta : Q \times \Sigma \to Q$$

i.e. the transition function of an NFA returns a subset of states in Q.
We note in our above NFAs some accepting states are equivalent (connected by an ϵ transition).
We define

Definition 2.1 (ϵ-closure I)**.** The ϵ-closure(S) of a set of states S is the set of states reachable from S by (0 or more) ϵ-transitions.

Another equivalent recursive definition

Definition 2.2 (ϵ-closure II)**.** Smallest set S' such that

$$S' \supseteq S$$
$$S' \supseteq \{q \mid q' \in S', q \in \delta(q', \epsilon)\}$$

We note that any NFA can be converted in a corresponding DFA. The input is an NFA $(\Sigma, Q, q_0, A, \delta)$ and we'd like to get a DFA $(\Sigma, Q', q_0', A', \delta')$ where

$$q_0' = \epsilon\text{-closure}(\{q_0\})$$
$$\delta'(q', a) = \epsilon\text{-closure}\Big(\bigcup_{q \in q'} \delta(q, a)\Big)$$

we note that each state of our DFA is a set of states in the original NFA e.g. $\{1, 2, 4\}$ may be a state in the DFA.

We generate more states in our DFA by iterating through every $a \in \Sigma$ and applying rule two to our initial state q'_0. We do this until no further states can be generated from existing DFA states and all $a \in \Sigma$ have been exhausted. We note that the entire set of states Q' in the DFA can be recursively defined as the smallest set of subsets of Q such that

$$Q' \supseteq \{q'_0\}$$
$$Q' \supseteq \{\delta'(q', a) \mid q' \in Q'\} \qquad \forall a \in \Sigma$$

We note that if any accepting state is included in a state of the DFA, we can accept it (since we can reach the corresponding accepting state in the NFA). Thus

$$A' = \{q' \in Q' \mid q' \cap A \neq \varnothing\}$$

3 January 14, 2019

3.1 DFA recognition scanning

The algorithm for using a DFA to recognize if a word is valid in the grammar

Algorithm 1 DFA recognition

input word w, DFA $M = (\Sigma, Q, q_0, \delta A)$
output boolean $w \in L(M)$?

```
q ← q_0
for i from 1 to |w| do
    q ← δ(q, w[i])
return q ∈ A
```

Scanning is similar where we take a *sequence of symbols* and convert it into a *sequence of tokens* using a DFA. In **maximal munch** we have

Algorithm 2 Maximal munch (abstract)

input DFA M specifying language L of valid tokens, string of symbols w
output sequence of tokens, each token $\in L$ that concantenates to w

```
while until end of output do
    Find a maximal prefix of remaining input that is in L
    ERROR if no non-empty prefix in L
```

note that **maximal munch** takes the **maximal prefix**: if we do not take the maximal prefix we may run into ambiguity. A more concrete implementation

Algorithm 3 Maximal munch (concrete)

```
while until end of output do
    Run DFA and record last seen accepting state until it gets stuck (or it transitions to ERROR state)
    Backtrack DFA and the input to last seen accepting state
    ERROR if there is no accepting state
    Output prefix as the next token
    Set DFA back to start state
```

Note that in Java which uses maximal munch, $a - -b$ would be parsed as $a(- - b)$ which would return a parsing error (as opposed to $a - (-b)$ since $--$ is a token in Java).

3.2 Constructing the scanning DFA

The overall algorithm to convert regular expressions denoting tokens to a scanning DFA is

Algorithm 4 Regex to scanning DFA

input REs $R_1, \ldots, R_n$ for token kinds in priority order
output DFA with accepting states labelled with token kinds
1: Construct an NFA M for $R_1 \mid R_2 \mid \ldots \mid R_n$
2: Convert NFA to DFA M' (each state of M' is set of states in M)
3: For each accepting state of M', output highest priority token kind of the set of NFA accepting states

4 January 16, 2019

4.1 Context-free grammar

Regular expressions are great for scanning but would not work for an arbitrary depth of tokens when it comes to parsing grammar.
To address this we define **context-free grammars** which uses recursion to specify arbitrary structures in the grammar at an arbitrary depth in the parse tree.

Definition 4.1 (Context free grammar)**.** A **context-free grammar** is a 4-tuple $G = (N, T, R, S)$ where we have (NB: notation for each class)

Terminals T (e.g. a, b, c; denoted with lowercase alphabet)

Non-terminals N (e.g. A, B, C, S; denoted with uppercase alphabet)

Symbols The set of symbols are $V = N \cup T$ (e.g. W, X, Y, Z)

String of terminals T^* (e.g. w, x, y, z)

String of symbols V^* (e.g. α, β, γ)

Production rules $R \subseteq N \times V^*$ (e.g. $A \to \alpha$)

Start non-terminal S

Definition 4.2 (Directly derives)**.** $\beta A \gamma \Rightarrow \beta \alpha \gamma$ if $A \to \alpha \in R$: that is $\beta A \gamma$ **directly derives** $\beta \alpha \gamma$.

Definition 4.3 (Derives)**.** $\alpha \Rightarrow^* \beta$ if $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$: that is α **derives** β.

Definition 4.4 (Sentential form)**.** α is a **sentential form** if $S \Rightarrow^* \alpha$.

Definition 4.5 (Sentence)**.** x is a **sentence** if $x \in T^*$ and x is a sentential form.

Definition 4.6 (Language)**.** $L(G) = \{x \in T^* \mid S \Rightarrow^* x\}$ is the **language generated by** G (set of sentences).

4.2 Recognizer and parsing

The task of a **recognizer** is to see if $x \in L(G)$ for some grammar G.
The task of **parsing** is to find a derivation from S to x.

Example 4.1. Suppose we have a grammar G with $R = \{A \to BgC, B \to ab, C = ef\}, S = A, N = \{A, B, C\}, T = \{a, b, e, f, g\}$.
We can derive the following sentence

$$A \Rightarrow BgC \Rightarrow abgC \Rightarrow abgef$$

notice this is the only sentence in $L(G)$.
We could represent this as a tree where A is at the root, B, g, C are each children of A (3 children), a, b and e, f are children of B and C, respectively.
Note that for this particular grammar, we may have multiple derivations for $abgef$ (we could have expanded C first) but we have one unique parse tree.

Definition 4.7 (Ambiguous grammar)**.** A grammar is **ambiguous** if $\exists > 1$ parse tree for the same sentence.

Definition 4.8 (Left(Right) derivation)**.** In a **left(right) derivation** we always expand the left(right)-most non-terminal.

There is a *one-to-one correspondence* between parse trees, left derivations, and right derivations: that is given a sentence with a unique parse tree, it has a unique left derivation and a unique right derivation.

4.3 Top-down parser

The simplest approach for parsing a sentence x from S (start symbol) is using a **top-down parser**:

Algorithm 5 Top-down parser

1: $\alpha \leftarrow S$
2: **while** $\alpha \neq x$ **do**
3: Replace **first** non-terminal A in α with β, assuming $A \to \beta \in R$

Replacing the *first* non-terminal results in a left derivation.

5 January 21, 2019

5.1 LL(1) parser

Example 5.1. Given a grammar with the following production rules

$$E \to aE'$$
$$E' \to +a$$
$$E' \to \epsilon$$

Suppose we wanted to derive $a + a$. Intuitively we have $E \Rightarrow aE' \Rightarrow a + a$. Our intuition told us to use $E' \to +a$ instead of $E' \to' \epsilon$.

To improve on our top-down parsing, we introduce the LL(1) parser:

Algorithm 6 LL(1) parser

input x is the input string to parse
1: $\alpha \leftarrow S$
2: **while** $\alpha \neq x$ **do**
3: Let A be the first non-terminal in α ($\alpha = yA\gamma$)
4: Let a be the first terminal after y in x ($x = ya\zeta$)
5: $A \to \beta \leftarrow \text{predict}(A, a)$
6: Replace A with β in α

where predict(A, a) follows our intuition of picking the rule with A on the LHS that works best when a is the next terminal: our **lookahead**.

Remark 5.1. The first "L" represents scanning the **input from left to right**; the second "L" denotes **left-canonical derivation** (always expanding the leftmost non-terminal); the "1" denotes a 1 **symbol lookahead**

Definition 5.1 (Augmented grammar)**.** In order to make our 1 lookahead algorithm work with the start rule, we need to augment it:

1. Add a fresh terminal "$"
2. Add production $S' \to S\$$ where S' is our new start and S was the start of our original grammar.

We also need to append "$" to our input. Our augmented grammar becomes

$$S' \to E\$$$
$$E \to aE'$$
$$E' \to +a$$
$$E' \to \epsilon$$

If we implement $LL(1)$ naively with string replacements we could see that the algorithm runs in $O(n^2)$.
To do this in $O(n)$ time we could utilize a stack for our α derivation. Our revised algorithm proceeds as

Algorithm 7 LL(1) parser

input x is the input string to parse
1: Push $S\$$ onto stack
2: **for** a in $x\$$ **do**
3: **while** top of stack is a non-terminal A **do**
4: Pop A
5: Find $A \to \beta$ in predict(A, a) or ERROR
6: Push β
7: Pop b, terminal on top of stack
8: **if** $b \neq a$ **then**
9: ERROR

We define

$$\text{predict}(A, a) = \{A \to \beta \in R \mid \exists\gamma \quad \beta \Rightarrow^* a\gamma \text{ or } (\beta \Rightarrow^* \epsilon \text{ and } \exists\gamma, \delta \quad S' \Rightarrow^* \gamma A a \delta\}$$

A grammar is $LL(1)$ iff $|\text{predict}(A, a)| \leq 1$ for all A, a.

5.2 First, nullable and follow sets

Definition 5.2 (First set)**.** If $\beta \Rightarrow^* a\gamma$ then $a \in \text{first}(\beta)$ where first(β) is the **first set** of β.

Definition 5.3 (Nullable set)**.** If $\beta \Rightarrow^* \epsilon$ then β is **nullable**.

Definition 5.4 (Follow set)**.** If $S' \Rightarrow^* \gamma A a \delta$ then $a \in \text{follow}(A)$ where follow(A) is the **follow set** of A.

Note that in our above example grammar, we have

$$\text{nullable} = \{\epsilon\}$$

and

$$\begin{aligned}\text{first}(\epsilon) &= \{\}\\ \text{first}(+a) &= \{+\}\\ \text{first}(aE') &= \{a\}\\ \text{first}(E\$') &= \{a\}\end{aligned}$$

In general to compute the **follow set**, we define it as

1. If $B \to \alpha A \gamma$ then $\text{first}(\gamma) \subseteq \text{follow}(A)$
2. If $B \to \alpha A \gamma$ and $\text{nullable}(\gamma)$ then $\text{follow}(B) \subseteq \text{follow}(A)$

6 January 23, 2019

6.1 Note on LL(k) parsers

Note that in LL(k) parsers we perform leftmost derivation. This means that the sentence $3-2-1$ would be parsed (with parentheseses denoting derivations/subtrees) $3-(2-1)$ which would incorrectly result in 2.
Ideally we wanted $(3-2)-1$ or **left associative**. In general, LL(k) parser cannot parse arbitrary left associative languages.

Remark 6.1. One could imagine an LL(1) parser as generating the parse tree from top to bottom, left to right where the parse tree is rooted at S and the leaves are the tokens in the input x.

An alternative to LL(k) parsers which is a top-down parser (from start symbol S to input) are bottom-up parsers.

6.2 Bottom-up parsing

Example 6.1. Suppose we have the same grammar as before

$$\begin{aligned}S &\to E\$\\ E &\to E + a\\ E &\to a\end{aligned}$$

Suppose we wanted to parse the sentence $a + a\$$. We proceed in a bottom up fashion

$$\begin{aligned}a + a\$ &\Leftarrow E + a\$\\ &\Leftarrow E\$\\ &\Leftarrow S\end{aligned}$$

where we scan from left to right the tokens in the input and reduce with "intuition".

Remark 6.2. Given a parse tree rooted at S with leaves as tokens of input x, a bottom-up parser generates the parse tree from the bottom-left corner and moves upwards and rightwards until the start symbol is reached.

proceed to look at a particular bottom-up parser.

6.3 LR(0) parser

The best way to implement these parsers is to think about invariants and ensuring our algorithm always maintains the invariants:

LL(k) invariant The invariant for an LL(k) parser is that

$$S \Rightarrow^* \alpha$$
$$S \Rightarrow^* \text{ seen input } + \text{ stack}$$

i.e. S always derives the string of processed symbols we've derived so far.

LR(k) invariant Here the invariant is that

$$\alpha \Rightarrow^* x$$
$$\text{stack } + \text{ unseen input } \Rightarrow^* x$$

that is our current string of processed symbols plus the rest of the unseen input should be able to derive our entire input. Equivalently

$$S \Rightarrow^* \alpha$$
$$S \Rightarrow^* \text{ stack } + \text{ unseen input}$$

in other words we require the stack to always be a viable prefix:

Definition 6.1 (Viable prefix)**.** α is a **viable prefix** if it is a prefix of some sentential form i.e.

$$\exists \beta \text{ s.t. } S \Rightarrow^* \alpha\beta$$

In LR(0) parsing as above, we keep a stack of symbols we have processed so far. We have the following algorithm

Algorithm 8 LR(0) parser

```
1: for a in x$ do
2:     while Reduce(stack) = {A → γ} do
3:         Pop γ off stack (i.e. pop |γ| tokens)          ▷ Reduce
4:         Push A onto stack
5:     if Reject(stack + a) then ERROR
6:     Push a onto stack                                  ▷ Shift
```

where we define

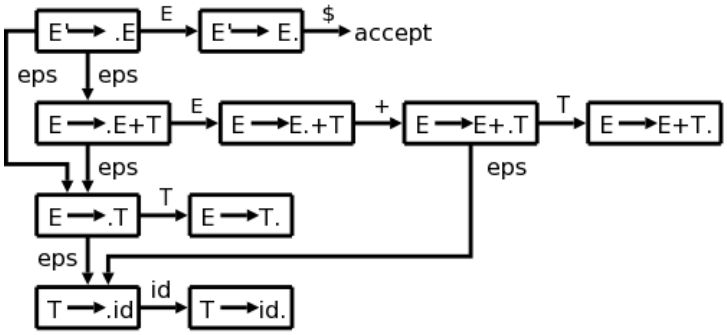
$$\text{Reduce}(\alpha) = \{A \to \gamma \mid \exists \beta \text{ s.t. } \alpha = \beta\gamma \text{ and } \beta A \text{ is a VP}\}$$

where VP stands for **viable prefix** (see above) and Reject(α) is true if α is not a VP.

Question 6.1. How do we check if α is a VP? Note that the set of *all* VPs is itself a context-free language.

Exercise 6.1. Given a CFG G, construct a CFG G' for the set of VPs of G.

Remark 6.3. Only some VPs α's occur during the algorithm, thus we can build a simpler NFA for VPs that actually occur.
We can construct the NFA (such as the following) systematically from a given grammar



We define the construction of the LR(0) NFA as follows:

$\Sigma = T \cup N$
$Q = \{A \to \alpha \cdot \beta \mid A \to \alpha\beta \in R\}$
$q_0 = S' \to S\$$
$A = Q$ all states are accepting i.e. stack is always a VP
$\delta(A \to \alpha \cdot B\beta, \epsilon) = \{B \to \cdot\gamma \mid B \to \gamma \in R\}$
$\delta(A \to \alpha \cdot X\beta, X) = \{A \to \alpha X \cdot \beta\}$

7 January 28, 2019

To parse, we can simply just follow the NFA and reduce if the NFA ends up in state $A \to \gamma\cdot$ on input $\beta\gamma$ (note: Reduce($\beta\gamma$) contains $A \to \gamma$ because the NFA would also accept βA; a VP is always kept on the stack).
However, to make our process deterministic, we need to convert the NFA to a DFA.

Definition 7.1 (Reduce-reduce conflict (LR(0)))**.** If the resulting DFA state contains $A \to \gamma\cdot$ and $B \to \delta\cdot$ for two different productions, a **reduce-reduce conflict** occurs.

Definition 7.2 (Shift-reduce conflict (LR(0)))**.** If the DFA state contains $A \to \gamma\cdot$ and $B \to \alpha \cdot X\beta$, then a **shift-reduce conflict** occurs.

Definition 7.3 (LR(0) grammar)**.** An LR(0) grammar is **unambiguous** if there are no conflicts using the above NFA and DFA construction.

Remark 7.1. Instead of walking the DFA and returning the start state every time a reduction happens, we can **optimize** this procedure to be *linear time* using an additional stack for DFA states. The invariant is that $\delta(q_0,$ processed stack$) =$ top of state stack i.e. the top of our state stack is always our current state in the DFA. We also need to synchronize the number of times we push and pop from both stacks (e.g. a reduction rule that pops 3 symbols would require popping the state stack 3 times).

Example 7.1. Here is an example of LR(0) parsing based on the DFA states:

20

Example LR Parsing

Grammar:
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ **id**

Stack	Input	Action
$ 0	**id*id+id$**	shift 5
$ 0 **id** 5	***id+id$**	reduce 6 goto 3
$ 0 *F* 3	***id+id$**	reduce 4 goto 2
$ 0 *T* 2	***id+id$**	shift 7
$ 0 *T* 2 * 7	**id+id$**	shift 5
$ 0 *T* 2 * 7 **id** 5	**+id$**	reduce 6 goto 10
$ 0 *T* 2 * 7 *F* 10	**+id$**	reduce 3 goto 2
$ 0 *T* 2	**+id$**	reduce 2 goto 1
$ 0 *E* 1	**+id$**	shift 6
$ 0 *E* 1 + 6	**id$**	shift 5
$ 0 *E* 1 + 6 **id** 5	**$**	reduce 6 goto 3
$ 0 *E* 1 + 6 *F* 3	**$**	reduce 4 goto 9
$ 0 *E* 1 + 6 *T* 9	**$**	reduce 1 goto 1
$ 0 *E* 1	**$**	accept

7.1 LR(1) parser

For the grammar

$$S \rightarrow E\$$$
$$E \rightarrow a + E$$
$$E \rightarrow a$$

we end up having $E \rightarrow a \cdot +E$ and $E \rightarrow a\cdot$ in the same DFA state (reduce-shift conflict), therefore the grammar is NOT LR(0).
Is there a way to disambiguate the grammar? Surely if we could *lookahead one token* to see if either the next token of the input is \$ (then we would reduce with $E \rightarrow a\cdot$) or if it is + (then we would shift +), then we could remedy our conflict.
This is the idea behind **LR(1) parsing**: LR parsing with 1 lookahead token. The algorithm is similar to LR(0) parsing:

Algorithm 9 LR(1) parser

```
1: for a in x$ do
2:     while Reduce(stack, a) = {A → γ} do
3:         Pop γ off stack (i.e. pop |γ| tokens)          ▷ Reduce
4:         Push A onto stack
5:     if Reject(stack + a) then ERROR
6:     Push a onto stack                                  ▷ Shift
```

where we define our new Reduce function with a lookahead token a:

$$\text{Reduce}(\alpha, a) = \{A \rightarrow \gamma \mid \exists \beta \text{ s.t. } \alpha = \beta\gamma \text{ and } \beta A a \text{ is a VP}\}$$

7.2 SLR(1) parser

How do we determine if βAa is a VP for a reduction rule $A \to \gamma\cdot$?

Option 1 (SLR(1)) If $a \notin \text{follow}(A)$ then βAa definitely is not a VP

Option 2 (LR(1)) Construct a better DFA (discussed later)

For **Option 1** (called **SLR(1)**) we can combine both the DFA for LR(0) and our follow set:

- Use LR(0) DFA to decide if βA is a VP
- Use follow(A) to decide if βAa is a VP

We redefine what it means for there to be conflicts in our SLR(1) DFA:

Definition 7.4 (Reduce-reduce conflict (SLR(1)))**.** If the resulting DFA state contains $A \to \gamma\cdot$ and $B \to \delta\cdot$ for two different productions, *and* $\text{follow}(A) \cap \text{follow}(B) \neq \varnothing$ then a **reduce-reduce conflict** occurs.

Definition 7.5 (Shift-reduce conflict (SLR(1)))**.** If the DFA state contains $A \to \gamma\cdot$ and $B \to \alpha \cdot X\beta$, *and* $X \in \text{follow}(A)$ then a **shift-reduce conflict** occurs.

Remark 7.2. SLR(1) uses the same parser as LR(1): the only difference is how the two checks if βAa is a VP.

7.3 Distinction between LR(0), SLR(1), LALR(1), LR(1)

The distinction between different types of parser depends on how it determines VPs (ranked from most general to most specific; prior types imply later types):

General Determines if $\beta A\gamma$ is a sentential form (where γ is the rest of input)

LR(1) Determines if βAa is a VP

LALR(1) Determines if βA is a VP and $a \in \ldots$??? (discussed later)

SLR(1) Determines if βA is a VP and $a \in \text{follow}(A)$

LR(0) Determines if βA is a VP

8 January 30, 2019

8.1 Generating the LR(1) parse table

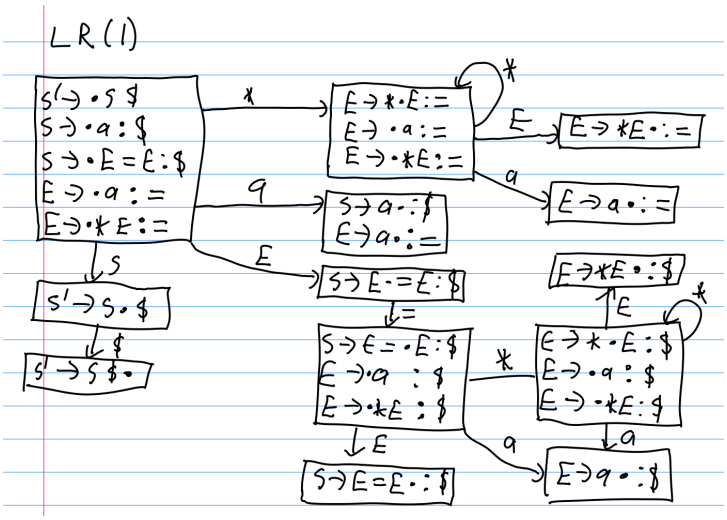
As noted previously, the difference between LR(1) and SLR(1) is how they determine whether a proposed prefix is a **viable prefix**.

Example 8.1. The following grammar is LR(1) but not SLR(1):

$$S' \to S\$$$
$$S \to a$$
$$S \to E = E$$
$$E \to a$$

Note that given input $a\$$, SLR(1) will try to determine whether to reduce a to S or E. It will check if E is a VP and if the lookahead token, \$, is in follow(E). Similarly for S.
Note that in SLR(1) both E and S are viable candidates but only S is the correct reduction since if we reduce to E we would require an $=$ afterwards.

The **idea** behind LR(1) (vs SLR(1)) is to keep *specific follow sets* in each NFA state. In the example above, we would keep track of the specific follow set of the first E_1 and for the second E_2 in their corresponding NFA states. When we are constructing the NFA, if we are in an NFA state where we have a bullet point $\cdot$ in front of a non-terminal (e.g. $S \to \cdot E = E\$$), we generate an ϵ transition out of that state (e.g. to $E \to \cdot a\$$).
For the new generated state we annotate it with the follow set of the specific instance of the non-terminal (in this case it is the first E so we annotate the new state with $=$).
Finally once we convert the NFA to a DFA we end up with something like this:



where each state is annotated with the lookahead symbol from the follow set.

Remark 8.1. The lookahead generated from the follow sets only matter for states with *reductions*; however, to carry through the follow set lookaheads we need to include them in intermediary state during generation.

That is: if NFA ends up in a state $A \to \gamma \cdot : a$ on a stack $\beta\gamma$, then reduce$(\beta\gamma, a)$ contains $A \to \gamma$ because the NFA would also accept $\beta A a$.
This answers the question: is $\beta A a$ a VP?

Definition 8.1 (Reduce-reduce conflict (LR(1)))**.** If DFA state contains $A \to \gamma \cdot : a$ and $B \to \delta \cdot : a$ then we have a **reduce-reduce conflict**.

Definition 8.2 (Shift-reduce conflict (LR(1)))**.** If DFA state contains $A \to \gamma \cdot : a$ and $B \to \alpha \cdot a\beta : b$ then we have a **shift-reduce conflict**.

To build the LR(1) NFA we define it declaratively as:

$$\Sigma = T \cup N$$
$$Q = \{A \to a \cdot \beta : a \mid A \to \alpha\beta \in R, a \in T\}$$
$$q_0 = \{S' \to \cdot S\$: \$\}$$
$$\delta(A \to \alpha \cdot X\beta : a, X) = \{A \to \alpha X \cdot \beta : a\}$$
$$\delta(A \to \alpha \cdot B\beta : a, \epsilon) = \{B \to \cdot\gamma : b \mid B \to \gamma \in R \text{ and } b \in \text{first}(\beta a)\}$$

9 February 4, 2019

9.1 LALR(1) grammar

Idea: we use the LR(0) DFA with follow sets local to each LR(1) DFA state.

Definition 9.1 (Core of a state)**.** The **core** of a LR(1) DFA state is defined as

$$core(q) = \{A \to \alpha \cdot \beta \mid (A \to \alpha \cdot \beta : a) \in q\}$$

i.e. the set of items belonging to the state q.

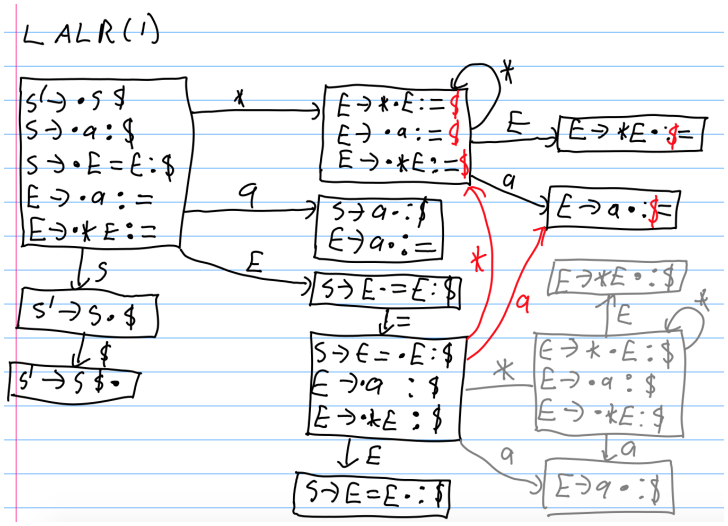
Fact 9.1. We can replace each LR(1) DFA state by its **core** i.e. its LR(0) DFA state.

The algorithm to convert an LR(1) DFA's states to LALR(1) DFA states:

Algorithm 10 LR(1) to LALR(1)

input M is the LR(0) DFA, M' is the LR(1) DFA
1: **for** each state q of M **do**
2: **for** each item $A \to \alpha \cdot \beta$ of q **do**
3: $l' \leftarrow \{a \mid q' \text{ is a state of } M', q = core(q'), (A \to \alpha \cdot \beta : a) \in q'\}$
4: q with lookaheads l' is the new LALR(1) state/core

That is: we simply coalesce the item sets and merge (take the union) of the lookahead tokens per each item/production rule. An example of a transformation to LALR(1):



9.2 Abstract syntax tree

The grammar for the abstract syntax tree can be ambiguous (i.e. we can have multiple parse trees for a given sentence in the AST). This allows us to keep the AST grammar simpler.
The AST is built recursively from the parse tree. We define a new type for each kind of AST node (e.g. `Expression`, `Statement`, etc.).
The later phases of the compiler will compute things about the AST nodes in order to do semantic analysis.

Remark 9.1. We could theoretically proceed with just the parse tree from e.g. LR(1) parsing instead of building an AST, but building an AST helps us simplify concepts and reduces the complexity of pattern matching.
E.g. instead of having to match `Term (Factor * Factor)` for an arithemetic expression we can simply match `Expression`.

9.3 Weeding

Weeding permits us to verify invariants that may be easier by traversal of the AST. For example in Java (Joos) a class cannot be both abstract and final. While we can specify this in the grammar and catch it during parsing, it may be easier to simply traverse the AST and verify the two modifiers do not exist underneath a given class declaration.

Definition 9.2 (Weeding)**. Weeding** is the process of checking the language requirements that could have been enforced by grammars but not easily in code.

9.4 Context-sensitive/semantic analysis (middle end)

The next phase of the compiler is to associate identifiers and nodes in the AST with their semantic meaning.
We will first look at **declarations** and and **usages**. Take the following example:

```
1  class A {
2    public int x;
3    public boolean y;
4    public void m(int y) {
5      x = y+1;
6    }
7  }
```

We note that there are declarations A, x, y, m, y (paramaeter). We can map the usages of every variable to their corresponding declarations intuitively.
Things can get even more complicated with subclasses and nested declarations:

```
1  class B extends A {
2    String z;
3    public void m(boolean b) {
4      y = b;
5      {
6        int y = 42;
7        x = y;
8        (new B()) m(y);
9      }
10     {
11       String y = "foo";
12       y = y + z;
13     }
14   }
15 }
```

We note that the declaration of m is separate from the declaration of m in `class A`. We also note the reference of y in the first bracketed scope is to `int y` and not `public boolean y` declared in A.
Finally, note that the usage of m in the first bracketed scope references `public void m` declared in `class A`: we infer this by the fact that its reference to y is a reference to the y declared as `int y`.

10 Februrary 11, 2019

10.1 Name resolution

The goal of name resolution is to **link usages of names with their corresponding declarations**.

Definition 10.1 (Scope)**.** A **scope** is an area of a program where a declaration **has effect** and **is visible**.

Definition 10.2 (Environment)**.** An **environment** is a map from names (strings) to declarations/meanings. During implementation, one can either map the names to the AST declaration node or to some other data structure used for semantic anaylsis.

Note that a *new scope* should correspond to a *new environment*. We form a stack of environments as we traverse through scopes.
How would operations on our environment look as we traverse our AST?

Declaring a name If we find a declaration for a name, we:

1. Search for name in current environment
2. If name already exists, ERROR
3. Else insert name into environment

Name lookup/resolution To resolve a usage:

1. Search innermost environment
2. If not found, search recursively in enclosing environments
3. If not found in any enclosing environments, ERROR

We look at an example where **namespaces** come into play:

Example 10.1. Suppose we have the following code snippet

```
1  int x = 0;
2  int x (int x) { ... }
3  x = x(x);
```

We note that in the third line, the three `x`'s refer to `int x`, `int x (int x)`, and `int x`, respectively.
For the declaration `int x` and `int x (int x)`, we note that they may both be declared within the same scope, howeer their literal identifier collide. We therefore must introduce **namespaces** for variables and methods in *separate environment*. We **syntactically determine** which namespace a declaration belongs to.

In Java (JLS 6.5.1) we have 6 namespaces:

1. package
2. type (class, interface)
3. method
4. expression (variable, parameter, field)
5. package ∪ type

   For example, given `import P.C` and assuming we infer that `P.C` is a class, `P` could either be a package or an (outer) class.

   Outer classes and thus this namespace is not required in Joos.
6. ambiguous

10.2 Name resolution in Java

While the general idea of name resolution can be applied to all languages, in Java specifically we can perform name resolution as follows:

1. Build global environment (set of classes)
2. Resolve type names (type namespace)
3. Check class hierarchy (inheritance)
4. Disambiguate namespaces of ambiguous names (since we have a hierarchy now)
5. Resolve "expressions" (variables, static fields)
6. Type checking
7. Resolve methods and instance fields

Note that in Java (and Joos), we are able to topologically perform the above steps in order. Some other languages have mutual dependency between the above stepsand thus name resolution will need to be performed recursively. Assignment 2 consists of the first 3 steps and assignment 3 consists of the last 4 steps.

10.3 Building the global environment

The global environment should record all class names along with their corresponding package names from the files passed to the compiler for linking.

Remark 10.1. In a normal Java compiler, it is only necessary to specify the single Java file to be compiled and the compiler will automatically resolve imported packages and classes.

10.4 Resolving type names

There are two ways to reference a name:

Qualified names always have `.` in their names (e.g. `a.b.c.d`).

We simply traverse the sequence of names listed starting from the top-level name.

Remark 10.2. If there is a usage `c.d` and a single-type import `a.b.c`, then `a.b.c.d` will never be resolved to `c.d`.

Simple names have no `.` in their names. We traverse the namespaces in the following priority order:

1. Enclosing class/interface
2. Single-type imports (e.g. `import a.b.c`)
3. Type in same package
4. Import-on-demand package (e.g. `import a.b.*`)

Remark 10.3. If there is any ambiguity within any of those namespaces (e.g. two type `c` imported by two single-type imports) then we raise a compile error.

Remark 10.4. The default package (packages without a name e.g. package containing `main`) is **NOT** the root package.

Therefore one would need to prefix the default package with some unique package name to avoid usages referencing the default package in other packages.

10.5 Simple class hierarchy checks (JLS 8,9)

We first verify class declarations. Some simple constraints in JLS 8 and 9:

1. For `class A extends B`, `B` must be a **class** (8.1.3)
2. For `class C implements D`, `D` must be an **inferface** (8.1.4)
3. No duplicate interfaces e.g. `class E implements F, F` (8.1.4)
4. For `class A extends B`, `B` cannot be `final` (8.1.3)
5. Constructors for the same class must have distinct signatures (i.e. parameter types) (8.8.2)

11 February 13, 2019

11.1 Formal constructs for class hierarchy

We attempt to define a formal model for inheritance and the class hierarchy. Consider the following examples:

Example 11.1. Given `class A extends B implements C, D, E` then we define

$$\text{super}(A) = \{B, C, D, E\}$$

which are the **direct superclasses**.

Remark 11.1. If inheritance unspecified (e.g. `class A`) then super(A) = {java.lang.Object}.
Note that super(java.lang.Object) = {}.

Example 11.2. Given `interface F extends G, H, I` we have

$$\text{super}(F) = \{G, H, I\}$$

We now give the formal inference rules. First let's look at types (classes and interfaces):

Definition 11.1 (Subtypes)**.** Let $S < T$ denote that S is a **strict subtype** of T where

$$\frac{T \in \text{super}(S)}{S < T}$$
$$\frac{S < T' \quad T' < T}{S < T}$$

We also define $S \leq T$ as S is a **subtype** of T where

$$\frac{S < T}{S \leq T}$$
$$\frac{}{S \leq S}$$

Definition 11.2 (Super set)**.** We define the **super set** super(T) as all super classes of T i.e. super(T) = $\{S \mid T < S \text{ or } T \leq S\}$.

Definition 11.3 (Declare set)**.** We define the **declare set** declare(T) to be the set of *methods and fields* declared in T.

Remark 11.2.
- interface methods are implicitly `public abstract`
- interface fields are implicitly `static`

Definition 11.4 (Inherit set)**.** We define the **inherit set** $\text{inherit}(T)$ as the *methods and fields* that T inherits.

Definition 11.5 (Contain set)**.** We define the **contain set** $\text{contain}(T) = \text{declare}(T) \cup \text{inherit}(T)$.

Now we look at type methods:

Definition 11.6 (Replace (methods))**.** We say $\text{replace}(m, m')$ if method m overrides method m'.

Definition 11.7 (Replace (fields))**.** We say $\text{replace}(f, f')$ if field f hides field f'.

Definition 11.8 (Signature)**.** We define $\text{sig}(m)$ as method m's signature which consists of its **name** and its **parameter type**.
It does **not** include *return type* or *modifiers*.

We have the following inference rule for **overriding methods in superclasses** (8.4.6):

$$\frac{S \in \text{super}(T) \quad m \in \text{declare}(T) \quad m' \in \text{contain}(S) \quad \text{sig}(m) = \text{sig}(m')}{(m, m') \in \text{replace}}$$

Definition 11.9 (No declaration)**.** We define $\text{nodecl}(T, m)$ as T does not declare method with $\text{sig}(m)$.
That is $\forall m' \in \text{declare}(T), \text{sig}(m') \neq \text{sig}(m)$.

Definition 11.10 (Modifications)**.** We define $\text{mods}(m)$ as the set of method modifiers on m (e.g. `abstract`).

We also have the inference rule for **inheriting non-abstract methods** (8.4.6.4):

$$\frac{S \in \text{super}(T) \quad m \in \text{contain}(S) \quad \text{nodecl}(T, m) \quad \texttt{abstract} \notin \text{mods}(m)}{m \in \text{inherit}(T)}$$

Definition 11.11 (All abstract)**.** We define $\text{allabs}(T, m)$ as:

$$\forall S \in \text{super}(T) \forall m' \in \text{contain}(S), \text{sig}(m') = \text{sig}(m) \Rightarrow \texttt{abstract} \in \text{mods}(m')$$

That is all inherited methods with the same $\text{sig}(m)$ are abstract.

So we have the inference rule for **inheriting abstract methods** (8.4.6.4):

$$\frac{S \in \text{super}(T) \quad m \in \text{contain}(S) \quad \text{nodecl}(T, m) \quad \texttt{abstract} \in \text{mods}(m) \quad \text{allabs}(T, m)}{m \in \text{inherit}(T)}$$

that is: we can only inherit `abstract` methods if there are concrete methods with the same signature. We thus need to define **replacing abstract with concrete methods** (8.4.6.4):

$$\frac{\begin{array}{c} S, S' \in \text{super}(T) \quad m \in \text{contain}(S) \quad m' \in \text{contain}(S') \\ \texttt{abstract} \notin \text{mods}(m) \quad \texttt{abstract} \in \text{mods}(m') \quad \text{sig}(m) = \text{sig}(m') \end{array}}{(m, m') \in \text{replace}}$$

Finally we have the inference rule for **inheriting fields**:

$$\frac{S \in \text{super}(T) \quad f \in \text{contain}(S) \quad \forall f' \in \text{declare}(T), \text{name}(f') \neq \text{name}(f)}{f \in \text{inherit}(T)}$$

A note on interfaces and `java.lang.Object`:

Remark 11.3. An interface with no superinterfaces implicitly declares an `abstract` version of every `public` method in `java.lang.Object` (JLS 9.2).
For example an empty interface `interface I` implicitly has `i.equals(i)`.
So `class C implements I` would implicitly inherit `extends java.lang.Object`.

11.2 Hierarchy checks (JLS, Joos)

We enumerate the class hierarchy checks we need to perform using the notation and definitions we introduced above:

1. No cycles in hierarchy i.e. $\nexists T, T < T$
2. No duplicate methods i.e.
$$\forall m, m' \in \text{declare}(T), m \neq m' \Rightarrow \text{sig}(m) \neq \text{sig}(m')$$
3. One return type per unique signature i.e.
$$\forall m' \in \text{contain}(T), \text{sig}(m) = \text{sig}(m') \Rightarrow \text{type}(m) = \text{type}(m')$$
4. Classes with abstract methods must be abstract i.e.
$$\forall m \in \text{contain}(T), \texttt{abstract} \in \text{mods}(m) \Rightarrow \texttt{abstract} \in \text{mods}(T)$$
5. Static methods can only override static methods i.e.
$$\forall (m, m') \in \text{replace}, \texttt{static} \in \text{mods}(m) \iff \texttt{static} \in \text{mods}(m')$$
Remark 11.4. Note this is an $\iff$ relationship: we cannot override with a non-static method if a static method is inherited *nor* can we override with a static method the inherited method is non-static.
6. Methods can only override methods with the same return type i.e.
$$\forall (m, m') \in \text{replace}, \text{type}(m) = \text{type}(m')$$
7. Only public methods can override public methods i.e.
$$\forall (m, m') \in \text{replace}, \texttt{public} \in \text{mods}(m') \Rightarrow \texttt{public} \in \text{mods}(m)$$
Remark 11.5. We only have a $\Rightarrow$ relationship: this means we may override with a `public` method m with the same signature of m' if m' is **not** `public`.
8. Omitted.
9. We cannot override `final` methods i.e.
$$\forall (m, m') \in \text{replace}, \texttt{final} \notin \text{mods}(m')$$
10. We cannot declare two fields with the same name i.e.
$$\forall f, f' \in \text{declare}(T), f \neq f' \Rightarrow \text{name}(f) \neq \text{name}(f')$$

11.3 Disambiguating namespaces (JLS 6.5.2)

Consider the expression $a_1.a_2.a_3.a_4$. We need to ascertain whether a_i are packages, types, or fields.
Consider also the expression $a_1.a_2.a_3.a_4()$. Then we know a_4 is a method, but we still need to disambiguou the rest of a_is.
Consider the following code snippet:

```
1 class X {
2   X X (X X) {
3     return (X)X.X(X);
4   }
5 }
```

how would the `X`'s be disambiguated? We follow the below procedure.
Given $a_1.....a_n$ proceed left to right and start with a_1:

1. If local variable a_1 is in scope, we use it.

   $a_2, \ldots, a_n$ must be *instance fields*.

2. If field $a_1 \in$ contain(current class) we use it.

   $a_2, \ldots, a_n$ must be *instance fields*.

3. For each k from 1 to n, if $a_1.....a_k$ is a **type** in the *global environment*, then a_{k+1} is a `static` field and $a_{k+2}, \ldots, a_n$ are *instance fields*.

Remark 11.6. If at any step e.g. step 1 we discover a_2 is not an instance field of a_1, we *do not* proceed to later steps and instead throw a compile error.

12 February 25, 2019

12.1 Resolving variables/static fields

In general this is straightforawrd: we simply lookup the variable in the innermost environment and if not found, we search outwards in enclosing scopes.
An issue with Java specifically is **shadowing**. A solution is to create a new environment for each block, but check outer environments when adding a name.
For example considder the following code snippet:

```
1  { int x;
2      {
3          int y;  // OK
4          int z;  // OK?
5      }
6      {
7          int x;  // not allowed
8          int y;  // OK
9      }
10     int z;  // OK?
11 }
```

Note that declaring `int x` again inside a nested scope when `int x` is already declared in an enclosing scope is not allowed. However, re-declaring `int y` in disjoint non-nested scopes is fine.
A solution is to create a *new scope* for every variable declaration (which would end at its semantic scope). For example given:

```
{
  int x;
  y = 3;  // error
  int y;
  y = 5;  // OK
  int z;
}
```

The corresponding program with scopes created for each new variable declaration would look something like:

```
{
  int x;
  y = 3;  // error
  {
    int y;
    y = 5;  // OK
    {
        int z;
    }
  }
}
```

12.2 Type checking

We provide two different perspectives/definitions of a type:

Definition 12.1 (Type (definition 1))**.** A **type** is a set of values (with operations on them).

Definition 12.2 (Type (definition 2))**.** A **type** is a way to interpret bit sequences.

We now differentiate between two categories of types:

Definition 12.3 (Static type)**.** The **static type** of an expression E is a set containing all possible values of E.

Definition 12.4 (Dynamic type)**.** The **dynamic type** is a runtime value that indicates how to interpret some of the other bits.

Definition 12.5 (Declared type)**.** The **declared type** of a variable is an assertion that the variable will only contain values of that type.

We now distinguish between two different ways we go about type checking:

Definition 12.6 (Static type checking)**.** Prove (using mathematical inference/deduction) that every expression will evaluate to a value in its type.

Definition 12.7 (Dynamic type checking)**.** Runtime check that the dynamic type (tag) is declared in the variable to which it is assigned.

A static type checker does two things:

1. Infers a type for each sub-expression
2. Check that expressions are used correctly with operations (e.g. `1 + true` is an error).

Definition 12.8 (Type correct)**.** A program is **type correct** if type assertions hold in all executions.

Remark 12.1. Note that a naive definition of type correctness is undeciable. Consider the following program:

```
1  int x;
2  if(program halts) {
3    x = true;
4  }
```

Note that if the program halts then `x = true` makes the entire program type incorrect, otherwise the program is indeed type correct since `x = true` is not evaluated.
Therefore we could reduce the halting problem to this definition of type correctness.

Instead we have a more refined definition of type correctness:

Definition 12.9 (Statically type correct)**.** A program is **statically type correct** if it obeys a system of type inference rules (**type system**).

Definition 12.10 (Soundness)**.** A type system is **sound** if statically type correct $\Rightarrow$ type correct.

12.3 Introduction to type system for Joos

We introduce some notation:

$$C, L, \sigma \vdash E : \tau$$

In class C, local environment L, if the current method has return type σ, then expression E has tpe τ.
Also:

$$C, L, \sigma \vdash S$$

means statement S is statically type correct.
We now introduce the inference rules:

Literals Literals are straightforward. For example integer literals have the rule:

$$\frac{}{C, L, \sigma \vdash 42 : \text{ int}}$$

The other literals:

$$\frac{}{\texttt{true} : \text{ boolean}} \qquad \frac{}{\text{“abc”} : \text{ string}}$$

$$\frac{}{'a' : \text{ char}} \qquad \frac{}{\texttt{null} : \text{ null}}$$

Note “null” is a special type with only the `null` literal.

Operations The *not* operator:

$$\frac{E : \text{ boolean}}{!E : \text{ boolean}}$$

We first define the *numeric type class* as:

$$num(\sigma) = \sigma \in \{\text{int, short, char, bytes}\}$$

Then the addition operation is:

$$\frac{E_1 : \tau_1 \quad E_2 : \tau_2 \quad num(\tau_1) \quad num(\tau_2)}{E_1 + E_2 : \text{ int}}$$

$$\frac{E_1 : \text{ string} \quad E_2 : \tau_2 \quad \tau_2 \neq \text{ void}}{E_1 + E_2 : \text{ string} \quad E_2 + E_1 : \text{ string}}$$

12.4 Instance fields/methods

After type-checking, every sub-expression has a type. Consider the following two usages:

```
1 a.b        // field access
2 a.b(c)     // method access
```

We know the type of `a` from type-checking, thus for the first usage we can simply lookup the field `b` in type `a`, and for the second usage we can look up for a method `b` in type `a` (we check the contains set).
If `b` is an *overloaded* method, we disambiguate based on the arguments `c`.

13 February 27, 2019

13.1 Pseudocode for type checking

We have the following pseudocode for type checking an AST node E:

1. Find an inference rule of the form:
$$\frac{\text{premises}}{C, L\sigma \vdash E : \tau}$$
2. Check premises
3. Return τ if premises are satisfied

We repeat the above for all inference rules for E until either we find one whose premises are satisfied or we raise a type/compile error.

13.2 More inference rules for type system of Joos

For **local variable usages** we have

$$\frac{L(n) = \tau}{C, L, \sigma \vdash n : \tau}$$

$$\frac{}{C, L, \sigma \vdash \texttt{this} : C}$$

where $L(n)$ is the type (if it exists) of a variable.
For **statements**, note that we do not assign it a type in Java:

$$\frac{C, L, \sigma \vdash E : \text{boolean} \quad C, L, \sigma \vdash S}{C, L, \sigma \vdash \texttt{if(E) S}}$$

For a **block of statements** we have:

$$\frac{\forall i \vdash S_i}{\vdash \{S_1; S_2; \ldots S_n;\}}$$

To handle **variable declarations**, we have the rule:

$$\frac{C, L[n \to \tau], \sigma \vdash S}{C, L, \sigma \vdash \{\tau n; S\}}$$

For **assignments**, we have:

$$\frac{C, L, \sigma \vdash E : \tau_2 \quad C, L, \sigma \vdash L(n) = \tau_1 \quad \tau_1 := \tau_2}{C, L, \sigma \vdash n = E : \tau_1}$$

where the $:=$ is a special relation known as **assignability (JLS 5)**, which has the inference rules (for Joos specifically):

$$\frac{}{\tau := \tau}$$

$$\frac{}{\text{int} := \text{short}}$$

$$\frac{}{\text{short} := \text{byte}}$$

$$\frac{}{\text{int} := \text{char}}$$

$$\frac{D \leq C}{C := D}$$ assignment to superclass

$$\frac{\sigma := \tau \quad \tau := \rho}{\sigma := \rho}$$ transitivity

$$\frac{}{C := \text{null}}$$

For **fields** we have the rules:

$$\frac{\vdash \texttt{static}\ \tau f \in \text{contains}(D)}{\vdash D.f : \tau}$$

$$\frac{\vdash \texttt{static} \notin \text{mods}(f) \quad E : D \quad \tau f \in \text{contains}(D)}{\vdash E.f : \tau}$$

$$\frac{\vdash E : \tau_2 \quad \texttt{static}\ \tau_1 f \in \text{contains}(D) \quad \tau_1 := \tau_2}{\vdash D.f = E : \tau_1}$$

$$\frac{\vdash E_1 : D \quad E_2 : \tau_1 \quad \texttt{static} \notin \text{mods}(f) \quad \tau_2 f \in \text{contains}(D) \quad \tau_2 := \tau_1}{\vdash E_1.f = E_2 : \tau_2}$$

For **comparison** we have (note these rules hold for both `==` and `!=`)

$$\frac{\vdash E_1 : \tau_1 \quad E_2 : \tau_2 \quad \text{num}(\tau_1) \quad \text{num}(\tau_2)}{\vdash E_1 == E_2 : \text{boolean}}$$

$$\frac{\vdash E_1 : \tau_1 \quad E_2 : \tau_2 \quad \tau_1 := \tau_2 \vee \tau_2 := \tau_1}{\vdash E_1 == E_2 : \text{boolean}}$$

For **casting**, we have two ways to cast:

$$\frac{\vdash E_1 : \tau_1 \quad \tau_1 := \tau_2 \vee \tau_2 := \tau_1}{\vdash (\tau_2)E : \tau_2}$$

Note $\tau_2 := \tau_1$ corresponds to **upcasting** (*always succeeds*) whereas $\tau_1 := \tau_2$ corresponds to **downcasting** (*may fail, needs runtime check*).

For numeric types:

$$\frac{\vdash E_1 : \tau_1 \quad \text{num}(\tau_1) \quad \text{num}(\tau_2)}{\vdash (\tau_2)E_1 : \tau_2}$$

Finally we have **runtime checks** for the expression `instance of`:

$$\frac{\vdash E : \tau \quad \tau := D \vee D := \tau}{E \ \texttt{instance of} \ D : \text{boolean}}$$

Remark 13.1. Note that if there is no assignability between τ_1, τ_2 then it *always fails* **except for interfaces**.

For **methods invocations**:

$$\frac{\vdash E : D \quad \forall i \vdash E_i : \tau_i \quad \tau m(\tau_1, \ldots, \tau_n) \in \text{contains}(D)}{\vdash E.m(E_1, \ldots, E_n) : \tau}$$

Remark 13.2. We do not have implicit casting of arguments with assignability semantics in Joos.
In Java there are specific rules for determining the "maximally specific" method declaration since the method may be overloaded (see 15.12.2).
One can simply use upcasts in Joos to work around this.

For **method returns**:

$$\frac{C, L, \sigma \vdash E : \tau \quad \sigma := \tau}{C, L, \sigma \vdash \texttt{return} \ E}$$

$$\frac{}{C, L, \text{void} \vdash \texttt{return}}$$

For **arrays**:

$$\frac{\vdash E_1 : \tau_1[] \quad E_2 : \tau_2 \quad \text{num}(\tau_2)}{\vdash E_1[E_2] : \tau_1}$$

$$\frac{\vdash E_1 : \tau_1[] \quad E_2 : \tau_2 \quad \text{num}(\tau_2) \quad E_3 : \tau_3 \quad \tau_1 := \tau_3}{\vdash E_1[E_2] = E_3 : \tau_1}$$

$$\frac{\vdash E : \tau[]}{\vdash \texttt{E.length} : \text{int}}$$

Note that **array assignability (JLS 5)** has its own rules for the standard library types:

$$\frac{}{\texttt{Object} := \sigma[]}$$

$$\frac{}{\texttt{Cloneable} := \sigma[]}$$

$$\frac{}{\texttt{java.io.Serializable} := \sigma[]}$$

We also have the following rule for **multidimensional arrays**:

$$\frac{\sigma[] := \tau[]}{\sigma[][] := \tau[][]}$$

14 March 4, 2019

14.1 Potential issues with arrays

Recall that we have the inference rules

$$\frac{D \leq C}{C[] := D[]}$$
$$\frac{D \leq C}{C := D}$$

Now consider the following code fragment:

```
1   Apple[] as = new Apple[1];
2   Fruit[] fs = as;
3   fs[0] = new Orange();
4   Apple a = as[0];
```

Note that `as[0]` now points to an `Orange`, but we've explicitly typed `a` as an `Apple`. This is an **unsound** program.

Remark 14.1. In Java, the above code statically type checks and compiles but the assignment `fs[0] = new Orange()` will throw a runtime error.
It does this by marking the initial `as` allocated array as type `Apple[]` (dynamic type tag) and checking it during assignment.

Remark 14.2. Unsound programs are possible even if they statically type check.
Formally, we have statically type correct $\subseteq$ type correct $\subseteq$ all programs where unsound programs can occur at all levels.

How do we deal with this? To preserve type safety, we must check the dynamic type tag of an array at every array write (JLS 10.10) and raise an `ArrayStoreException` if a violation occurs.

14.2 Static program analysis

Consider another "unsound" behaviour with casts:

```
1 Orange o = (Orange) new Apple();
```

As a programmer there may be instances where we want to intentionally introduce unsoundness e.g. with the above cast.

Remark 14.3. The above cast raises a `ClassCastException` runtime error in Java.

In **static program analysis**, we want to prove *properties* of run-time behaivour without actually running the program.
Some applications:

- Prevent bugs
- Generate efficient code
- Inform programmer

Some properties we can derive:

- Constant expression

- Method execution always ends with return statement
- Will current value of variable ever be read?
- Will a statement ever be executed?
- Read/write dependencies
- Will program terminate or loop infinitely? (in general this is undecidable but we can still do this analysis for some programs)
- Is an array access in bounds?

Example 14.1. Suppose we wanted to see if the following outputs "Hello world":

```
1  main() {
2    if(...) { ... }
3    printf("Hello world");
4  }
```

Suppose we wanted to write an assembler that given a program f, decides whether the output is "Hello world". Clearly this is undecidable in general.

Theorem 14.1 (Rice's theorem)**.** Let R be any **non-trivial** property of the output of the program.
Given program P, it is **undecidable** whether P has property R.
We define *non-trivial* as $\exists P \in R$ and $\exists P \notin R$.

We can however give a **conservative approximation**.

Definition 14.1 (Conservative analysis)**.** An analysis is **conservative** if its result is never untrue.
For example an analysis outputting **maybe** and **no** is conservative (assuming it's sound).

Definition 14.2 (More precise analysis)**.** A more **precise** analysis gives definitive answers for more programs.
For example an analysis outputting **maybe**, **no** and **yes** is more precise than a conservative one (assuming it's sound).

Java requires **reachability analysis** and **definite assignment** (every local variable must be written to before it is read).

14.3 Java reachability analysis (JLS 14.20)

Java specifies specific reachability analysis that must be performed:

Algorithm 11 Java reachability analysis

1: Define: in[s] - can statement s start executing? (no/maybe)
2: Define: out[s] - can statement s finish executing? (no/maybe)
3: Error if in[s] = no for any s
4: Define out[return] = no
5: Error if out[method body] = maybe

We define $in[s]$ and $out[s]$ for every AST node recursively.

For `if` statements we have `L : if (E) S` where:

$$in[S] = in[L]$$
$$out[L] = out[S] \vee in[L] = in[L] \qquad out[S] \Rightarrow in[S] \Rightarrow in[L]$$

where $in[S] = in[L]$ since to execute S, regardless of whether E evaluates to `true` or `false` we require at least $in[L]$ and we define

$$\text{maybe} \vee \text{maybe} = \text{maybe}$$
$$\text{no} \vee \text{maybe} = \text{maybe}$$
$$\text{no} \vee \text{no} = \text{no}$$

(this holds only for this specific analysis: one will need to define this merge $\vee$ specifically for a given analysis). For `if-else` statements we have `L : if (E) S1 else S2` where:

$$in[S1] = in[L]$$
$$in[S2] = in[L]$$
$$out[L] = out[S1] \vee out[S2]$$