LR(1) Parsing

- LR(0) parsers may generate
  - shift-reduce conflicts (both actions possible in same configuration set)
  - reduce-reduce conflicts (two or more reduce actions possible in same configuration set)

- The configurations for LR(1) are extended to include a lookahead symbol

  \[ A \rightarrow X_1 \ldots X_i \ldots X_j \text{, } \lambda \in \{V_t \cup \{\lambda}\} \]

  Configurations that differ only in the lookahead symbol are combined:

  \[ A \rightarrow X_1 \ldots X_i \ldots X_j \text{, } \{l_1 \ldots l_m\} \]

Configuration Set Closure for LR(1)

\[
\text{closure}_1(\{S \rightarrow \cdot E$, (l)\}) =
\begin{cases}
S \rightarrow \cdot E$, (l)
E \rightarrow \cdot E + T \text{, } (l)
E \rightarrow \cdot T \text{, } (\lambda)
T \rightarrow \cdot T \text{, } (\lambda)
T \rightarrow \cdot T \text{, } (+)
E \rightarrow \cdot T \text{, } (+)
T \rightarrow \cdot \text{ID} \text{, } (+)
T \rightarrow \cdot \text{ID} \text{, } (+)
T \rightarrow \cdot \text{ID} \text{, } (+)
\end{cases}
\]

Goto and Action Table for LR(1)

- The function goto1(configuration-set,symbol) is analogous to goto0() for LR(0)

- Goto table is also created the same way as for LR(0).

- The Action table makes the difference. The lookahead symbol is used to decide if a reduction is applicable. Hence, the lookahead symbol resolves possible shift-reduce conflicts.

Example: LR(1) for G3

- Exercise:
  - create states and the goto table
  - create the action table
  - explain how you see that this is LR(1) and not LR(0)
  - Hint – look at state 7

Problems with LR(1) Parsers

LR(1) parsers are very powerful. However,

- The table size can grow by a factor of \(|V_t|\)
- Storage-efficient representations are an important issue.

Example: Algol 60 (a simple language) includes several thousand states.
Solutions to the LR(1) Size Problem

Several parser schemes similar to LR(1) have been proposed:
- LALR: merge certain states. There are several LR optimization techniques (will not be discussed further).
- SLR (simple LR): build a CFSM for LR(0) then add lookahead. Lookahead symbols are taken from the Follow sets of a production.

Exercise

- Determine if G3 is an SLR Grammar:
  Hint: the states 7 and 11 have shift-reduce conflicts. Can they be resolved by looking at the Follow set?
  Follow(E) = { $, +, \)}
  Shift is on "*"
  Can distinguish between the shift/reduce action

We have covered ...

- Scanners, scanner generators
- Parsers:
  - Parser terminology
  - LL(1) parsing and parser generation: building stack-based parsers, including action symbols.
  - Overview of LR parsers: shift-reduce parsers.
  CFSM. Basics of LR(1).

Semantic Processing

Properties of 1-Pass Compilers

- efficient
- coordination and communication of passes not an issue
- single traversal of source program restricts semantics checks and actions.
- no (or little) code optimization (peephole optimization can be added as a separate pass)
- difficult to retarget, architecture-dependent. Architecture-dependent and independent decisions are mixed.
1-Pass Analysis + 1-Code Generation Pass

- More machine independent
- Can add optimization pass
- There is an intermediate representation (IR) that represents the analyzed program. It is input to the code generator.
- Each pass can now be exchanged independently of each other

Note: The use of “analysis” in this context is different than in optimization papers.

Multi-Pass Analysis

- Scanner can be a separate pass, writing a stream (file) of tokens.
- Parser can be a separate pass writing a stream of semantic actions.
- Analysis is very important in all optimizing compilers and in programming tools.
- Advantages of Multi-Pass Analysis:
  - Can handle Languages with variable declarations (need multi-pass analysis for static semantics checking)
  - No “forward declarations” necessary
  - For memory bound applications, can make very small compilers this way.

Multi-Pass Synthesis

We view a compiler as performing two major tasks.

- Analysis: understanding syntax and semantics of the source program
- Synthesis: generating the output (usually the target code)

- Simple multi-pass synthesis: code-generation + peephole optimization
- Several optimization passes can be added
- Split into machine independent and dependent code generation phases is desirable
- Importance of early multi-pass compilers: space savings.

Families of Compilers

- Compilers that can understand multiple languages.
  - Syntax analysis has to be different.
  - Some program analysis passes are generic.
  - The choice of IR influences the range of analyzable languages.
- Compilers that generate code for multiple architectures.
  - Analysis and architecture-independent code generation can be the same for all machines.
  - Example: GNU C compiler. GCC uses two IRs: a tree-oriented IR and RTL.

Families of Compilers

- Within a given part of the compiler (e.g., the frontend (analysis) part of the compiler, or the backend (synthesis) part, the IR may be transformed into a form more appropriate to the job to be done
  - IBM w-code uses stack based language for its IR
    - Each pass translates from w-code into a pass-specific IR
    - Passes use (fairly complicated) trees of expressions, lists of statements, etc.
    - Passes may be written in different languages