Recusive-Descent Parsing

Translating EBNF to BNF

Replace optional construct $[S]$ by a new nonterminal $OptS$ where $OptS \to S|\epsilon$.
Replace grouping construct (S) by a new nonterminal $GrpS$ where $GrpS \to S$.
Replace repetition construct $\{S\}$ by a new nonterminal $RepS$ where $RepS \to SRepF|\epsilon$.

Left Factoring & Left Recursion

Left Factoring Productions

Left Factor Rewriting Rule

To remove the left factor from
$$A \to \alpha\ \beta\ |\ \alpha\ \gamma,$$
we can rewrite the production using an aditional nonterminal A' as
$$A \to \alpha\ A' \qquad A' \to \beta\ |\ \gamma$$

Left Recursive Productions

Immediate Left Recursion Rewriting Rule (Simple)

To remove the left recursion from
$$A \to A\ \alpha\ |\ \beta,$$
we can rewrite the production as
$$A \to \beta\ A' \qquad A' \to \epsilon\ |\ \alpha\ A'$$

Immediate Left Recursion Rewriting Rule (General)

To remove the left recursion from the general case
$$A \to A\alpha_1|A\alpha_2|\dots|A\alpha_n|\beta_1|\beta_2|\dots|\beta_m,$$
we can use grouping to see the structure in the same form as the simple case
$$A \to A(\alpha_1|\alpha_2|\dots|\alpha_n)|(\beta_1|\beta_2|\dots|\beta_m)$$
which allows us to rewrite the production as follows,
$$A \to (\beta_1|\beta_2|\dots|\beta_m)A'$$
$$A' \to \epsilon|(\alpha_1|\alpha_2|\dots|\alpha_n)A'$$

Indirect Left Recursion Rewriting Rule

The following productions have indirect recursion from $A \to B \to C \to A$.
$$A \to B\ \alpha \qquad B \to C\ \beta \qquad C \to A\ \gamma_1\ |\ \gamma_2$$
To remove the recursion we can first collapse B into A as follows.
$$A \to C\ \beta\ \alpha \qquad C \to A\ \gamma_1\ |\ \gamma_2$$
Then we can collapse C into A.
$$A \to (A\ \gamma_1\ |\ \gamma_2)\ \beta\ \alpha$$
$$\downarrow$$
$$A \to A\ \gamma_1\ \beta\ \alpha\ |\ \gamma_2\ \beta\ \alpha$$

This now leaves a simple direct left recursion which we can remove as follows.
$$A \to \gamma_2\ \beta\ \alpha\ A' \qquad A' \to \gamma_1\ \beta\ \alpha\ A'$$

First & Follow Sets

LL(1) Grammar

A BNF grammer is LL(1) if for each nonterminal, N, wher $N \to \alpha_1|\alpha_2|\dots|\alpha_n$,

- the First sets for each pair of alternatives for N are disjoint, and
- if N is nullable, $First(N)$ and $Follow(N)$ are disjoin.

Bottom Up Parsing

Shift/Reduce Parsing

Example Table

$$S \to A \qquad A \to (\ A\ ) \qquad A \to a$$

Parsing stack	Input	Parsing action
\$0	((a))\$	shift
\$0(2	(a))\$	shift
\$0(2(2	a))\$	shift
\$0(2(2a3	))\$	reduce $A \to a$
\$0(2(2A4	))\$	shift
\$0(2(2A4)5	)\$	reduce $A \to (A)$
\$0(2A4	)\$	shift
\$0(2A4)5	\$	reduce $A \to (A)$
\$0A1	\$	accept

Derived Items

If a state has an LR(1) item of the form
$$[N \to \alpha \bullet M\beta, T]$$
where the nonterminal M has productions
$$M \to \alpha_1|\alpha_2|\dots|\alpha_m$$
& $T = \{a_1, a_2, \dots, a_n\}$, then the state also includes the derived items
$$[N \to \bullet\alpha_1, T]\dots[N \to \bullet\alpha_m, T]$$
where if β is not nullable $T' = First(\beta)$ & if β is nullable, $T' = First(\beta) - \{\epsilon\} \cup T$.

Runtime Environment

Stack Organisation

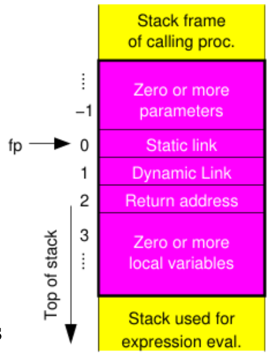
Definitions

- **stack pointer -** contains the address for the top of the stack + 1
- **stack limit -** contains the address of the upper limit for the stack & the bottom of the heap
- **frame pointer -** contains the address of teh stack frame for the current procedure
- **program counter -** contains the address of the next machine instruction

Procedures

A pointer to the current procedures' frame is stored in a globally known location.

Calling a Procedure


Stack frame of calling proc.
Zero or more parameters
Static link
Dynamic Link
Return address
Zero or more local variables
Stack used for expression eval.
fp
–1
0
1
2
3
Top of stack

- parameters to the procedure are pushed to the stack
- a static link is pushed onto the stack
- the current frame pointer is pushed onto the stack to create the dynamic link
- the frame pointer is set so that it contains the address of the start of the new stack frame
- the current value of the program counter is pushed onto the stack to form the return address
- the program counter is set to the address of the procedure
- space is allocated on the stack for any local variables

Returning From a Procedure

- the program counter is set to the return address in the current activation record
- the frame pointer is set to the dynamic link
- the stack pointer is set so that all the space used by the stack frame (but not parameters) is popped from the stack
- execution continues at the instruction addressed by the (restored) program counter
- after return, the calling procedure h&les deallocating any parameters

Local Variables

Local variables are stored within the procedures frame. They are accessed by an offset relative to the frame pointer.

Non-local Variables

We continuesly access the static link of the enclosing procedures until we are in the procedure in which n is defined. We can then add the offset to the variable n to get the final address of the non-local variable.

Parameters

- **Formal parameters** are the paremeters used in the declaration of the procedure in its header.
- **Actual parameters** are the actual parameters passed to a procedure on a call.

Call-by-value — The formal parameters are variables which are assigned a copy of the value of the actual parameter.
Call-by-const — The formal parameter acts as a read-only local variable that is assigned the value of the actual parameter expression.
Call-by-result — The formal parameter acts as a local variable whose final value is assigned to the actual parameter variable.
Call-by-value-result — A single parameter acts as both a value & a result parameter.
Call-by-reference — The formal parameter is the address of the actual parameter variable. All references to the formal parameter are applied to the actual parameter variable immediately.
Call-by-sharing — The same as call-by-value, but what is passed is a reference to an object (e.g. Java) rather than the values of the object.
Call-by-name — The actual parameter expression is evaluated every time the formal parameter is accessed.
Passing Procedures as Parameters — The address of the procedure as well as the static link for the procedure's environment is passed.

Function Results

To ensure the returned value is before the stack frame, free space is allocataed for the result before the parameters are loaded onto the stack & the frame is set up.
Returning Procedures — Return the address of the procedure as well as the static link for the procedures environment. This requires the environment of the returned procedure to be maintained which makes the simple stack-based allocation of frames insufficient.

Variable Aliasing

Parameter Aliasing — In languages with call-by-reference, the same variable can be passed to two (or more) different parameters leading to variable aliasing.
Global Variable Aliasing — If a variable is passed as a reference parameter to a procedure that can access the same variable as a global variabel, then within the procedure there are two aliases for the same variable.

Pointer Aliasing

Parameter Aliasing — In languages with call by sharing, the same reference to an object can be passed to two different

parameters leading to one having two aliases for the same reference.
Global Variable Aliasing — If a reference that is passed as a parameter to a procedure is also directly accessible as a global variable from the procedure. this leads to the procedure having two aliases for the one reference.

Objects

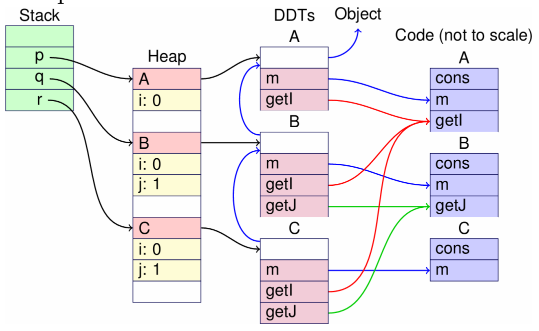
Each field in an object is at a fixed offset from the start of every instance in the heap.

Subclassing

Aditional fields follow the inherited fields & have fixed offsets from the start. Fields that are redeclared replace the previously declared field at the same offset.

Dynamic Dispatch Table

- it has an entry for every method of the class (including inherited methods)
- the entry gives the address of the code for the method
- the entries are at a fixed offset from the start of the dynamic dispatch table
- the entries for inherited & overridden methods are at the same offsets as in the superclass



This

this is passed as an additional implicit parameter to each method.

Static Fields & Methods

Static fields & methods are resolved statically (at compile time). Static methods are not called on an object & hence they do not have an object reference as an implicit this parameter.

Heap Organisation

Issues w/ Explicit Deallocation

- **Dangling references** — an object can be freed via one reference but still be accessible via other references.
- **Memory leaks** — all references to an object are removed but the object was not freed.
- **Memory fragmentation** — the memory consists of alternating allocated & free areas, with no large area of free memory.
- **Locality of reference** — how spread out the memory is in the address space.

Garbage Collection

An object is accessible if it can be reached either

- directly via a reference in a global or local variable, or
- indirectly via a reference in an object that is accessible.

Garbage collection determined which objects are not accessible & recovers the space used by them.

Mark-&-sweep

- a phase that **marks** all the accessible objects
- a phase that **sweeps** up the objects left unmarked & adds them to the free list
- quick since it only needs to mark memory as free (no moving required)
- results in fragmentation since it doesn't compact memory
- can use a lot of memory if heavily fragmented with small free areas

Stop-&-copy

- divides the available memory into two spaces
- memory is allocated sequentially from one space until it runs out
- garbage collection consists of relocating all accessible objects from the first space to the second space
- slow since it must copy all words from one heap to the other
- operation compacts memory so low fragmentation
- uses double the available memory since 2 identically sized heaps are needed

Overcomes memory fragmentation problems, but copying can be expensive in time.

Generational Schemes

Generational schemes use a scheme similar to the stop-&-copy scheme, but make us of more spaces

- the spaces are organised based on the length of time its objects have survived
- older objects are migrated to an old object space & newer objects go in the a new object space
- the newer the space, the more frequently it is garabage collected
- more efficient since avoids repeatedly copying old objects
- memory fragmentation low since objects are compacted when moved between generations
- uses high-ish memory utilisation since multiple heaps are needed (older heaps can use less memory)

Issues w/ Garbage Collection

- Garbage collection has a greater time overhead than explicit deallocation
- Response time can vary because garbage collection can happen at any time & is time expensive
- Real-time response hard to guarantee

On-the-fly

- Garbage collection process takes place incrementally, interleaed with execution of the program.
- Each time an object is allocated, the garbage collector can do a bit of work.

Concurrent

- The garbage collector runs concurrently with the program on a separate processor.

Regular Expressions

Deterministic Finite Automaton (DFA)

A DFA, D, consists of

- A finite alphabet of symbols, Σ
- A finite set of states, S
- A transition function, $T : \Sigma \times S \to S$, which maps an (input) symbol & a (current) state to the (next) state; the function T may not be defined for all pairs of symbols & state.
- A start state, s_0
- A set of final (or accepting) states, F

NFA

From an Regex to a NFA

In the translation from a regular expression, r, to an NFA, the generated NFA has a few properties that do not necessarily hold for an arbitrary NFA (i.e. one not generated from a regular expression).

- The NFA has a single final (accepting) state.
- The initial state of the NFA only has outgoing transitions.
- The final state only has incoming transitions.

The translation rules preserve these properties.

From NFA to DFA

A DFA cannot have

- more than one transition leaving a state on the same symbol
- any empty transitions

An NFA N can be translated to an equivalent DFA D.

- equivalent in the sense that they accept the same languages, i.e., $\mathcal{L}(N) = \mathcal{L}(D)$.

From NFA to DFA

An NFA is transformed to a DFA in which the labels on the states of the DFA are sets of states from the NFA. The sets of states that label a DFA state are formed by collecting all the states that can be reached from NFA states by empty transitions.

Empty Closure of a state

Empty Closure of a state The empty closure of a state x in an NFA N, ϵ-closure(x, N), is the set of states in N that are reachable from x via any number of empty transitions

Empty Closure of a set of states

The empty closure of a set of states X in an NFA N, ϵ-closure(X, N), is the set of states in N that are reachable from any of the states in X via any number of empty transitions.

From NFA to DFA

The following process is repeated until there are no unmarked DFA states left:

- An unmarked DFA state S is selected (the first one is S_0).
- For each symbol a,
  - we consider the set of states that can be reached from any state in S by a transition on a; call this set of states X.
  - If X is nonempty, we add a new state to the DFA labeled with $X' = \epsilon$-closure(X, N), unless a state with that label already exists, in which case it is reused.
  - A transition from S to X' on a is added to the DFA.
- The state S is marked as having been processed.

Minimising a DFA

To minimise the DFA we merge states that have the same transition to the equivalent states. For the example, A, B & C are equivalent.